Java, concisely

Adrian Johnstone Elizabeth Scott

October 16, 2017

Department of Computer Science
Egham, Surrey TW20 0EX, England

Contents

1 Programs

A program is a list of instructions for completing a task

Many of the things we do in life can be thought of as programs. For instance, a recipe is a set of instructions for constructing a meal using food ingredients and the operations supported by a kitchen: chopping, boiling, baking and so on. Of course, we need a human cook to interpret the instructions.

The travel instructions provided by route planning software are also a sort of program. It's a pretty high level kind of program: it tells us where to turn and how far to go but we wouldn't expect the route planner to tell us when to change gear. Sheet music is a program too: the dots on the stave tell the musician which notes to play, and for how long.

1.1 The nature of programs

Let's take a closer look at a food-constructing program. Here's a recipe for making a Victoria sponge.

Ingredients

110g soft butter
110g caster sugar
110g sifted self-raising flour
2 large eggs
vanilla essence

Method

1. Heat oven to 170°C;
2. cream butter and sugar in bowl;
3. beat eggs;
4. add eggs to bowl one teaspoon at a time, beating after each addition;
5. if curdled then abandon cake; else carry on;
6. stir in a few drops of vanilla essence;
7. sift flour into bowl, 25g at a time, folding in after each addition;
8. if consistency is stiff, add 1–2 teaspoons of hot water;

9. divide into baking tins;
10. bake for 25–30 minutes until cake is springy.

In principle, at least, you could build a robot that could perform the task of making a Victoria sponge cake using these instructions. Just to show what kind of thing your colleagues have achieved, please have a look at

`http://video.google.com/videoplay?docid=2056850841271245254&hl=en`

which shows a video of a robot constructed by Nguyen An Khuong to solve the Rubik's Cube puzzle.

Now, we want to draw your attention to several features of the food-constructing program that we'll see again when we look at computer programs: *portability, data, methods* and *control flow*.

1.1.1 Portability

The Victoria sponge program is really not very detailed. Just as the route finding software doesn't tell us when to change gear, the recipe does not tell us where to find our baking tins or how to manipulate the controls on our cooker. To do that would be to make the program too specific to one person's kitchen. Instead, we set the level of the instructions so that any reasonable kitchen user can fill in the gaps. This allows the food-constructing program to be followed by almost anybody. In computing-speak, we say that the program is **portable** between different users' kitchens. Sometimes it is not easy to write programs that are portable. Heston Blumenthal produces food using, amongst other things, liquid nitrogen. Nitrogen liquefaction is not a feature supported by many kitchens, so some of Heston Blumenthal's food construction programs are not very portable.

1.1.2 Data

We begin the Victoria sponge program with a list of the things that we are going to manipulate (the ingredients). In programming language-speak, we call these ingredients **data** and the statements of what we're going to use **declarations**. Now, in the recipe we've limited ourselves to listing the food elements only. Actually we also need a bowl, a whisk and a cooker but we've just assumed that these things will be to hand. In computer programs we also usually assume that things like a keyboard, a screen and a disk drive will be available to us, but every last item of data needs to be described in detail. This can make programs intimidatingly clerical until you acquire the discipline of carefully listing what you need.

1.1.3 Methods

Some of the technical terms used in the program might not be familiar to a novice cook. When I first started making cakes, I had to find out what it meant

to cream butter and sugar and to fold. Elsewhere in my recipe book I found descriptions of a whole set of basic cooking methods such as how to make a reduction, how to clarify butter and so on. A lot of the recipes made use of these basic methods. In computer programs, we often make use of **libraries** of **methods** which by themselves don't do much, but which can be composed together into useful programs. Actually, it turns out that even this whole Victoria sponge food-construction program is really only a sub-program or method: in reality I use the same recipe to make (i) a big sponge which I fill with cream and jam, and (ii) lots of little cakes which I've added chocolate chips to.

When we write computer programs, we tend to try to limit each method to around a page or so, starting with very simple ones that are then joined together within higher level methods and so on until we get the whole program going. It is *really* important to try to keep to the discipline in which the individual methods are small, and often form a **hierarchy**. Complete computer programs are often very large and complex: the hierarchy enables us to focus on one small part of the system at a time. Wherever possible, keep things simple; keep them tidy; keep then compact.

1.1.4 Control flow

The last feature we want you to think about is the sequencing of the instructions, or the **control flow**. At first glance, it looks like the recipe has just ten steps, to be followed one after another. We've already seen that Step 2 involved a sub-method: the first time we made a cake we would need to stop, turn to the page of the recipe book that explained what to do, and then come back and continue. In computer programming, this is known as **calling** a method or function.

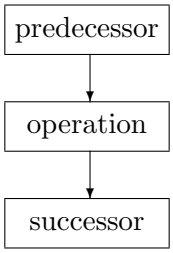
Step 8 contains a **conditional** action. We have to look at what we've done, and then add some water only if we need to. One of the distinguishing features of computers is their ability to make decisions on the basis of previous actions.

You can see another conditional action at Step 5. It turns out, that if you add the eggs too quickly you get a slimy lumpy mixture instead of a nice fluffy one. A mixture like this has *curdled*, and it won't cook into a nice sponge cake so you may as well throw it away and start again: abandon cake. In computer programs, we call these kinds of events **exceptions**. A classic example is when we try to divide a number by zero, an operation with an undefined result. Left to itself, a computer will usually simply stop a program that does something like this: that's what is really happening when a program crashes. Something happened that shouldn't have done, because the programmer wasn't careful enough to consider all possibilities and guard against them. Handling exceptions cleanly is an important part of engineering robust computer applications.

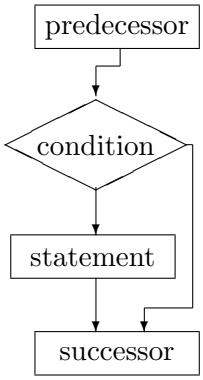
The conditional actions act like a switch, enabling or disabling instructions. Steps 4, 7 and 10 are repeating actions. These tell us to keep on doing something until some condition is met: adding the eggs; adding the flour and baking the cake respectively. One way of thinking about these things is that they are conditional actions that chase their tails: you test a condition; do something; and then go back and test the condition again.

It turns out that we really only use four different things to construct all of the different control flows we need: *sequencing*, *selection*, *iteration* and *calling*.

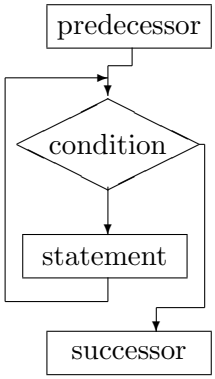
A *sequence* is a list of actions that will always be performed one after another.

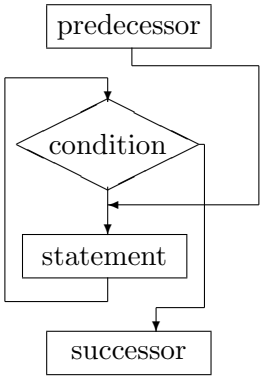


A *selection* involves making a test, and then performing one of a set of actions depending on the result. The other actions are ignored.

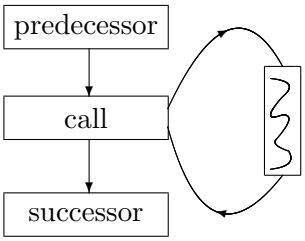


An *iteration* repeats an action as long as a test is passed. Sometimes we do the action first, and then test whether to repeat. The alternative is to do the test first, and if it passes do the action.





A *call* tells us to stop what we're doing, go and perform some other subprogram, and then come back and carry on where we left off.



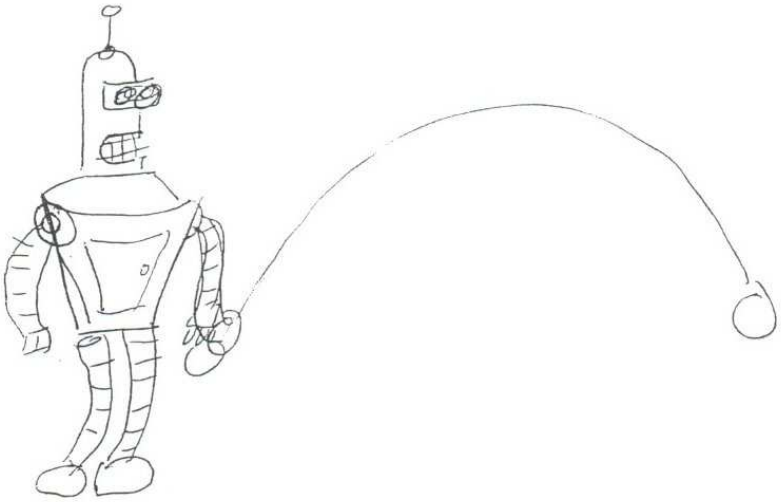
1.2 Writing down programs

The recipe is written pretty informally. It took you a long time to learn English well enough to have a reasonable chance of following those instruction properly. Normal spoken English isn't a suitable language for describing computer programs because we tend to be pretty imprecise when we're talking to humans. Computers usually cannot cope with that level of imprecision: we are going to need a rather stylised version of English that avoids ambiguity, vagueness and imprecision.

Let's look at a simple program that we can run through on a single sheet of paper.

1.2.1 Where will the ball land?

Mathematics gives us lots of useful formulae for calculating things. If you want to design a robot to throw a ball so that it lands at a particular position, you need to know that balls follow parabolic paths through the air.

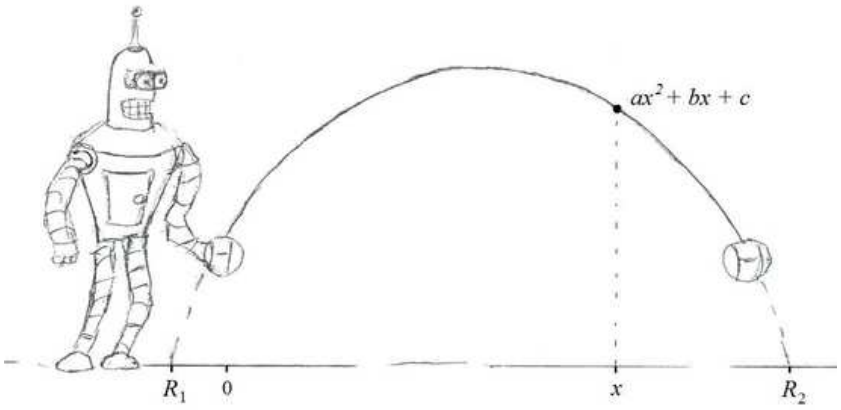


On the website you will find a video showing Sony's QRIO prototype robot throwing a ball: just in case you think this an artificial example!

Parabolic paths are described by quadratic formulae which can be written in the form

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

When applying this equation to the motion of a ball we need to know the velocity of the ball when it is released (that is, both it's speed and direction) and we also need to know the height at which it is released. In a later chapter, we'll show you how to derive the coefficients a, b and c. For now, all you need to know is that the point on the ground below the release point is taken as the origin and for a given distance x from the origin the equation tells us the height y. The point at which our ball hits the ground corresponds to the value(s) of x for which the height is zero, and as you probably know these are called the *roots* of the equation.



For quadratics, there is a formula which tells us the roots if we know a, b and c. It is

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

Visual programs

We will now write a program to compute the roots. The idea here is to break the computation down into a series of single steps which correspond to the operations of a four-function calculator. (Actually, we cheat a bit: we need a four function *plus square root* calculator.) We lay the steps out on paper so as to reflect the order in which they must be performed. Sometimes this kind of program is called a *visual* program.

We start with three boxes which hold the input data (values for a, b and c.) These are like the ingredients that we listed at the start of the Victoria sponge recipe. We then have one box for each basic operation. To the left of the box is the operation for that step, written as a mathematical formula containing exactly one operation. To the right, is a label for the result, which can be used in later operations.

What should we do with operations whose label never appears further down?

Data

a	-4.0	b	5.0	c	1.5

Operations

$-b$		T_1
b^2		T_2
ac		T_3
$4T_3$		T_4
$T_2 - T_4$		T_5
$\sqrt{T_5}$		T_6
$2a$		T_7
$T_1 + T_6$		T_8
$T_1 - T_6$		T_9
T_8/T_7		R_1
T_9/T_7		R_2

There are eleven basic operations here. By slavishly following the worksheet, we can compute the roots of any quadratic without even knowing the whole formula: each step is self contained.

More than sixty years ago this was exactly how computations were done, and the clerks that performed the basic operations were called computers.

In the 1940's machines that could sequence their way through a set of basic computations without human intervention began to appear. These were *automatic* computers, and that's why the names of many early computers ended in *-ac*: for instance EDSAC, ENIAC and even MANIAC.

1.2.2 Textual descriptions

Although these worksheets are helpful for humans, it would be very tedious to always have to draw all those boxes. We need a way to tell a computer what to do using text, and in fact using only the characters that we find on a normal computer keyboard. You won't find a key marked $\boxed{\sqrt{\ }}$ for instance on most keyboards.

Here's an equivalent description that we can type. We give the little program a name, and after the name we list the input data. We then give a list of all the boxes that we shall need for intermediate results, and then we write out the operations.

It turns out that in real computers, different kinds of data need different amounts of space in memory: integers can be represented in less space than real numbers with decimal places, for instance. As a result, when we list data items, we have to specify what kind of data they are, so that the right amount of space can be reserved. In the next chapter we'll tell you about all the different kinds of data. For now, you need to know that most computers provide integers, real numbers called *floating point* numbers and extended floating point numbers call *double precision* numbers. The usual text names for these kinds of data are `int`, `float` and `double`.

The sign $\boxed{:=}$ should be read as *is assigned* or more simply just *gets*.

```
quadratic
{
  double a = -4.0, b = 5.0, c = 1.5;
  double T1, T2, T3, T4, T5, T6, T7, T8, T9, R1, R2;

  T1 := -b;
  T2 := b * b;
  T3 := a * c;
  T4 := 4 * T3;
  T5 := T2 - T4;
  T6 := SQRT(T5);
  T7 := 2 * a;
  T8 := T1 + T6;
  T9 := T1 - T6;
  R1 := T8 / T7;
  R2 := T9 / T7;

  print("First root ", R1);
  print("Second root ", R2);
}
```

1.2.3 Compilers

It would be a lot easier if we could write chunks of formulae out without having to break them down into individual steps. Now, it turns out that although the computer needs everything to be written in terms of these very simple steps, we are allowed to write things out as we would mathematically because a special computer program called a *compiler* can take our formulae and expand them into the list of individual basic operations the machine really needs.

Let's assume we have a compiler. Here's a simpler version of our quadratic program.

```
quadratic
{
  double a = -4.0, b = 5.0, c = 1.5;
  double T1, R1, R2;

  T1 := SQRT(b * b - 4 * a *c);
  R1 := (- b + T1) / (2 * a);
  R2 := (- b - T1) / (2 * a);

  print("First root ", R1);
  print("Second root ", R2);
}
```

In fact real compilers allow even more compression:

```
quadratic
{
  double a = -4.0, b = 5.0, c = 1.5;
  double T1 := SQRT(b * b - 4 * a *c);

  print("First root ", (- b + T1) / (2 * a)));
  print("Second root ", (- b - T1) / (2 * a));
}
```

This third version is more compact, and (to our eyes) easier to understand. We'd say it is an *elegant* solution. We're teaching you how to write programs, but really we want you to learn to write *elegant* programs.

We've slipped in something terribly important here: if a compiler is needed to translate our human-friendly program into one that the computer can really work with; and if a compiler is itself a computer program, how on earth does the compiler program get written?

The simple answer is that the very first compiler is written the hard way, and then we use that compiler to write better compilers: the full story of compiler development will have to wait until you get to the third year.

1.3 Real programming languages

Time for a short history lesson. Early computers were used mostly for numerical calculations, often with military applications. By the early 1950's, some far sighted business people began to apply computers to the massive clerical effort that went into calculating the payroll for large companies, and other accounting operations.

Writing programs for early computers was hard because, as we've seen, every program had to be broken down into single operations, and the writer had to keep track of where all the intermediate results would be stored. By the mid-1950's systems called *autocodes* were beginning to appear. These allowed complex formulae like those of our quadratic program to be written in one line, but control flow, sub program calling and other things still had to be written using the particular computer's special primitive instructions. As a result, every program was tied to the particular kind of computer it was designed for, since different kinds of computer have different primitive instructions.

In 1957 the first compiler that could be used on more than kind of computer appeared. The programmer wrote programs in a notation, or programming language, called FORTRAN which is short for FORmula TRANslation. Not long after, other languages started to appear, and in the last fifty years thousands of different programming languages have been design and implemented. FORTRAN was a breakthrough because it allowed *portability*. A program written in FORTRAN could, in principle at least, be run on any computer with a FORTRAN compiler. It turns out that the portability offered by FORTRAN wasn't really good enough: subtle differences in the way numbers were represented and exceptional events were handled on different computers meant that you could never be sure that large programs would run correctly on a different machine. The portability problem has continued to vex systems designers right down to the present day. The ultimate solution is to use a sort of intermediate 'idealised' computer which has highly standardised behaviour. The first successful system to work this way was *Java* the programming language, whose intermediary is called the *Java Virtual Machine* (JVM). Java can be run on desk top computers, on laptops and even on mobile phones: in fact most Android mobile phone applications are written in Java.

Rather few languages become widely used: perhaps twenty or thirty languages in the last thirty years have had genuinely broad application. At the moment, the most prevalent languages are Java, C++, and C, along with some scripting languages such as Python and Visual basic. Historically, the choice of language to use has been dictated by execution speed, but modern computers are 'fast enough' for many applications so these days we are often more interested in how well a language supports the *writing* of software rather than how rapidly it allows the software to run. So, for instance, the guaranteed portability of Java removes the need to worry aout the differences between, say, a Windows laptop and an Apple desktop. As a result, we use Java even though Java applications usually run a little slower than equivalent programs written in C.

The main thing is to learn to program, not to learn just a language. Com-

puter Science is replete with what we call ROK – Rapidly Obsolete Knowledge. Once you can program properly, you'll find that you can transfer your skills from one language to another. Try not to get too hung up on the idiosyncrasies of each system. Also try not to get into heated debates about the merits or demerits of one system or another. When you've written programs in ten languages, device drivers for three operating systems and graphics programs for at least two windowing systems you'll be well placed to perform an objective critical analysis.

1.4 Starting Java: Hello world!

There's a fine tradition in Computer Science. Every time you start learning a new programming language, the first thing you do is make a program to display `Hello world!` on the screen.

Here's a Java version.

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

In Java, all code is written inside *classes*, which have names. We use classes to organise our code and data. It turns out that classes, when used properly, allow us to re-use bits of code from one application in another.

If you haven't written a program before, most of this one will seem pretty incomprehensible. All will slowly become clear, but for now just note the following.

1. Java programs are written as a sequence of classes which are written in the form `class` *`name`* { }

2. Classes contain a sequence of *members*. The `HelloWorld` class has only one member called `main`. Members can be data or they can be *methods*: small bits of program in the form of instructions to be performed.

3. Each program must have at least one method called `main` otherwise the computer does not know where to start.

1.4.1 Why learn Java?

Java is just one of the many programming languages that we could have chosen to use. It's a good choice for a first programming language for two reasons.

1. Java programs are portable to a far higher degree than earlier languages.

2. Java comes with a lot of built in support for window based programs: that is programs with a *Graphical User Interface* (GUI).

The portability is so good that we can write programs that will run both on your desktop machines and on your mobile phone. Java achieves this by putting a layer of software called the *Java Virtual Machine* (JVM) between your compiler program and the real computer hardware. A Java compiler does not attempt to expand your program into a list of operations for any real piece of computer hardware: instead it expands your program into a list of JVM operations.

1.4.2 Running Java programs

You should think of the JVM as being the specification for a pretend, paper computer. It's quite simple, and for any real computer we can write a small program which will directly interpret the JVM operations. Most web browsers have a JVM interpreter built in, and that's how we can write programs that will run almost anywhere.

You need to know all this so that you can understand the four steps involved in running a Java program.

1. Use a text editor to type the program into a file called something like `Myprog.java` where `Myprog` is the name of the class that contains the `main` method.
2. Compile it form the command line by typing `javac Myprog.java`
3. Run the compiled JVM code by typing `java Myprog`
4. Sit back and admire your handiwork.

On my laptop, this is how I perform these steps.

```
> emacs HelloWorld.java

> javac HelloWorld.java

> java HelloWorld
Hello world!
```

1.4.3 Command line windows and Graphical User Interfaces

On this degree course, you'll spend quite a lot of time controlling the computer from a *command line window*. Until about thirty years ago, computers were mainly controlled from text mode terminals, and the command line window mimics that device. Back in 1972, the first design for a graphical computer that used a mouse and windows appeared. A machine called the Alto was constructed by Xerox in California. It was never fully commercialised, but it directly influenced the design of the Apple Lisa (and subsequently the Mac)

and really should be seen as the prime ancestor of all modern computers. We'd like to encourage you to look up the Alto on the Web and trace it's historical importance. We used to call these kinds of graphical interfaces Windows-Icon-Mouse-Pointer systems, not least because us old-timers were amused by the acronym – WIMP. Nowadays it is more polite to call them GUI's.

Now, before Java came along GUI's themselves were a main source of non-portability. It turns out that writing programs that take full advantage of a GUI's capabilities is quite hard, and different windowing systems such as the original Apple system, Microsoft Windows and the UnixX-windows system need programs to be written differently.

Java includes a standard set of methods for handling GUI's, and the JVM interpreter for each real machine translates these accordingly. As a result, we get portable GUI programs too. Here's our Hello World! program modified so that it puts the message into a window.

```
import javafx.application.*;
import javafx.scene.*;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.stage.*;

public class HelloWorld extends Application {

  public void start(Stage primaryStage) {
    StackPane root = new StackPane();
    root.getChildren().add(new Button("Hello World"));
    primaryStage.setScene(new Scene(root, 300, 250));
    primaryStage.show();
  }
}
```

We compile and run this the same way as before, but this time the message pops up in a window of its own.

Now, there's something new in this program. The first five lines specify **import** packages. Basic Java only understands command line output and input. If we want to use the GUI, we need to tell the Java compiler to look up a library of methods that we're going to need. It is as if your recipe book required you to go and look in another book to find some of the cooking methods. The imports here declare that we are going to use the various parts of the JavaFX GUI: the name of the library is something like `javafx.application` and the final `.*` means that we want to import all of the methods in the JavaFX library. There are alternative GUI's, and that's why we have to specify which one we want, rather than just using some standard methods.

1.4.4 Customising Hello World!

Our Hello World! program would be friendlier if it addressed us by name. The next program uses a special class Scanner to read in characters from the keyboard. For the first time, you are seeing here *object oriented design*. Java can create entities whilst a program is running that represent things in the real world that we want to model or manipulate, and we call these entities objects. If we import `java.util.Scanner` we are allowed to make an object that can be connected to, amongst other things, a real physical keyboard. In this program we make an object called `keyboard`, which represents the user's keyboard, using the incantation

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

Once we've done that, we're allowed to read lines of text using the method `keyboard.nextLine()`.

```
import java.util.Scanner;

public class HelloName
{
  public static void main(String[] args)
  {
    Scanner keyboard = new Scanner(System.in);
    String temp;

    System.out.print("What is your name? ");
    temp = keyboard.nextLine();
    System.out.print("Hello " + temp);
  }
}
```

When we compile and run `HelloName` we get the following screen dialogue.

```
> java HelloName
What is your name? Adrian
Hello Adrian
```

1.4.5 Quadratic roots in Java

Now we know enough to turn our textual description of the quadratic roots formula on page 9 into real working Java.

```
import java.math.*;

public class Quadratic
{
  public static void main(String[] args)
  {
    double a = -4.0, b = 5.0, c = 1.5;
    double t1, r1, r2;

    t1 = Math.sqrt(b * b - 4 * a *c);
    r1 = (- b + t1) / (2 * a);
    r2 = (- b - t1) / (2 * a);

    System.out.println("First root " + r1);
    System.out.println("Second root " + r2);
  }
}
```

When we compile and run this program we get the following output:

```
> java Quadratic
First root -4.0
Second root 24.0
```

1.4.6 Punctuation and capitalisation

One of the things that is hardest to get used to about programming languages is the need for everything to be perfectly punctuated and named. If you get just one character wrong in the name of a method or variable the compiler will complain. You also need to distinguish between commas, which should be read as *and-also* and semi-colons which mean *end of statement*.

Now although it can be very frustrating, the language is carefully designed to catch most of the common errors. For instance, if I mix up commas and semicolons in declarations, the compiler will complain. In this fragment from the `Quadratic` program I have replaced the first comma in the declaration with

a semicolon.

```
public class QuadBroken
{
  public static void main(String[] args)
  {
    double a = -4.0; b = 5.0, c = 1.5;
  }
}
```

Rather impressively, the compiler issues a very helpful error message.

```
> javac QuadBroken.java
QuadBroken.java:5: ';' expected
    double a = -4.0; b = 5.0, c = 1.5;
                              ^
1 error
```

You might wonder why the compiler doesn't just change the character, since it seems to know what is necessary. Look closely at the error message: actually the compiler is suggesting the comma be changed to a semicolon, but really we need the previous semicolon turned into a comma.

Whilst it is quite easy to find typing errors in Java programs it is not always easy to fix them, so the compiler behaves conservatively by pointing out the error, making suggestions where it can, and leaves you to fix the problem.

1.5 Reading in numbers

We've already seen how to read in information from the keyboard: the `HelloName` program in Section 1.4.4 does that. Now we're going to read in two numbers,

and print the result of adding them together. Here's a first attempt.

```
import java.util.Scanner;

public class AddIntegersBroken
{
  public static void main(String[] args)
  {
    Scanner keyboard = new Scanner(System.in);
    String first, second;
    int sum;

    System.out.print("First number? ");
    first = keyboard.nextLine();
    System.out.print("Second number? ");
    second = keyboard.nextLine();
    sum = first + second;
    System.out.print("Sum is " + sum);
  }
}
```

When we try to compile this, though, we get the following disappointing error message:

```
> javac AddIntegersBroken.java
AddIntegersBroken.java:15: incompatible types
found   : java.lang.String
required: int
    sum = first + String;
                ^
1 error
```

The problem here is one of *type incompatibility*. In manual calculation, we often blur the distinction between a number and its representation on paper. Think about Roman numerals for instance: XIV represents the same mathematical entity that we write as 14. Now computers are a bit like this too. They have an internal way of storing actual numbers that is different to the sequence of characters that we use to represent the number on paper. These internal representations are more space efficient, and allow faster arithmetic computations.

In detail, the method `nextLine()` simply reads a string of characters off of the keyboard, but simply trying to 'add' together the string of characters that we read from the keyboard won't do. The `Scanner` class has another method, though, which can convert raw characters into the internal representation of an integer: it's called `nextInt()`. If we read the input that way we can make two

'proper' integers that can be added together directly.

```
import java.util.Scanner;

public class AddIntegers
{
  public static void main(String[] args)
  {
    Scanner keyboard = new Scanner(System.in);
    int first, second;
    int sum;

    System.out.print("First number? ");
    first = keyboard.nextInt();
    System.out.print("Second number? ");
    second = keyboard.nextInt();
    sum = first + second;
    System.out.print("Sum is " + sum);
  }
}
```

This version successfully compiles, and when we run it we get:

```
> java AddIntegers
First number? 12
Second number? 21
Sum is 33
```

1.5.1 Exceptions on numeric input

It's an unfortunate fact that humans often type nonsense into computer programs. In Chapter 1.1.4 we shall see how to catch these kinds of errors. For now, let's see what happens if we type a non-number into our program.

```
> java AddIntegers
First number? adrian
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at AddIntegers.main(AddIntegers.java:12)
```

When something goes wrong with a running Java program, the exceptional event is reported and then we get a list of all the methods that were being executed at the time the exception occurred. The line

```
at AddIntegers.main(AddIntegers.java:12)
```

tells us that the program was in the `main` method when it stopped, at line 12 of the program text file `AddIntegers.java`.

The lines before that refer to the internal workings of the `Scanner` class which we don't necessarily know the details of. The golden rule, then is that when you get an `Exception in ...` message, look at the last line because that should refer to a line of your own code.

1.6 Algorithms

The programs we have written so far have been rather straightforward. Once you have mastered the basic Java notation, translating any mathematical formula into a Java program is simply a clerical exercise. It's time now to think about harder problems. Let's start with the detection of prime numbers. A prime number is a natural number with exactly two natural number divisors (itself and 1). This definition reminds us that 1 is not a prime number because it only has one natural number divisor – itself.

There's no formula that directly generates all the prime numbers. Instead we need some procedure that will enable us to look for them. In computer science, we use the name *algorithm* for a well defined procedure that will generate a result we need in finite time. Algorithms are absolutely at the heart of our subject: they are the building blocks from which we construct working, useful computer systems. As a preview of the rest of this course, we're now going to look at one of the oldest known algorithms, and show you how to execute it on a computer.

1.7 The sieve of Eratosthenes on paper

Eratosthenes was a Greek mathematician who lived from 276BC to 194BC. He was the first person to calculate the circumference of the earth: to an accuracy of 1%!

He also described a method to find all primes between 2 and some upper bound. The idea is to write down all the numbers between 2 and the upper bound. Then we start at 2, which we call the *base* number, and cross out the multiples of *base*, that is *4, 6, 8, 10, ...* up to the upper bound.

Having completed this first pass, we have detected a new prime: the first uncrossed number greater than base is prime (in this case it will be 3).

Now we set *base* to be the new prime, and delete all its multiples. Again, the first uncrossed number greater than *base* is prime. And so we go on.

Although we've talked about crossing out the multiples, in detail we just step from one to the next: in fact the Sieve allows us to find prime numbers using only addition!

You'll find a very nice animation of Eratosthenes' algorithm in the Wikipedia page at

```
http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes
```

The best way to understand the algorithm is to try it out by hand. Here's a grid showing the natural numbers up to 55 for you to use.

0	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31	32	33	34
35	36	37	38	39	40	41
42	43	44	45	46	47	48
49	50	51	52	53	54	55

1.8 The sieve of Eratosthenes in Java

Now we are going to jump far ahead. Ideally, we would introduce computer programming to you one little step at a time. Mostly, that's what we're going to do but right now we need to show you a complete real program that has complicated control flow. If you look back at the other Java programs in this chapter you will see that they are all just sequences. In Section 1.1.4 we described four kinds of control flow: sequences, selections, iterations and calls. This program has them all. In addition, we're going to use a special kind of *collection* of data called an **array** to represent the grid of numbers.

We don't expect you to fully understand all the subtleties at this stage: we just want to give you a flavour of things to come. We hope that when you have finished the course, you'll be able to look again at this program and see that actually it can be improved, in the sense that there is a similar program that generates the same output but using fewer computer operations: that is, there is a *faster* version.

1.8.1 Arrays

In Java, an array is a one-dimensional collection of data items, all of the same type or class. Each element of the array is given a sequential number called its *index* starting at zero. Arrays are of fixed size: you specify the number of elements you want when the array is created and after that the array cannot expand or contract.

Declaring arrays

Array declarations create an array *variable* but do not reserve the actual memory in which the array elements live. This is useful, because it means we can delay the creation of the array elements until the program runs, which allows

us to tailor the size of the array. Array index expressions are written in square brackets, and the array variable syntax uses an empty pair of brackets to indicate that we want an array variable, not a simple variable:

```
int [] intArray;
```

Creating arrays

Once we have declared an array variable, we can create an actual array for it to reference. There are two ways to make an array: with the `new` operator as is usual for objects and with the `{ ... }` curly brackets initialiser.

```
// Declare two array variables which don't yet have any actual
// array elements
int [] intArrayA, intArrayB;

// Use new to create an int array with indices 0..19
intArrayA = new int[20];

// Create an int array with indices 0..7 using an intialiser
// values set to the first eight prime numbers
intArrayB = {2, 3, 5, 7, 11, 13, 17, 19}
```

After initialisation, `intArrayB` looks like this: eight cells indexed 0...7 containing the first eight prime numbers.

0	1	2	3	4	5	6	7
2	3	5	7	11	13	17	19

It is only at the point of creation that the size of the array must be specified, either explicitly with `new` or implicitly with the initialiser. Java then allocates enough memory for the array: since the adjacent pieces of memory will in general be used for other things, the size of an array is fixed for the duration of the program. If you need to extend an array, then you must create a new array object of the required size and copy values from the old version into it, as we shall show you in Section 5.3.8. As long as you do not continue to use the old array, it will be cleaned up and its memory reclaimed by the *garbage collector*.

The length of an array is held in a field called `length`, so `intArrayA.length` would contain 20 and `intArrayB.length` 8.

1.8.2 Selection control flow

There are several kinds of selection in Java. The simplest is the `if` statement which has two parts: a *condition* which computes a true/false value and a *body* which is only executed when the condition part is TRUE.

When the computer encounters the `if` statement it evaluates the condition and then only executes the body if the condition is true.

```
int x = 7;

if (x == 3)
  System.out.println("x is three");

if (x != 3)
  System.out.println("x is not three");
```

In this case the program will output

```
x is not three
```

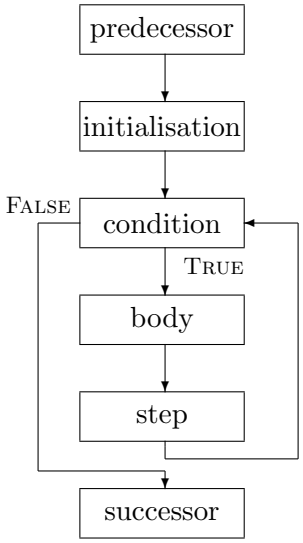
We'll tell you more about control flow, including the different kinds of selection statements, in Chapter 3.

1.8.3 Iteration control flow

There are several ways to do iteration in Java. Probably the most complicated, but also the most useful, is the `for` statement which has four parts: an *initialisation*, a *condition*, a *step* and a *body*. We write these as

```
for ( initialisation ; condition ; step ) body
```

Here's how Java executes the `for` statement:



The *initialisation* is performed first. Then we perform the *condition*. If it fails (returns FALSE) we go on to the successor statement. Otherwise, we perform the *body* followed by the *step* and then come back round to the *condition* again.

We'll tell you more about control flow, including the different kinds of iteration statements, in Chapter 3.

1.8.4 Iterating over arrays with `for` statements

Often, we want to process the elements of an array in some way: print them all out, or add them up, for instance. We can do this by using a `for` loop to step over all the elements. For instance, we can find the sum of the elements in `intArrayB` like this:

```
int sum = 0;

for (int i = 0; i < intArrayB.length; i++)
  sum = sum + intArrayB[i];
```

Occasionally, we want to scan an array in reverse order. This needs a little care because we need to remember that the highest available index is one less than the length. Here's how to print the first eight primes in reverse order.

```
for (int i = intArrayB.length - 1; i >= 0; i--)
  System.out.println(intArrayB[i]);
```

1.8.5 The Java code

Now, let's look at the full program.

```
public class Sieve
{
  public static void main(String[] args)
  {
    int upperbound = 56, base, multiple;
    int buffer[] = new int[upperbound];

    for (base = 2; base < upperbound; base++)
      for (multiple = 2 * base;
           multiple < upperbound;
           multiple = multiple + base)
        buffer[multiple] = 1;

    // Output results
    for (base = 2; base < upperbound; base++)
      if (buffer[base] == 0)
        System.out.println(base);
  }
}
```

There's quite a lot to take in here, and you are not expected to get it all first time.

The first thing to note is that we have defined a vale `upperbound` which sets the size of the grid. We can change the program to, say, generate primes up

to 300 just by changing this value. The grid itself is represented by the array called `buffer`. It starts off with each element set to zero. As we strike out an element, we change its value to 1.

The next new thing is the `++` operator. When I write `base++`, this is shorthand for `base = base + 1`, so `base++` means: increment base by one.

The output loop is easiest to understand. We step through the whole array from element 2 up to the upper bound, incrementing by one each time: `for (base = 2; base < upperbound; base++)`. We examine each element to see if it contains a zero: `if (buffer[base] == 0)`. If it does, then it was never struck out so it must be a prime; so we print it out on the screen.

Here's the output from program `sieve`.

```
> java Sieve
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
```

The hardest part of the program is the section that does the striking out. This contains two *nested* `for loops`, by which we mean that we step through the elements of the array with one loop, and at every element we stop and run another loop.

The outer loop uses the *induction variable* `base` to keep track of where we have got to, and the inner loop uses `multiple`. `base` increments from 2 to the upperbound in steps of 1. At each point, we stop and step `multiple` from the current base value to the upperbound in steps of `base`, that is we visit each multiple of the `base` value. We strike out each element visited, by setting its array value to 1.

Now, try *animating* the program by hand, stepping through it as though you were the computer. You know what the program is supposed to do from using the paper grid: see if you can match the program to your manual version.

1.9 Next steps

This chapter has introduced to the elements of Java programming. The rest of the course comes in five section, each with its own chapter:

- ◇ *data*, in which we look at the different kinds of internal representation used by Java and the many associated built in operators;
- ◇ *control*, on the different control flow structures;
- ◇ *packaging*, which describes the facilities for making code reusable;
- ◇ *structures*, which describes arrays in more detail along with *stretchable* data structures like linked lists; and
- ◇ *interaction*, which focuses on input, output and graphical user interfaces.

You'll build your understanding through programming laboratories and assignments. Remember that learning to program is a bit like learning to drive: you can read as many books as you like but there is no substitute for practice, practice, practice.

2 Data

Data are our ingredients

We have already seen that there is an important difference between a number, and the string of characters that we use to display a number on the screen or for keyboard input. In Section 1.5 we found that we could do arithmetic with numbers, but we could not do arithmetic directly with strings of characters: we needed to convert the string to a number first.

Ever since the earliest days of computing, machines have had separate support for characters, integers and floating point numbers. Most machines allow several different kinds of integers and several kinds of floating point numbers, allowing the programmer to trade memory space for precision.

2.1 Numbers inside the computer

On paper, we represent numbers using a positional notation based on powers of ten in which numbers are written as a string of the digits 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. A number like 489 is really shorthand for the number

$$4 \times 100 + 8 \times 10 + 9 \times 1$$

or

$$4 \times 10^2 + 8 \times 10^1 + 9 \times 10^0$$

We use positional notation because it allows some straightforward algorithms for addition, subtraction, long multiplication and long division. Most of us learn these algorithms in primary school, and they become so instinctive that we sometimes forget that positional notation has not been used by all cultures: try doing long multiplication directly in roman numerals, for instance. If you are interested in how ancient cultures did perform arithmetic, we recommend Lucas Bunt *et al*'s survey of Egyptian, Babylonian and Greek mathematics [**?**].

Inside the computer, we do the same thing except that we are only allowed to use the digits 0 and 1, and as a result each position in the binary number corresponds to a power of two, instead of a power of ten. So the binary number 1 1110 1001 is shorthand for

$$1 \times 2^8 + 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

or

$$256 + 128 + 64 + 32 + 8 + 1 = 489$$

Notice the way that the binary digits 1 1110 1001 are grouped into fours from the right, with left a space between groups. This makes it easier to read long binary numbers. Computer languages don't usually allow this convention, so we would have to write the number as `111101001` instead.

Each digit of a binary number is called a *bit* which is short for *binary digit*. An n-digit decimal number integer lies between zero and $(10^n - 1)$; in the same way an n-bit (binary digit) number can represent any value in the range $0 \ldots (2^n$ -1). So, an eight-bit binary number can represent positive decimal integers in the range $0 \ldots 127$.

A computer's memory is really just a very long string of bits which we can set to one or zero. It is not immediately obvious that we can generate all the complex behaviours of modern computers simply by flicking the state of bits back and forth but really that is all our programs do.

2.1.1 Why do computers use binary?

This section is background material that you do not need to learn for the exam.

Nearly all the computers that have ever been made use a binary (two state) representation internally. It is natural to use binary numbers to store data in computers because many of the physical mechanisms that we use to store information are inherently two-state. For instance, data is stored on a CD or DVD as a string of bits represented by a spiral of *pits*: small indentations in the foil layer in the middle of a CD's plastic-foil-plastic sandwich. The bits are read by a laser beam which bounces off the shiny foil into a photo-detector. When a pit passes under the beam, it is scattered instead of being cleanly reflected: the photo detector sees a break in the beam which registers as 1-bit.

It also turns out that digital electronics is more reliable if we use a two-state system because electrical noise is less likely to cause a state change if the available power supply voltage is divided into only two zones rather than three or even ten (both of which have been tried).

2.1.2 Names for big numbers

We need names for large numbers because just reading out a string of digits is confusing. In decimal we have the words hundred, thousand, million, billion and trillion. American usage, which is now mostly also used in the UK is as follows. (In the UK, a billion used to mean 10^{12}.)

name	multiplier	decimal power	prefix
hundred	100	10^2	
thousand	1 000	10^3	k (kilo-)
million	1 000 000	10^6	M (mega-)
billion	1 000 000 000	10^9	G (giga-)
trillion	1 000 000 000 000	10^{12}	T (tera-)
quadrillion	1 000 000 000 000 000	10^{15}	P (peta-)

Once we have these names, we can say things like: the US government will buy up to 0.7 trillion dollars worth of bad loans, although they seem to prefer to say 700 billion dollars, perhaps because it might sound smaller.

We need names for big binary numbers too. This is what we use.

multiplier	binary power	prefix
1 024	2^{10}	Ki (KilabInary-)
1 048 576	2^{20}	Mi (MegabInary-)
1 073 741 824	2^{30}	Gi (GigabInary-)
1 099 511 627 776	2^{40}	Ti (TerabInary-)
1 125 899 906 842 624	2^{50}	Pi (PetabInary-)

Note that the binary prefixes include a trailing i to indicate the use of binary multipliers. Although recommended, the use of the trailing i has not really been taken up by manufacturers, who sometimes exploited this potential confusion by, for instance, by (correctly) labelling a disk drive as 160GByte which is only 149GiByte.

2.1.3 Hexadecimal notation

Long binary numbers are so hard to read that we rarely write them out. Instead, we extend the group-by-four convention to use *hexadecimal notation*. The idea is to take each group of four bits, and replace it with a single digit as follows:

Binary	Decimal	Hex
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Using this notation, 489 in decimal (base 10) represents the same number as 111101001 in binary (base 2) and 1E9 in hexadecimal (base 16).

When writing a hexadecimal number in a computer program, we use the prefix `0x`: so hexadecimal `0x100` represents the same number as the decimal `256`.

2.2 Positive and negative integer numbers

So far we have only really talked about strings of bits, and their equivalent positive decimal integer values. How should we represent negative numbers? Well, one way would be to use the leftmost bit as a *sign* bit, with say zero indicating a positive number and one a negative number. We then use the bits in the rest of the string to hold a positive value. This *sign-magnitude* representation makes it reasonably easy to read negative numbers, but there is one resulting unpleasantness: we have both positive and negative values for zero. Imagine using four bits in the string to hold values in the range -7 ... 7.

0111	+7
0110	+6
0101	+5
0100	+4
0011	+3
0010	+2
0001	+1
0000	+0
1111	-7
1110	-6
1101	-5
1100	-4
1011	-3
1010	-2
1001	-1
1000	-0

Apart from being confusing (why have two values for zero?) this approach is wasteful: our n-bit representation has only $2^n - 1$ unique values.

Almost universally, modern computers reject sign-magnitude in favour of *two's complement* notation. The idea is that we match the sequence of binary numbers with the values along the integer number line. The number zero is represented by a string of zeros. As we increment through the binary sequence from zero we go through positive numbers in the sequence 0, 1, 2, 3.... As we decrement from zero we roll-under into the negatives -1, -2, -3....

0111	+7
0110	+6
0101	+5
0100	+4
0011	+3
0010	+2
0001	+1
0000	+0
1111	-1
1110	-2
1101	-3
1100	-4
1011	-5
1010	-6
1001	-7
1000	-8

We can extend this idea to any length of bit string. For an n-bit string, a two's complement interpretation can represent values in the range

$$(-2^{(n-1)}) \ldots (+2^{(n-1)} - 1).$$

Doing arithmetic with this arrangement is easy: to add, say, three to two, we find the entry in the sequence for two and move forward three places (0010 $\rightarrow$ 0011 $\rightarrow$ 0100 $\rightarrow$ 0101) to arrive at five. More interestingly, to subtract three from two we again start at the entry for two and then move *backwards* by three places (0010 $\rightarrow$ 0001 $\rightarrow$ 0000 $\rightarrow$ 1111) to arrive at -1.

This very simple rule allows easy addition and subtraction. There is a problem though: the sequence is circular in the sense that if we move forward one from 0111 we go to 1000; if we move backward from 1000 we go to 0111.

Think what happens as we try to add two to six. The sequence goes (0110 $\rightarrow$ 0111 $\rightarrow$ 1000) giving a result of -8! This effect is called *overflow* because the arithmetic result has overflowed the available number of bits. Really we needed a *five*-bit representation to hold the result which could encode values from -32 to 31. Of course, we would still then have a problem if we tried to add two to 30.

2.3 Real numbers, and floating point approximations

Representation of real numbers within computers is a hard problem, and some uncomfortable compromises need to be made. In any given base (decimal, binary or hexadecimal for instance) there will be some rational numbers that do not have a finite representation. So, for instance, 1/10 can be exactly represented by 0.1 in decimal, but 1/3 corresponds to 0.33333... in decimal. We read this as 0.3 recurring.

The fundamental problem is that computers are *finite*, and so we can only use finitely long bit strings to represent quantities. In some cases there are only finite approximations to the real numbers we actually want. We call these finite

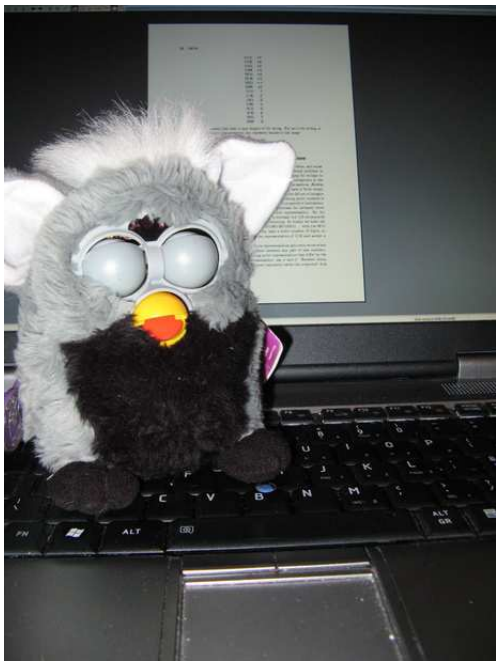


Figure 2.1 The FLurbie of doom

approximations *floating point* numbers. We've already seen that computer integers have a finite range. In that sense the computer's integers cannot represent the full set of integers as understood by mathematicians. Instead we represent only a contiguous selection of integers around zero. With floating point numbers, not only are we limited as to the absolute magnitude of the values that we can use, but in addition every time we use a real number we are likely to have to *round* it to the nearest available value: at least integer values in the computer are exact where they available.

This problem of *rounding error* is a direct consequence of the fact that real numbers do not have successors: as you will find in the maths course, there are *uncountably many* reals between any pair of real numbers. To give you some insight into this problem, imagine taking two elements from our floating point representation that differ by the smallest amount allowed in our representation: say x and x'. Between them, there will be another real y that we cannot represent inside the computer! To make matters worse, there will also be another real between that x and y, and a different one between y and x'. In fact, by extending this argument you can see that there will be infinitely many non-represent able reals between any two floating point numbers! This is very uncomfortable.

Life gets even harder when we actually start doing arithmetic with our floating point numbers. If we add together two approximations, will the result be even more of an approximation to the 'mathematically correct' computation? Well, quite possibly. You cannot assume that rounding errors only affect the last few decimal places since it is quite possible for the error to grow during a long

sequence of calculations and overwhelm the expected result. Such calculations are called *numerically unstable*.

It is wise to treat floating point calculations with some scepticism. Really, it is as though every time we manipulate floating point numbers there is a risk that a small demon inside the computer will get in there and fiddle with the result so that a small error is introduced. Over a long calculation, the small errors can really cause a problem. We call this demon the FLurbie of Doom: Figure 2.1 shows a picture of one we found hiding in a laptop. He may look cute, but he's trouble. Don't use floating point numbers where it is important to maintain exactness: in financial accounting, for instance, it is best to use an integer count of the number of pennies in the account.

As a very simple example, let's rework the program from Section 1.5 to add floating point numbers together.

```
import java.util.Scanner;

class AddDoubles
{
  public static void main(String[] args)
  {
    Scanner keyboard = new Scanner(System.in);
    double first, second;
    double sum;

    System.out.print("First number? ");
    first = keyboard.nextDouble();
    System.out.print("Second number? ");
    second = keyboard.nextDouble();
    sum = first + second;
    System.out.print("Sum is " + sum);
  }
}
```

When we run this program with inputs 0.1 and 0.2 we get this output:

```
> java AddDoubles
First number? 0.1
Second number? 0.2
Sum is 0.30000000000000004
```

The problem here is that 1/10 does not have an exact representation in binary, just as 1/3 does not have an exact decimal representation. The computer does its best, uses the nearest available approximation to 0.1, but the FLurbie of Doom introduces a small error in the seventeenth decimal place just to remind us that *real numbers inside computers are not exact*.

2.4 Characters and strings of characters

After all these difficulties, it is a relief to find that representing characters is quite straightforward. Since there are only a finite number of characters, we do not have problems arising from our finite representations. We simply put the characters into some order, and then allocate each a small integer corresponding to their position.

In the early days, each computer had its own character code, which naturally made interchange of textual data very difficult. By 1963, the American Standard Code for Information Interchange (ASCII, pronounced ass-key) was published, and most computers standardised on it. ASCII is an eight-bit code in which, for instance, the letter A corresponds to code decimal 65 and the letter a to decimal 97. ASCII is quite limited though: it doesn't include some European characters such as Å and £, and it has no support at all for the myriad other script characters that are in use, or have been used, around the world.

In 1991, a new standard called Unicode was published. Unicode characters are 16-bits long, and through the use of a clever paging system allows essentially every character ever used in human history (including, for instance, Egyptian Hieroglyphs) to be encoded. Most new computer languages and systems use Unicode: the ASCII code corresponds to the *Basic Latin* part of Unicode giving backwards compatibility with older programs.

Characters on their own are of limited application: mostly we have to deal with whole words and sentences. A piece of text is really just a lot of individual characters strung together. Since text is so important, Java provides a `string` data type which is not actually one of the primitive types, but which has some special features. In detail, a string is an array of characters: we'll describe them in more detail in section 2.13.4.

2.5 Logical values

In computer programming we make extensive use of *logical* (TRUE/FALSE) values to control the flow of execution through our applications. These logical values are often called *Boolean* values after the British mathematician George Boole (1815–1864) who first brought an algebraic approach to the study of logic; work that underpins the design of all digital computers.

Since Boolean values can only have one of two states, a single bit is sufficient to represent them. In practice, computers often use a single byte (the smallest sequence of bits that can be fetched from memory) or small integer because extracting individual bits from a byte needs several machine operations, and so is slower than simply using the entire byte and ignoring the top seven bits.

2.6 The pigeonhole model of memory

When the computer is running, it continually fetches data from memory and writes results back. It is more efficient for computers to access memory a few

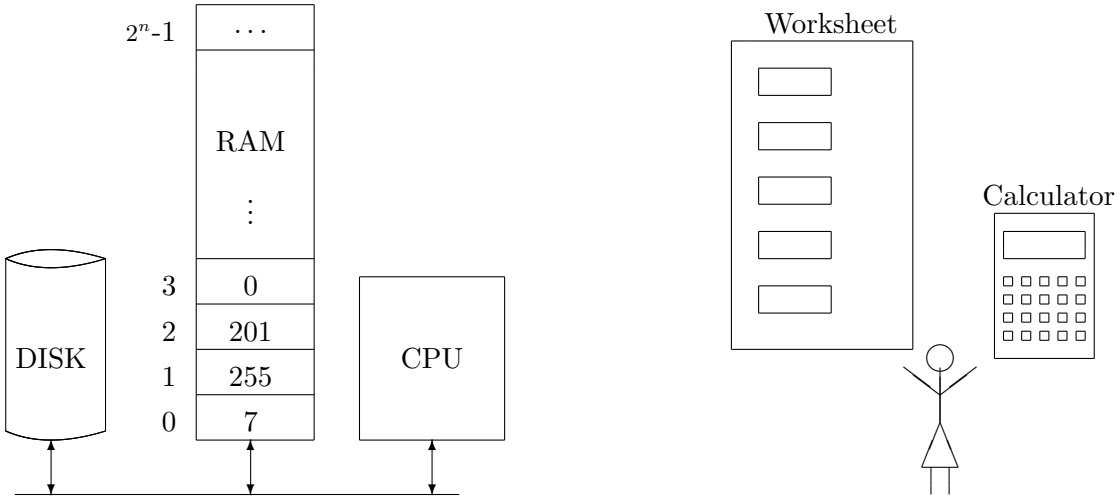


Figure 2.2 Inside the computer

bits at a time. In the early days, some machines fetched as many as 60 bits at a time, but practice soon settled down to fetching enough bits to represent one text character. Most machines built since the mid-1960's have standardised on eight bits per character, so memory is organised as a set of eight-bit *bytes*. A byte (pronounced bite) is the smallest bite of memory that a computer can access in one go: one bite at memory. Although strictly speaking a byte could be almost any number of bits, it is universally understood to mean eight bits.

Now, what does it mean to say that the computer accesses memory? Well, Figure 2.2 shows a simplified picture of how a computer is organised inside. We have drawn both an electronic computer, and a human using a four function calculator and the visual worksheet from the last chapter. The Central Processing Unit (CPU) performs the actual calculations: it is like the calculator in the visual worksheet case. The *Random Access Memory* (RAM) holds all the data – it is the equivalent of our worksheet with its set of named cells. The difference is that the computer's RAM has many more locations, and instead of having names they have numeric *addresses*.

The best way to think of this is that the computer's RAM is like a lot of pigeonholes, each of which can hold one number. We can change the contents of any pigeonhole at any time. The pigeonholes are all arranged in order, and we number them from zero, so irrespective of the contents, each pigeonhole responds to a permanently allocated number called its address.

In practice, the computer accesses memory using operations like *'assign the number 57 to the third pigeonhole'* and *'get the contents of pigeonhole with address number 78'*. When we write in Java an expression like

```
x = 15;
```

this really means *'assign the number 15 to the pigeonhole corresponding to variable* `x`*'*.

On a typical modern computer, and also on the JVM pseudo-computer, the memory can have up to 2^{32} (which in base 10 is 41,947,234,304) separate locations with addresses from 0 . . . 41,947,234,303. Each memory location holds eight bits: a byte. We say that there can be up to 4GiByte of memory, since 4Gi is 2^{32}. Although this is an enormous number, it is still not enough for some applications. The latest generation of computers use 64-bit memory addressing, allowing up to 2^{64} separate memory locations.

2.7 Declarations

As you know, we have to write down all our variables before we make use of them and specify what type they are: we call specifications these *variable declarations*.

2.7.1 Valid identifiers

In Java (and most other programming languages) variable names must start with a letter and can continue with a mix of numbers, letters and the underscore character (_). We call these names *identifiers* and they can be used for other things as well as variable names, as we shall see.

In Java it is technically legal to begin an identifier with an underscore or a dollar sign, but many systems reserve such identifiers for internal use, so we most strongly recommend that you never start an identifier with an underscore, and that you never use dollar signs at all. You must also avoid accidentally using a Java keyword such as `class` of `break`.

Java is case sensitive, which means that `Adrian` and `adrian` signify different identifiers. It is not a good idea to have similarly named identifiers that differ only in their case – that would make your programs hard to read.

When choosing names for identifiers try to choose things that spell out words that have some mnemonic value. The usual convention is to use lower case characters for variable names, except that if your identifier represents two or more words (such as `totalPrice`) then capitalise the second and subsequent words. (An older convention uses lower case throughout but separates words with an underscore like this: `total_price`. We strongly recommend you use only the Java capitalisation convention when writing Java programs.) The convention for class names is that they begin with a capital letter.

2.7.2 Types and memory consumption

Depending on the size of the data type for the variable, the compiler allocates one or more memory pigeonholes to the variable. The name of the variable is matched to the address of the first pigeonhole for that variable. This is how the compiler translates your variable names into the numeric addresses used by the JVM.

So, when we declare a variable, we are really telling the JVM to allocate a box in memory big enough to hold values of the variable's type. Here's what

happens when we declare an `int`.

```
int x;          x
```

The JVM has made a box for the `x` big enough to hold an `int`. Now we *initialise* `x` by assigning three to it. The number `3` is put into the box for `x`.

```
x = 3;          x   3
```

The fact that variables have to be declared and given a type also allows the compiler to carry out consistency checks and pick up some programming errors. You should always read compiler errors carefully to identify missing declarations, initialisations and typing errors in variable names.

2.8 Operators and expressions

We are used to using operators for everyday arithmetic: everybody agrees on what $3+4$ means. You also know that operators can have different *precedences*: $3+4\times 2$ yields 11 *not* 14 because multiplication is always done before addition and subtraction. Really, operator precedence is just a convention that allows us to drop parentheses. We could write the expression out longhand as $3+(4\times 2)$ in which case there would be no ambiguity.

Operators are the building blocks of expressions. We've already seen (in Sections 1.2.2 and 1.2.3) how individual operators may be put together to make compact expressions that can replace many lines of source code. Java provides a lot of operators grouped into 14 precedence levels, which is a lot to remember. In practice, it is quite easy to make mistakes when relying on operator precedences, and we recommend that you use parentheses to explicitly specify ordering except for simple cases.

Table 2.1 summarises the available operators: the meaning of some of these will not become clear until later chapters but we have included everything here so that you have a complete reference in one place.

2.8.1 Selection operators - level 1

In chapters four and five we shall look in detail at Java's way of packing data and code together into *objects*. Objects have fields which can be simple variables; methods representing code we can call; or arrays of other objects or primitive types. The three selection operators `[ ]`, `( )` and `.` are used to select elements of an array, to call a method, and to select a field, respectively. We'll delay further discussion to those later chapters.

2.8.2 Unary operators - level 2

The unary operators, as their name suggests, take only a single operand. Negation is represented by unary minus `-` as you would expect: `3 + -4` yields `-1`.

Syntax	Semantics	Precedence	Associativity
`[ ]`	Array element select	1	Left
`( )`	Method call	1	Left
`.`	Class member select	1	Left
`+`	Unary posite	2	Right
`-`	Unary negate	2	Right
`!`	Unary logical complement	2	Right
`~`	Unary bitwise complement	2	Right
`++`	Unary postfix or prefix complement	2	Right
`--`	Unary postfix or prefix complement	2	Right
`(type)`	Type cast	2	Right
`new`	Create new object	2	Right
`*`	Multiply	3	Left
`/`	Divide	3	Left
`%`	Remainder after division	3	Left
`+`	Add	4	Left
`-`	Subtract	4	Left
`+`	String concatenation	4	Left
`<<`	Signed left shift	5	Left
`>>`	Signed right shift	5	Left
`>>>`	Unsigned right shift	5	Left
`<`	Less than	6	Left
`<=`	Less than or equal	6	Left
`>`	Greater than	6	Left
`>=`	Greater than or equal	6	Left
`instanceof`	Reference is an instance of	6	Left
`==`	Equal to	7	Left
`!=`	Not equal to	7	Left
`&`	Bitwise AND	8	Left
`&`	Logical AND	8	Left
`^`	Bitwise XOR	9	Left
`^`	Logical XOR	9	Left
`|`	Bitwise OR	10	Left
`|`	Logical OR	10	Left
`&&`	Logical short circuit AND	11	Left
`||`	Logical short circuit OR	12	Left
`? :`	Conditional	13	Right
`=`	Assignment	14	Right
`+*=`	Operation and assign	14	Right
`-=`	Operation and assign	14	Right
`*=`	Operation and assign	14	Right
`/=`	Operation and assign	14	Right
`%=`	Operation and assign	14	Right
`<<=`	Operation and assign	14	Right
`>>=`	Operation and assign	14	Right
`>>>=`	Operation and assign	14	Right
`&=`	Operation and assign	14	Right
`^=`	Operation and assign	14	Right
`|=`	Operation and assign	14	Right

Table 2.1 Java operator summary

We also have unary + which actually doesn't do anything, but which we sometimes include in the definition of constants to underline that they are positive values.

Logical NOT is represented by the exclamation mark operator !. New programmers often confuse NOT with BITWISE-COMPLEMENT (~) which inverts every binary digit in its operand. The NOT (!) operator is meant to be used with boolean values only.

We shall discuss the ++ and -- autoincrement and autodecrement operators in Section 2.9 below.

2.8.3 Binary arithmetic operators - levels 3, 4 and 5

The common arithmetic operations such as addition, subtraction, multiplication and division can be applied to both integer and real operands. They take two data values of the same type and produce a third value of that type. You are allowed to write expressions that mix integer and real types, but the results can catch out the unwary: see Section 2.10.1 for more detail.

You'll find some less familiar operators such as % which provides the *remainder* after integer division, and <<, the signed left shift operator which yields the same result as multiplying an integer by two, but is more efficient (in the sense of running faster) on many computers.

2.8.4 Relational operators - levels 6 and 7

The *relational* operators such as < and >= take two data values and produce a Boolean value result. We can use the boolean results in combination with the logical operators to check numbers against limits. For instance, if we want to know whether an exam mark is valid we might write

```
if (mark >= 0 && mark <= 100)
  println("Valid")
else
  println("Invalid");
```

String comparisons

We have to be careful when comparing strings, because strings are implemented as arrays, and when we ask if two array variables are the same we are really asking if they are actually the same array. So for instance if we write

```
String first = "adrian";
String second = "adrian";
```

you might hope that the expression `first == second` would evaluate to true. But it doesn't because `first` and `second` are independent array variables that happen to contain the same characters...

This is a good example of the way that in computer science we always have to be careful to distinguish between a thing and it's name. Really, when we say

`first == second` we are asking if the strings have the same *name*, not whether the strings contain the same characters.

Java does of course provide a way to compare strings: you need to write `first.equals(second)`. Strings have a set of built in methods, and this expression asks the `first` string to use its own `equals()` method with argument `second` to check the contents of the strings. You can also use the form `first.compareTo(second)` which will return a negative integer if first is lexicographically less than second, a positive integer if it is greater and zero if they are the same.

Broadly speaking, 'lexicographically less than' means 'would appear in a dictionary before' — `first` is lexicographically less than second if the first character in which they differ is less (in Unicode terms) for the `first` string, or if they are the same right up to the end of `first` but `second` is longer, that is, if `first` is a proper prefix of `second`.

2.8.5 Logical operators - levels 7 – 10

Java provides logical OR (`|`), logical `exclusive-or` (`^`) and logical AND (`&`) operators to operate on Boolean variables and the same operator forms that perform bitwise operations on integer operands.

Here are the truth tables for those three operations.

a	b	a OR b
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

a	b	a EX-OR b
TRUE	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

a	b	a AND b
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

2.8.6 Short circuit logical operators - levels 11 – 12

In computer science we are always interested in efficiency. Efficient programs are fast and small. Inefficient programs are slow and bloated.

Java provides two special *short circuit* operators that can be used to save computation. Consider an expression like

```
(a > b) | first.equals(second)
```

Now, string comparisons can require a great deal of computation — comparing all the characters in two long strings that share a long prefix can take a long time. If in fact `a` is greater than `b` then we do not actually need to evaluate the string comparison: the truth table for logical OR shows that once one operand is TRUE, then the overall expression must yield TRUE.

In Java, we can rewrite this expression as

```
(a > b) || first.equals(second)
```

The `||` operator is the *short circuit* form of the `|` operator. It evaluates its left operand, and if that yields TRUE then it skips the evaluation of the right operand. Similarly, the `&&` operator short circuits of its left operand evaluates to FALSE because a logical AND must yield FALSE if either of its operands is FALSE.

Side effect operators in short circuits

There is a potential for a subtle bug here. If the right operand expression of a short circuit operator contains assignment operations, or autoincrement/autodecrement operations then they will only be executed if the short circuit is not applied. See the next section for more information.

2.8.7 The conditional operator - level 13

Sometimes it is useful to be able to make a selection as part of an expression. Here is an `if` statement that computes the positive value of a number, and then adds 20 to it:

```
int b;

if (a < 0)
  b = -a;
else
  b = a;

b += 20;
```

Java has a *conditional operator* that is *triadic*, that is, it takes three operands: a predicate, a then-expression and an else-expression. Here's the syntax:

```
predicate-expression ?  then-expression :  else-expression
```

The operator can be embedded in expressions just like any other operator. Here's a version of the previous program fragment that uses a conditional expression to form a more compact program.

```
int b = 20 + ( (a < 0) ? -a : a);
```

We declare `b` and initialise it to the expression `20 + conditional` where the *conditional* should be read as: *Is* `a` *less than zero ? If so then use* `-a` *else use* `a`.

Conditional expressions can be hard to read, so we wouldn't recommend that you use them for routine conditionals, but there are times when they are very convenient — selecting between strings inside a `print()` method call, for instance. Here's our favourite example. It is quite common for programmers to produce ugly messages such as

```
31 error(s) found
```

The point here is that in English we would say '`1 error`' but '`2 errors`': the programmer is lazy and has used the ugly form '`error(s)`' rather than producing a grammatically correct sentence. This is a shame because conditional expressions provide an elegant solution:

```
class Errors
{
  public static void main(String[] args)
  {
    int errorCount = 2;

    System.out.println(errorCount +" error" +
                       (errorCount == 1 ? "" : "s") +
                       " found");
  }
}
```

The conditional embedded within the `println()` call either adds an `s` to the word `error` or adds and empty string depending on whether `errorCount` is equal to one.

2.8.8 Assignment operators - level 14

Assignment is itself an operator in Java but the left hand side must evaluate to something that represents a box in memory into which we can assign something: writing `3 = x + 1` is meaningless because `3` is a constant and we can't assign a value to it. We discuss assignment and other operators with side effects in the next section.

2.9 Operators with side effects

In mathematics, operators and functions simply compute a result. Computer programs, at least ones written in *procedural* languages like Java, are based around the notion of boxes in memory into which we write results. This turns out to be an absolutely fundamental difference, and the notion of a variable in

computer languages is very different to a variable in a mathematical equation which is simply a placeholder for a value.

(As an aside, there exist computer languages that hide the notion of boxes so that variables and values work as they do in mathematics – those languages are called *functional* languages. It is easier (not easy) to reason mathematically about functional languages, but procedural languages are much more popular in practice mainly because functional languages often execute more slowly than their procedural cousins.)

In Java, assignment is itself an operator and can be built into an expression: as a result we can write things like;

```
int a, b, c;

a = b = c = 35;
```

which initialises all three variables to 35: the 'result' of an assignment operator is the value assigned. The assignment operators are *right* associative which means that a string of assignments is evaluated right-to-left, not left-to-right. So in detail, this code fragment means:

```
int a, b, c;

(a = (b = (c = 35)));
```

Now there's a curious thing going on here. Consider the expressions

```
a = 35;

a = a + 1;
```

Mathematically, this looks very odd: we seem to be suggesting that `a` is itself plus one, which can never be true. Actually, though, the `=` operator stands for assignment, not equality, and the way this operation executes is that `a + 1` is evaluated to yield `36` which is then written into the box associated with variable `a`. We say that the writing of the value is a *side effect* of the expression. Side effects turn out to be both very useful, and the source of some confusing errors.

Assignment abbreviations

We often find ourselves writing things like

```
a = a + 35;
```

To make things more concise, Java has a family of *op-and-becomes* operators: we can rewrite this fragment as

```
a += 35;
```

which we read as *a plus-and-becomes 35*.

Increment and decrement operators

Adding or subtracting one are such common operations that they have special forms: `++` and `--`. These operators have the peculiar property that they can be used in both *prefix* and *postfix* forms. If you use the prefix form, the variable is incremented (decremented) and the result returned is the final value. If you use the postfix form, then the value of the variable before incrementing (or decrementing) is used in the rest of the expression, but the variable's value itself is adjusted.

```
int a = 3;

int b = ++a;

int c = a++;
```

In this program, `a` is declared and initialised to 3. `b` is then declared and initialised to the result of `++a`, the pre-increment version of the `++` operator. `a` is pre-incremented to `4`, and `b` then is assigned `4`. The variable `c` is then declared and initialised to the result of `a++`. This means that `c` is assigned `4`, and then `a` is incremented to `5`. This can all get quite confusing. Just to really make the point, have a think about the value of `a` after running this program:

```
class MixedInc
{
  public static void main(String[] args)
  {
    int a = 3;

    a += a++;

    System.out.print("a is " + a);
  }
}
```

The result of this is 6. Why? Well, `a` is initialised to `3`. The right hand side of the assignment is then evaluated to `3` (because that is the pre-increment value of `a`). Java remembers that value whilst it post-increments `a` to 4. But Java then assigns the original value of `a` plus the result of the expression back into `a`, making `6`. This is too hard, and really we should avoid writing expressions like this: good programs should illuminate, not obfusticate, the intentions of the writer. In case you'd like further examples, what do you think this fragment prints out?

```
int a = 3;

a -= --a - a--;

System.out.print("a is " + a);
```

2.10 Programming languages and their use of data types

A computer has completely separate operations for performing, say, integer division and floating point division. It is as if the calculator used by our human programmer had two buttons labelled / one for integers and one to be used if the numbers to be divided, or the result of the division, had decimal places.

It would be quite tiresome to have to specify which kind of arithmetic we wanted to use each time we wrote an expression. Instead, we write down the names of the quantities to be manipulated in advance, and specify which type of data they are, and by extension which kind of operation the computer should use.

So, for instance, this fragment of program

```
double x = 5, y = 3, z;

z = x / y;
```

specifies that `x`, `y` and `z` are all double precision floating point numbers, and thus that the division to be performed is floating point division.

This (almost) identical program specifies the use of integer division instead.

```
int x = 5, y = 3, z;

z = x / y;
```

The integer division operation always returns an integer, but which integer? You might hope that the result of integer 5 divided by integer 3 would be 2, since in real number division the result would 1.666... and normal integer rounding would then yield 2. In fact, integer division in Java (and actually in nearly all computer languages) works by *truncating* the result, so the result will be 1.

2.10.1 The dangers of mixed mode arithmetic

In Java we are required to declare the type of all variables, but not the type of literal numbers. This can lead to some subtle errors because Java attempts to infer which operation we want from the types of the operands of an operator. The basic rule is that Java will use integer arithmetic until it has to force a conversion to a real number format.

A well known way to introduce a bug into your program is to be cavalier with your literals. Look at this program:

```
class Mixed
{
  public static void main(String args[])
  {
    double x = 5 / 3;
    System.out.println("Result:" + x);
  }
}
```

The result of running this program is:

```
Result:1.0
```

This is because `5 / 3` is an integer expression yielding integer 1. Java converts this to a double precision floating point number as it is being assigned to the variable `x`. The way to get the more conventional result is to use real literals:

```
class Mixed1
{
  public static void main(String args[])
  {
    double x = 5.0 / 3.0;
    System.out.println("Result:" + x);
  }
}
```

which yields:

```
Result:1.6666666666666667
```

Now, this is fine for literals, but what about integer variables? Well it turns out that we can *force* Java to convert integer variables to real with a *cast* operation. A cast is written as the type we want to convert to, in parentheses:

```
class Mixed2
{
  public static void main(String args[])
  {
      int p = 5, q = 3;
      double x = (double) p / (double) q;

    System.out.println("Result:" + x);
  }
}
```

Happily, the result of running this program is:

```
Result:1.6666666666666667
```

2.11 Why declare everything?

Although this need to *declare* the type of variables in advance is a significant clerical overhead, programmers soon found that the discipline of giving advance notice forced them to think carefully about what they were doing, and that it enabled the compiler to catch certain kinds of common programming errors

such as an attempt to do arithmetic on strings of characters, rather than their numeric equivalents, as we saw in Section 1.5.

The technique was so useful, that by the early 1960's programming languages began to appear that allowed programmers to define their *own* data types which were specific to a particular program. In time this led to the development of *Object Oriented* programming languages in which data types also came with user-specifiable methods. We'll talk a lot more about this in Chapter 4.

2.12 Formal data types

Each data type has four main characteristics:

range – the set of values that can be represented: an eight-bit positive integer type for instance would have a range of 0, 1, 2, . . . , 127;

size – the amount of the computer's memory that is needed to hold one value from the range: for an eight-bit number this would be one byte;

default value – the particular value that a newly declared variable holds: for some languages there is no default;

operations – the set of operators and methods that we can use with that data type: for numbers we would expect basic arithmetic but we would not expect to be able to add characters together.

Real languages provide a set of *primitive* types which can be built up by the programmer into more complicated types. For instance, nearly all languages provide a floating point type: by using the mechanisms that we shall discuss in Chapter 4 we can put two of these together to represent a coordinate pair which is very useful for graphics-drawing applications.

2.13 Java primitive types

Java provides eight basic types which can be used to represent integers, floating point numbers, characters and Booleans. In addition, there is special support for *strings* of characters and a type `void` for declaring methods that do not actually return a value. The Java primitive types are summarised in Table 2.2.

2.13.1 Integral number types

There are four types for integral numbers: `byte`, `short`, `int` and `long` corresponding to one-, two-, four- and eight-byte quantities respectively. In detail, it turns out that the JVM usually actually allocates four bytes for both `short` and `int` values, so no actual memory saving arises from using `short` variables.

Literal (constant) integer values are written without a decimal point, as you would expect. By default, the compiler treats literals like 123 as four-byte

Name	Size	Range	Default value
`byte`	1	-128 ... 127	`0`
`short`	2	-32 768 ... 32 767	`0`
`int`	4	-2 147 483 648 ... 2 147 483 647	`0`
`long`	8	-9 223 372 036 854 775 808 ... 9 223 372 036 854 775 807	`0`
`float`	4	$\pm$ 1.401 298 464 324 817 07$\times 10^{-45}$ ... 3.402 823 466 385 288 60$\times 10^{38}$	`0.0`
`double`	8	$\pm$ 4.940 656 458 412 465 44$\times 10^{-324}$ ... 1.797 693 134 862 315 70$\times 10^{308}$	`0.0`
`boolean`	1	`false`, `true`	`false`
`char`	2	`0x0000` ... `0xFFFF`	`0x0000`
`void`			

Table 2.2 Java primitive types

`int` values. This can cause problems in expressions declarations like `long x = 1024 * 1024 * 1024 * 10;` because the calculation on the right hand side of the = sign will be done using 32-bit arithmetic, leading to an overflow. You can force the compiler to use 64-bit `long` arithmetic by appending the letter `L` to a literal: `long x = 1024L * 1024L * 1024L * 10L;`

2.13.2 Floating point number types

Two floating point types are provided: single precision (`float`) and double precision (`double`) which uses twice as many bits so as to reduce (but not eliminate) rounding errors. Given all that we have learned about the instability of floating point calculations, you will not be surprised to find that most programmers use `double` for all floating point calculations, and we recommend that you do to.

In the same way that we can use a trailing `L` to specify that a numeric literal is a `long` value, we can use `F` and `D` suffixes to specify whether a literal is a `float` or a `double`.

Just as we use *scientific notation* to represent large numbers on paper, we can supply an exponent with a floating point literal: the number 1.3×10^6 may be written as either `1600000` or `1.6E6`, and the number 6×10^{-3} may be written as `0.006` or `6E-3`.

(Suffixes, and the exponent letter may be written as lower case, but we recommend that you use upper case exclusively.)

2.13.3 Booleans

Booleans only need one bit to represent them, but a JVM is free to decide how to store a Boolean. Typically one byte per Boolean will be used. The Boolean literals `false` and `true` are keywords in the language and should not be used as variable names.

2.13.4 Characters and strings in Java

Java uses sixteen-bit Unicode to represent individual characters, so each character needs two bytes of storage. A character literal is written between *single* quote marks like this: `'a'`.

A string literal may be written between *double* quotes like this: `"adrian"` .

Within string and character literals, there are some special *escape sequences* that can be used to represent characters that do not appear on a normal keyboard. They all begin with a backslash (\).

Sequence	Character
`\b`	backspace
`\f`	form feed (new page)
`\n`	newline
`\r`	carriage return (without advancing to a new line)
`\t`	horizontal tab
`\'`	the single quote character '
`\"`	the double quote character "
`\\`	the backslash character \
`\U`*xxxx*	Unicode character with hexadecimal number *xxxx*

We can concatenate strings with the `+` operator, and in fact we have already made extensive use of this feature within `println()` statements.

2.14 Allocating space for non-primitive types

The primitive datatypes (simple numbers, Booleans and characters) have the property that when they are declared, sufficient space is automatically reserved inside the computer for them. Arrays, and other kinds of objects that we shall encounter in later chapter need to be explicitly allocated using the `new operator`. The best way to understand this is that the variable that represents the array (or other structure) is a simple box which contains the machine address of the real data.

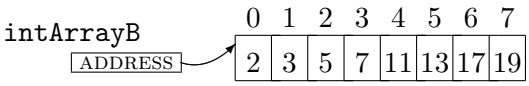
Let's revisit the array example from Chapter 1.

```
intArrayB = {2, 3, 5, 7, 11, 13, 17, 19}
```

Earlier, we claimed that after initialisation, `intArrayB` looks like this: eight cells indexed 0...7 containing the first eight prime numbers.

0	1	2	3	4	5	6	7
2	3	5	7	11	13	17	19

In detail, the situation is more complex. The variable `intArrayB` corresponds to a box in memory which contains the address of the first location of the array of integers. There's no actual data in `intArrayB`, merely the name of the place where the data is.



This notion of having variables that refer to other bits of memory is very powerful, and is the basis for data structures that can expand and shrink whilst a program is running – so-called *dynamic* data structures. We can allocate memory with `new` whenever we want, and to make things easy Java cleans up after us by releasing memory that we no longer need. This process is called *garbage collection*. We will discuss dynamic data structures at length in Chapter 5.

2.15 Enumerations

It is common when programming to need to represent small sets of symbolic values: the days of the week (`Monday, Tuesday, ... Sunday`) or type of student (`Undergraduate, Masters, PhD`), for instance.

One way to do this is to create a series of integer variables and initialise them with unique numbers like this:

```
int Monday = 1, Tuesday = 2, Wednesday = 3,
    Thursday = 4, Friday = 5, Saturday = 6,
    Sunday = 7;

int Undergraduate = 1, Masters = 2, PhD = 3;
```

This works, but we can do better. One of the problems of this approach is that it allows certain programming errors, such as adding together two days of the week that just don't make sense.

Java enumerations allow us to define a completely new data type whose values are symbols: just as the `Boolean` data types values are the symbols `false` and `true`.

```
enum DayOfTheWeek {Monday, Tuesday, Wednesday,
                   Thursday, Friday, Saturday,
                   Sunday};

enum StudentKind {Undergraduate, Masters, PhD};
```

There aren't many operations defined automatically for `enum` types (after all, would it necessarily make sense to add together two days of the week?) but later we shall see that there is a mechanism that we can use to create methods for enumeration types.

This is the first time we have seen a user-defined data type. We will discuss these much more in Chapter 4, but for now all you need to know is that it is as if we added types `DayOfTheWeek` and `StudentKind` to the eight primitive types

shown in Table 2.2, each with the range of values enumerated in braces. We can declare variables whose type is `DayOfTheWeek` or `StudentKind` and then assign values from the corresponding enumeration.

```
enum DayOfTheWeek {Monday, Tuesday, Wednesday,
                   Thursday, Friday, Saturday,
                   Sunday};

enum StudentKind {Undergraduate, Masters, PhD};

DayOfTheWeek today = Monday;

StudentKind joe = PhD;
```

If we try to write something like `joe = Tuesday;` the compiler can now catch our error because `Tuesday` is not a member of the enumeration type for variable `joe`.

2.16 Exceptions arising from runtime type errors

Some operations have undefined results. For instance, mathematically we do not know what the result of dividing a number by zero is, and any program that tries to do such a thing is probably erroneous.

Let's modify the program from Section 1.5 to take the quotient of two integers.

```
import java.util.Scanner;

class DivideIntegers
{
  public static void main(String[] args)
  {
    Scanner keyboard = new Scanner(System.in);
    int first, second;
    int quotient;

    System.out.print("First number? ");
    first = keyboard.nextInt();
    System.out.print("Second number? ");
    second = keyboard.nextInt();
    quotient = first / second;
    System.out.print("Quotient is " + quotient);
  }
}
```

Here is the result of two runs of the program.

```
> java DivideIntegers
First number? 10
Second number? 5
Quotient is 2
>
> java DivideIntegers
First number? 10
Second number? 0
Exception in thread "main" java.lang.ArithmeticException:
        / by zero
        at DivideIntegers.main(DivideIntegers.java:15)
```

When we enter a divisor of zero, Java attempts to divide the `10` by `0` and stops the program, telling us that a `/ by zero` exception was raised at line 18 of the program. In Chapter 3 we shall find out how to catch these exceptions so that this kind of error can be handled without causing the program to crash.

The equivalent floating point program shows different behaviour.

```
import java.util.Scanner;

class DivideDoubles
{
  public static void main(String[] args)
  {
    Scanner keyboard = new Scanner(System.in);
    double first, second;
    double quotient;

    System.out.print("First number? ");
    first = keyboard.nextDouble();
    System.out.print("Second number? ");
    second = keyboard.nextDouble();
    quotient = first / second;
    System.out.print("Quotient is " + quotient);
  }
}
```

When we run it, no exception is raised.

```
> java DivideDoubles
First floating point number? 10.0
Second floating point number? 5.0
Quotient is 2.0
>
> java DivideDoubles
First floating point number? 10.0
Second floating point number? 0.0
Quotient is Infinity
>
> java DivideDoubles
First floating point number? -10
Second floating point number? 0
Quotient is -Infinity
>
```

The Java floating point representation includes some special values, including `Infinity` which is the result of a divide-by-zero operation. Since this is a 'normal' value within the range of the data type, no exception is raised.

3 Control

Control flow governs the order of operations

Nearly all of the programs we have written so far have been straightforward *sequences* of instructions to be executed in order: the program is finished, and thus terminates, after the last operation has been performed.

The real power of computers derives from their ability to make choices, selecting between and iterating over operations according to tests made during program execution. There's a sense in which a program as written describes all the allowed ways in which the operations might be put together or rather, all the allowed routes through the operations. Any given run of the program will correspond to just one of those routes.

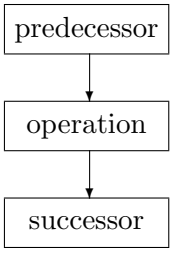
A straightforward sequence of operations has only one possible route, so we can immediately tell just from the program what should happen when the program runs. Most real programs use selection and iteration and are much harder to analyse in advance of their running. Sometimes we cannot even be sure that a program will ever reach the end and terminate. Sometimes we can be sure that it will not!

3.1 Control flow statements

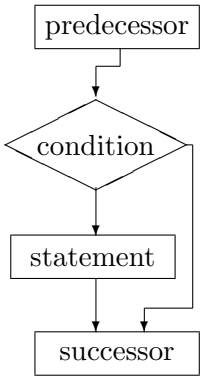
In the last chapter we concentrated on data and operators. We saw how to declare variables of particular types, how to write literals for various types and how to combine them into expressions using operators. Most of our programs so far have been simple sequences of expressions, each terminated with a semicolon.

Real programs are mostly composed of declarations, expressions and *control flow statements* which specify which operation should be performed next. In Chapter 1 we saw four kinds of control paths of which a sequence was one. Here's a reminder of the others.

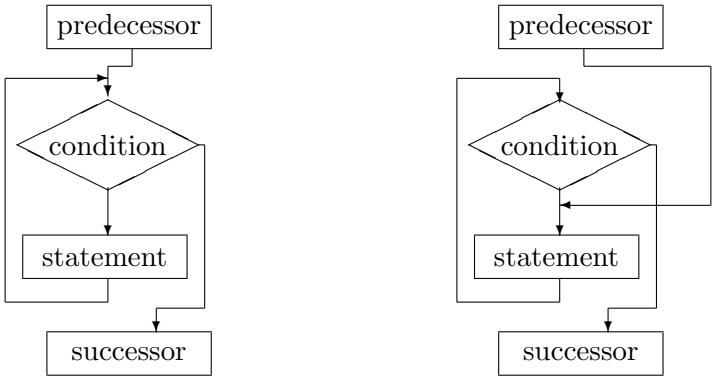
A *sequence* is a list of actions that will always be performed one after another.



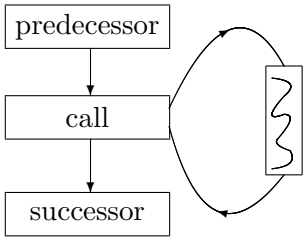
A *selection* involves making a test, and then performing one of a set of actions depending on the result. The other actions are ignored.



An *iteration* repeats an action as long as a test is passed. Sometimes we do the action first, and then test whether to repeat. The alternative is to do the test first, and if it passes do the action.



A *call* tells us to stop what we're doing, go and perform some other subprogram, and then come back and carry on where we left off.



3.1.1 Structured programming

This section is for interest only: you do not need to learn it for the exam.

In the early days of computing, programmers were very undisciplined in the way that they specified the possible routes through their programs. They

used so-called `goto` statements which allowed control to be transferred after any statement to any other statement.

By the 1960's, people had realised that this unrestricted approach to control flow allowed some very hard-to-read programs to be constructed. A new principle was established: that control flow constructs should have a single entry and a single exit. You'll notice that each of our diagrams here has a single predecessor and a single successor node: the arrows connecting to these represent the single entry/single exit model. One thing you are not allowed to do, for instance, is jump straight into the middle of a loop. This style of control flow is sometimes call *structured programming* and the structures themselves are called D-structures in honour of Edsger Dijkstra who led the rejection of the older style of control flow. D-structures nest, so quite complicated patterns can be built.

3.2 Describing syntax

We need to be precise about the way that statements are to be written. The *syntax* of a language (human or computer) is the set of rules that govern how words may be put together. For instance, in French it is a general rule that adjectives come after nouns but in English we usually put the adjective in front of the noun: so *The red house* in English translates as *La maison rouge*.

We can use a set of templates, or *grammar rules* to capture these kinds of effect. Here are some rules that describe adjective placement in English and French:

```
englishPhrase ::= The englishAdjective englishNoun

englishAdjective ::= red
englishAdjective ::= black

englishNoun ::= house
englishNoun ::= car
```

```
frenchPhrase ::= frenchArticle frenchNoun frenchAdjective

frenchArticle ::= Le
frenchArticle ::= La

frenchAdjective ::= rouge
frenchAdjective ::= noir
frenchAdjective ::= noire

frenchNoun ::= maison
frenchNoun ::= voiture
```

Each rule has an italicised left hand side identifier followed by the `::=` symbol which we read as 'expands to' followed by a mix of one or more italicised and non-italicised items. The idea is that each rule is a sort-template for things that we are allowed to write: the non-italicised items (called *terminal symbols* or just terminals) have to appear exactly as written. The italicised items (called *nonterminal symbols* or just nonterminals) are placeholders for other rules. One of the nonterminals is the *start symbol* and we make valid sentences by expanding the nonterminals through substitution. In the two examples here, the start rules are *englishPhrase* and *frenchPhrase*.

Notice that there might be several rules with the same left hand side nonterminal: these rules are alternatives and when we want to expand a nonterminal we select exactly one of them to use as a replacement.

Notice also that these *grammars* cannot always capture every aspect of phrase construction. For instance, in French there are feminine and masculine gendered nouns: maison is masculine and voiture is feminine. The article (le or la) must match the gender, but our rules allow them to be interchanged. It is possible to write a more complicated grammar that forces the matching in this case, but it turns out that there are some language constructs in human languages which genuinely cannot be accurately described this way. In programming languages, we try very hard to keep things simple.

We're going to use the same grammar notation to describe the shape of valid Java phrases. In fact we have already been doing it without really explaining what we're up to: it's such a straightforward notation. Remember that the non-italicised terminals represent symbols that must appear directly as written in the Java program, and the italicised nonterminal symbols are place markers for things rules that will be found elsewhere.

3.3 Sequences and the compound statement

In a sequence, the operations are executed one after another, reading from left to right and down the page. So

```
x = 3 *4; y = 2* 7;

z = x + y;
```

and

```
x = 3 *4;
y = 2* 7; z = x + y;
```

mean exactly the same thing.

In detail, any list of expressions, declarations and statements written within curly braces is a separate sequence or *compound statement*.

```
compound-statement ::=  operation-list

operation-list ::= declaration ; operation-list
operation-list ::= expression ; operation-list
operation-list ::= statement ; operation-list
operation-list ::= ϵ
```

There is some cleverness here. The rule *operation-list* has four parts, one of which ϵ which means 'empty'. The other three are all of the form

```
operation-list ::= something ; operation-list
```

that is, a statement, declaration or expression terminated with a semi-colon and then another *operation-list*.

These kinds of self-referential rules are called *recursive rules*. We shall return to recursion later in this chapter, but for now try to see that our sequence contains an operation, and our rule for an *operation-list* expands to, say, a *declaration* then another *operation-list*. The empty rule for *operation-list* stops this expansion process going on for ever.

It's not an accident that the body of our `main()` methods is enclosed in braces: the body is a sequence of operations.

3.4 Selection statements

Selection statements are used to pick one amongst several alternative routes through the program. We make the selection by testing the value of some expression. A *predicate* is an expression which yields a Boolean truth value: typically some sort of comparison. If we declare and initialise the following

variables

```
int x = 3;
boolean a = true, b = false;
```

then these expressions are all predicates:

```
x == 3; // equality - yields true
a | b;  // logical inclusive OR - yields true
a & b;  // logical AND - yields false
```

There are two main kinds of selection statement in Java: `if` statements which use predicates to choose between two routes, and `switch` statements that match the result of an integral expression (one that returns a value of type `byte, short, char, int` or an enumeration) to select one of many potential routes.

3.4.1 `if (` *predicate* `)` *statement* `;`

The simplest `if` statement takes a single predicate and a single statement called the *then-clause*. When the computer encounters the `if` statement it evaluates the predicate and then only executes the then-clause if the predicate yields true.

```
int x = 7;

if (x == 3)
  System.out.println("x is three");

if (x != 3)
  System.out.println("x is not three");
```

In this case the program will output

```
x is not three
```

It is *terribly* easy to get the assignment operator `=` muddled up with the *is-equal-to* operator `==`. Most of the time the compiler will catch you if you muddle them up, because the result of `x = 3` is an integer, and the `if` statement expects a boolean expression in its predicate: this is a nice example of strong type checking working so as to save programmers from simple errors.

3.4.2 `if (` *predicate* `)` *statement* `else` *statement* `;`

The kind of programming idiom in which we test a predicate and then perform two different actions depending on the outcome is very common. Rather than having two `if` statements with inverted predicates (as we've shown in the

previous example) we use an *else-clause* like this:

```
int x = 7;

if (x == 3)
  System.out.println("x is three");
else
  System.out.print("x is not three\n");
```

Quite apart from being simpler to write, this version is more efficient than the version with two `if` statements. It takes time for the computer to evaluate a predicate, and this version has only one predicate evaluation compared to the two in the previous version.

Note, by the way, the use of the escape sequence `\n` at the end of the string in the `else` clause to force `System.out.print()` to output a newline character.

3.4.3 Using sequences to control blocks of operations

Real programs usually require more than just single statements to be turned on and off. We said before that the D-structures can *nest*. We can put an entire sequence of operations together between curly braces, and use that as a then-clause or an else-clause.

In this example, we not only print out a message, but we remember the result of the predicate in a boolean flag variable for later use.

```
int x = 7;
boolean flag;

if (x == 3)
{
  System.out.println("x is three");

  flag = true;
}
else
{
  System.out.print("x is not three\n");

  flag = false;
}
```

Some people recommend that then-clauses and else-clauses should always be surrounded by braces, even if they are only a single statement long. This is a safe rule for new programmers, but in our programs we tend to not put the braces in unless strictly necessary.

3.4.4 Nested `if` statements

D-structures can nest so the following is a legal program.

```
int x = 3, y = 7;

if (x == 3)
{
  if (y > 10)
    System.out.println("x is three and y is greater than ten");
  else
    System.out.println(
      "x is three and y is less than or equal to ten"
                      );
}
else
  System.out.print("x is not three\n");
```

This program will print out `x is three and y is less than or equal to ten` because the `x == 3` predicate for the outer `if` statement returns true, so the inner `if` statement in the then-clause will be executed. Its predicate returns false, so the inner else-clause is performed.

3.4.5 Chained `if` statements

It is quite common to want to test the value of a variable against a whole sequence of values. Let's imagine we have an `int` variable called `dayOfTheWeek` which holds a number between 1 and 7, with 1 representing Monday and 7 representing Sunday. We can 'decode' this value and print an appropriate

message like this:

```
int dayOfTheWeek = 5;

if (dayOfTheWeek == 1)
  System.out.println("Monday");
else if (dayOfTheWeek == 2)
  System.out.println("Tuesday");
else if (dayOfTheWeek == 3)
  System.out.println("Wednesday");
else if (dayOfTheWeek == 4)
  System.out.println("Thursday");
else if (dayOfTheWeek == 5)
  System.out.println("Friday");
else if (dayOfTheWeek == 6)
  System.out.println("Saturday");
else if (dayOfTheWeek == 7)
  System.out.println("Sunday");

else
  System.out.println("Error: illegal value for dayOfTheWeek");
```

Technically, this is a nest of `if` statements, but it is easier to think of it as a ladder or chain. Really we are just sequentially testing the values until we either find one that works (in which case we print the text form of the day) or until we get to the final else-clause. In that case, we have failed to find any matching value, so the `dayOfTheWeek` variables must have a value that is less than 1 or greater than 7. Neither of these are allowed!

There is a subtle point here. Remember that `int` variables have a default value of zero. It is *very* good programming practice to allocate integer codes in such a way that zero is not an allowed value, as we have done here. That way, the common programmer error of forgetting to initialise a variable will be caught as soon as we try to print out the value of dayOfTheWeek. So, the rule is: allocate values starting from one, not from zero.

3.4.6 The `switch` statement

A chained `if` statement is really a multiway branch. If, as above, the branch is chosen on the basis of testing the result of an expression against a sequence

of integral values we can use the more compact `switch` statement instead.

```
int dayOfTheWeek = 5;

switch (dayOfTheWeek)
{
  case 1: System.out.println("Monday"); break;
  case 2: System.out.println("Tuesday"); break;
  case 3: System.out.println("Wednesday"); break;
  case 4: System.out.println("Thursday"); break;
  case 5: System.out.println("Friday"); break;
  case 6: System.out.println("Saturday"); break;
  case 7: System.out.println("Sunday"); break;

  default:
    System.out.println("Error: illegal value for dayOfTheWeek");
}
```

Each `case` statement labels a *case-clause*. The switch statement works by evaluating the control expression (which in this case is the single variable `dayOfTheWeek`) and then finding a matching label amongst the `case` statements.

If no match is found, the statement labelled with `default:` is used: this is like the trailing `else` in our chained `if` statement.

In this example, each case-clause finishes with a `break` statement. `break` tells the computer to jump to the statement after the `switch` statement: the successor statement. If you do not put the `break` in, then the computer will go on to the next case-clause and execute that as well!. It is unusual to want this to happen, so you need to be very disciplined: forgetting to put in a `break` can lead to some surprising results.

There are occasions, though, when you deliberately do this. Essentially, if two or more case-clauses have the same action, then they could be grouped

together. Here's a simple example.

```
int dayOfTheWeek = 5;

switch (dayOfTheWeek)
{
  case 1: System.out.println("Monday"); break;
  case 2: System.out.println("Tuesday"); break;
  case 3: System.out.println("Wednesday"); break;
  case 4: System.out.println("Thursday"); break;
  case 5: System.out.println("Friday"); break;
  case 6:
  case 7: System.out.println("Weekend"); break;

  default:
    System.out.println("Error: illegal value for dayOfTheWeek");
}
```

The program prints out `Weekend` whenever `dayOfTheWeek` is 6 or 7. It is as though there are two labels for the last case-clause.

3.4.7 Chained `if` or `switch`?

Use switch statements where possible: not only are they easier to read than chains of `if` statements, but they are faster too. The chained version has to evaluate seven predicates before printing out `Sunday`, and in general the cases lower down in the chain will take longer to process than those near the top. A `switch` statement only evaluates its expression once, and then jumps straight to the relevant case-clause.

However, `switch` statements are limited to expressions yielding a single value. If you want different actions dependent on a combination of values or ranges of value, then you need a chained `if`. For instance, there is no easy way to convert this chain to a `switch` statement.

```
int x = 10, threshold = 15;

if (x < 0)
  System.out.println("Negative");
else if (x < threshold)
  System.out.println("Less");
else if (x == threshold)
  System.out.println("Equal");
else if (x > threshold)
  System.out.println("Greater");
```

3.5 Iteration statements

Iteration or *loop* statements allow us to do things repeatedly. There are three main kinds of loop in Java: the `while` loop, the `do while` loop which always executes its body at least once and the `for` loop which allows us to set up *induction* variables which count the number of times we have been round the loop.

3.5.1 `while ( `*predicate*` ) `*statement*` ;`

The `while` statement is very closely related to the `if` statement. It takes a predicate and a statement called the *body* of the loop. The predicate is evaluated, and if it is `true` the body is executed once. The computer then goes back to the beginning of the loop, re-evaluating the predicate and executing the body until the predicate returns `false`.

```
int x = 15; y = 20;

while (x < y)
{
  System.out.println("x: " + x);
  x = x + 1;
}
```

In this example, x steps from 15 through to 19 inclusive, with the values being printed out as we go.

3.5.2 `do `*statement*` while ( `*predicate*` ) ;`

Sometimes it is useful to perform the predicate test at the *bottom* of the loop instead of on entry. The `do while` statement lists the body before the predicate.

```
int x = 15; y = 20;

do
{
  System.out.println("x: " + x);
  x = x + 1;
}
while (x < y)
```

On the face of it, this loop might seem completely equivalent to the previous `while` loop: x again steps from 15 through to 19 inclusive, with the values being printed out as we go. However, think about what happens if the variable `x` is initialised to 30. This `do while` loop will print out `x:  30` whereas the `while` loop version will print nothing.

This is because the body of a `do while` loop will *always* be executed at least once.

Most programmers work with `while` and `for` loops most of the time: the `do while` construct is a bit like a case-clause with no break: very useful on occasion but you need to be really sure you know what you are doing when you write one!

3.5.3 `for (`*initExpression* `;` *predicate* `;` *stepExpression*`)` *statement* `;`

The two previous loop examples feature a 'counting' or *induction* variable which keeps track of the number of loop iterations. This is such a common idiom that Java provides a special statement that lists the initialisation, predicate and variable update clauses together. Loops expressed as `for` statements are by far the most common kind of loop found in real Java programs.

```
int x, y = 20;

for (x = 15; x < y; x = x + 1)
  System.out.println("x: " + x);
```

This is certainly more compact than its `while` loop equivalent, and it brings together all of the information about the loop in one place. We can see what the initial state of the induction variable will be, what the loop termination condition is and how the step is applied to the induction variable.

In fact the `for` statement is quite general and the *initExpression*, *predicate* and *stepExpression* can be any valid expression, including ones with side effects. In detail, a `for` statement like

```
for (initExpression ; predicate ; stepExpression)
  statement ;
```

can always be written as

```
initExpression ;
while(predicate)
{
  statement ;
  stepExpression ;
}
```

3.5.4 Declaring induction variables within the loop

If the induction variable is not being used outside of the loop, it is good practice to declare it as part of the `for` statement itself.

```
int y = 20;

for (int x = 15; x < y; x = x + 1)
  System.out.println("x: " + x);
```

The induction variable `x` will not be accessible outside of the loop: it's value is discarded and any attempt to use `x` in an expression will result in a compiler error message.

3.5.5 Empty clauses in the `for` statement

None of the three control clauses – *initExpression, predicate, stepExpression* – is compulsory: we can leave them blank as long as we put in the separating semicolons. In fact this short program

```
for (;;) System.out.println("not reached infinity yet");
```

will loop forever. This sort of *infinite loop* is usually an error, but the outer loop of programs like those which run vending machines or televisions may be an infinite loop, because those programs are supposed to run for the lifetime of the device they are controlling. (We call such systems *embedded* computers, by the way, to distinguish them from *general purpose* computers which run lots of different programs during their lifetime.)

It should be obvious to you that any `while` loop can be simply rewritten as a `for` loop:

```
while (predicate) statement;
```

translates to

```
for ( ; predicate ; ) statement;
```

Opinions vary as to which of these is best to use. We favour using `while` statements wherever the termination of a loop depends on a simple boolean expression, and `for` statements wherever there is an associated induction variable.

Here's a more natural example of the use of `for` statements with missing clauses. In this case we are going to share an induction variable between two

loops.

```
int x, y = 20;

for (x = 15; x < y; x = x + 1)
  System.out.println("x: " + x);

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

for (; x > 10; x = x -1)
  System.out.println("x: " + x);
```

Here, the program prints out all the values from the initial value of `x` up to 19, and then from 20 down again to 11. The induction variable's final value from the first loop is used as the start value of the second loop.

The output of this program fragment is

```
x: 15
x: 16
x: 17
x: 18
x: 19
***
x: 20
x: 19
x: 18
x: 17
x: 16
x: 15
x: 14
x: 13
x: 12
x: 11
```

You need to be very careful when writing the termination predicates. Remember that the test is done at the entry to the loop, just as in a `while` loop (but *not* as in a `do-while` loop. Remember also that the loop is terminated as soon as the predicate is true, so a predicate of `x>10` means that the loop will not execute when `x` is ten! That's why the countdown stops at 11.

3.5.6 Using the ++ and -- operators

Most experienced programmers would rewrite the previous program in this way.

```
int x, y = 20;

for (x = 15; x < y; x++)
  System.out.println("x: " + x);

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

for (; x > 10; x--)
  System.out.println("x: " + x);
```

By and large, the `++` and `--` operators present no problems if used on their own in an expression, but see our discussion in section 2.9 for warnings about strange effects when they are used in combination with other operators.

3.5.7 `for (` *type var* `:` *collection*`)` *statement* `;`

There is a special form of the `for` statement for stepping through the elements of a *collection*. An array is the simplest form of collection: Chapter 5 discusses other kinds in depth. We'll return to this form of the `for` loop there.

3.5.8 Aborting loops with `break`

Sometimes we need to, for instance, search the contents of an array for a particular value. One way of doing this is to step through the elements of an array testing each, and then break out:

```
int arr[] = {45, 23, 17, 99, 10};

for (int x = 0; x < arr.length; x++)
  if (arr[x] < 20)
  {
    System.out.println("Found element < 20 at index " + x);
    break;
  }
```

The `break` statement tells the computer to 'break out' of the enclosing `for` loop, so at the end of the loop the induction variable `x` will contain 2 rather than 5.

Inside a `switch` statement, the `break` causes control to jump to the successor statement of the enclosing `switch`, in a loop `break` causes control to jump to the successor statement of the enclosing `for`.

Breaking out of nested loops

Sometimes we need to break out of deeply nested loops. We can do this by *naming* a loop with a label, and then using a *named break*:

```
int arr[][] = {{45, 23, 17, 99, 10},
               {32, 43, 78, 100},
               {13, 17, 9}
              };

  outer:
for (int y = 0; y < arr.length; y++)
  for (int x = 0; x < arr[y].length; x++)
    if (arr[y][x] < 20)
    {
      System.out.println("Found element < 20 at
                         ( " + y + ", " + x + ")"
                        );
      break outer;
    }
```

Here we have defined an array of arrays: note the nested initialiser. In detail, we have a column of three arrays of lengths five, four and three – there's nothing that forces all of the inner arrays to be the same length. We have then defined two nested `for` loop: one controlled by the `y` induction variable which scans up the column, and another controlled by `x` which scans along the elements of a inner array.

The basic program idea is the same as before: stop when you find an element that is less than 20. This time though we need to break out of two nested loops. We name the outer loop `outer:` by placing a label immediately before the `for` keyword, and then we can use the statement `break outer` to pass control to the statement following the `outer` loop body.

3.5.9 Skipping to the next iteration with `continue`

A `break` statement aborts a complete loop: the `continue` statement aborts one pass of a loop, transferring control to the *`stepExpression`* if there is one and thence to the test.

A loop of the form

```
for (initExpression ; predicate ; stepExpression )
{
    allStatement
    if ( continuePredicate )
        continue;


    someStatement
}
```

is equivalent to

```
for (initExpression ; predicate ; stepExpression )
{
    allStatement
    if ( !  continuePredicate )
        someStatement
}
```

so you might wonder whether `continue` is ever useful. Well, sometimes we want to break out of a loop iteration from some deeply nested bunch of `if` or `switch` statements in which case it is extremely useful, but it is probably still true that `continue` is one of the least used features of Java. You can use `continue` with a label to continue from the next iteration of a surrounding loop just as you can use `break` with a label.

3.6 Method calls

The fourth kind of control flow construct is the *call*. We read in Section 1.1.3 that an important way to manage complexity in large programs is to break the code down into small pieces called *methods*. In this section we are going to look at what are called *static* methods and a style of programming called *procedural programming*. In the next chapter we will learn about methods in the context of *object oriented* programming.

Here's a simple example of using a method to encapsulate a small piece of

code that is used in two places.

```
class Power
{
  static double square(double x)
  {
    return x * x;
  }

  public static void main(String args[])
  {
    double x = 2.1;

    System.out.println(x + " squared is " + square(x));
    System.out.println(2 * x + " squared is " + square(2 * x));
  }
}
```

Apart from just parcelling up code, the really interesting things about methods is that they can take parameters and can return a result. Our method `square()` takes a `double` called `x` and returns a `double` value which is the square of `x`.

Here's an expanded version that builds higher powers by reutilising the `square()` method.

```
class Power
{
  static double square(double x) { return x * x; }

  static double cube(double x) { return x * square(x); }

  static double pow4(double x) { return square(square(x)); }

  public static void main(String args[])
  {
      double x = 2;

      System.out.println(x + " squared is " + square(x));
      System.out.println(x + " cubed is " + cube(x));
      System.out.println(x+" to the fourth is "+pow4(x));
  }
}
```

It would become tedious to have to write a new method for each integral power of `x` that we might want, so let's try a general purpose method which

takes both the `double` value to be exponentiated and an `int` power.

```
class PowerIter
{
  static double powIter(double x, int power)
  {
    double result = x;

    for (int pow = 1; pow < power; pow++)
      result *= x;

    return result;
  }

  public static void main(String args[])
  {
      double x = 2;

      System.out.println(x+" to the fourth is "+powIter(x, 4));
  }
}
```

3.6.1 The syntax of method declarations and calls

Methods take parameters and return a single value (which may be of any type). There is a special type `void` which means 'no-value' and we use that for the return type of methods that do not return values.

When we define a method, we list the parameters giving both their name and their class type. The names are placeholders that are filled with actual values when the method is called.

```
method-definition ::= optional-static return-type identifier
( parameter-list ) compound-statement


optional-static ::= static | ϵ
return-type ::= void | valid-data-type
parameter-list ::= parameter | parameter , parameter-list
parameter ::= valid-data-type identifier-list
identifier-list ::= identifier | identifier , identifier-list


method-call ::= identifier ( optional-identifier-list )
optional-identifier-list ::= identifier-list | ϵ
```

3.6.2 Static methods

So far, all of the programs you have written have the word `static` written at the beginning of each method declaration. In the next chapter we will learn how to create *objects* and you will find that in Java there is a body of code called the *static* code that is available at all times, and the rest of the code which can only be executed via these *runtime* created objects. Since we don't know how to make objects yet, all of our code is static. The `main()` method of any application is always `static` because it is the first thing to be executed, and so no objects exist at the time that `main()` is started.

3.6.3 The return statement

All methods must return a result unless they are of type `void`. The `return` statement has two purposes: it indicates an exit point from a method, and it specifies the value to return.

For `void` methods only, the return value will be empty, and as a special case there is an implicit `return` statement at the end of the method. For non-`void` methods you will always have to have at least one `return` statement.

New programmers sometimes forget to put in all the `return` statements. Here's a program that will not compile:

```
import java.util.Scanner;

class ReturnFlow
{
  static String testReturn(int n)
  {
    if (n > 0)
      return "positive";
    else if (n < 0)
      return "negative";
  }

  public static void main(String[] args)
  {
    Scanner keyboard = new Scanner(System.in);
    int n;

    System.out.print("Number? ");
    n = keyboard.nextInt();

    System.out.print("Number is " + testReturn(n));
  }
}
```

The idea is that we read in a number and then test to see if it is positive or

negative, returning an appropriate `String` from method `testReturn`.

When we try and compile it, we get the following:

```
> javac ReturnFlow.java
ReturnFlow.java:11: missing return statement
  }
  ^
1 error
```

The Java compiler analyses your methods to find every *control path*, each of which must have a `return statement`. Essentially, if there is a way of getting through a method's code without encountering a `return` statement, then something must be wrong.

In this case, we have failed to take into account the case where argument `n` is zero. If we change the body of method `testReturn()` to include the zero case, then all is well.

```
  if (n > 0)
    return "positive";
  else if (n < 0)
    return "negative";
  else return "zero";
```

3.6.4 Recursion

It turns out that method calls can be used to generate iterative behaviour. This not-at-all obvious but extremely elegant technique is called *recursion*. The idea is to make a method call *itself* until some condition is met.

```
class PowerRec
{
  static double powRec(double x, int power)
  {
    if (power == 1)
      return x;

    return x * powRec(x, power - 1);
  }

  public static void main(String args[])
  {
      double x = 2;

      System.out.println(x+" to the fourth is "+powRec(x, 4));
  }
}
```

Since we are trying to show off the capabilities of the language, here's an equivalent recursive implementation that uses the `?  :` conditional operator to reduce the recursive method to a one liner.

```
class PowerRec1
{
  static double powRec(double x, int power)
  { return power == 1 ? x : x * powRec(x, power - 1); }

  public static void main(String args[])
  {
    double x = 2;

    System.out.println(x+" to the fourth is "+powRec(x, 4));
  }
}
```

Now, apart from being elegant, recursion is also particularly well suited to problems which are composed of smaller versions of themselves. Actually, our `Power` method is an example of this: x^4 is $x \times x^3$, and so we can define the function in terms of 'smaller' calls to itself. Another classic example is the Fibonacci sequence

1, 1, 2, 3, 5, 8, 13, 21 ...

in which each element is the sum of the two previous. Mathematically, we say

$$F_n = \begin{cases} 1 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

With recursion, we can turn this straight into code:

```
class Fibonacci
{
  static int Fibonacci(int n)
  {
    if (n == 0)
      return 1;
    else if (n == 1)
      return 1;
    else
      return Fibonacci(n - 1) + Fibonacci(n - 2);
  }

  public static void main(String args[])
  {
    for (int n = 0; n < 10; n++)
      System.out.println(Fibonacci(n));
  }
}
```

With a switch statement, we can make the code look even more like the mathematical definition:

```
static int FibonacciSwitch(int n)
{
  switch (n)
  {
    case 0: return 1;
    case 1: return 1;
    default: return Fibonacci(n - 1) + Fibonacci(n - 2);
  }
}
```

One thing to note in this code is that there are no `break` statements at the end of the switch branches. Why is this? Well, branches that end in a `return` statement cause control to jump right out of the function, so a following `break` statement which causes control to jump to successor of the `switch` statement would be redundant. In fact, technically it would be *unreachable code* (since such a `break` statement could never be executed) and some compilers would issue an error message if the `break` were included.

3.7 Exception handling

An *exception* is a runtime event that is outside of the normal flow of program logic. We have already seen that attempting to execute a division by zero

causes an *arithmetic exception* which without special action causes program termination.

The basic principle of exception handling is that when an exception occurs, the JVM looks in the currently executing method for some code to catch the exception — an *exception handler*. If it can't find a handler, it terminates the method and looks for a handler in the method that called the method that raised the exception. This continues until either a handler is found, or the `main()` method is terminated at which point the default system exception handler prints out a list of all the methods that were terminated.

Let's revisit the divide by zero example from the last chapter. This time, we'll use a method to do the actual division.

```
import java.util.Scanner;

class DivideIntegers
{
  static int quotient(int numerator, int denominator)
  { return numerator/denominator; }

  public static void main(String[] args)
  {
    Scanner keyboard = new Scanner(System.in);
    int first, second;
    int quotient;

    System.out.print("First number? ");
    first = keyboard.nextInt();
    System.out.print("Second number? ");
    second = keyboard.nextInt();
    quotient = quotient(first, second);

    System.out.print("Quotient is " + quotient);
  }
}
```

When we run this program and enter zero for the second number, a divide by zero exception is 'thrown' by the method `quotient()` and not caught by either `quotient()` or `main()`, so we get the standard stack dump.

```
> java DivideIntegers
First number? 23
Second number? 0
Exception in thread "main" java.lang.ArithmeticException:/by zero
        at DivideIntegers.quotient(DivideIntegers.java:6)
        at DivideIntegers.main(DivideIntegers.java:18)

>
```

Now we shall modify the program so that (a) the divide by zero exception is caught in `main()`, and (b) we keep asking for numbers until the division succeeds without raising the exception, which we do by wrapping a `while(true)` statement around the data entry, and using a `break` statement to get out of the loop when we have successfully read a non-zero denominator.

In Java, we catch exceptions by (i) enclosing the code that might throw the exception in a `try` block and (ii) appending one or more `catch` blocks. The exceptions themselves are given names: the exception thrown by a divide by zero is called `ArithmeticException arithmeticException`. If the code in the `try` block executes normally, then the `catch` clause is ignored. If an exception is thrown within the `try` block, then the JVM looks for a `catch` clause labelled with the specific exception, and if it finds one it executes that

code, and then continues with the statement after the `try` and `catch` clauses.

```
import java.util.Scanner;

class DivideIntegersHandled
{
  static int quotient(int numerator, int denominator)
  { return numerator/denominator; }

  public static void main(String[] args)
  {
    Scanner keyboard = new Scanner(System.in);
    int first, second;
    int quotient;

    while (true)
    {
      System.out.print("First number? ");
      first = keyboard.nextInt();
      System.out.print("Second number? ");
      second = keyboard.nextInt();
      try
      {
        quotient = quotient(first, second);
        break;
      }
      catch (ArithmeticException arithmeticException)
      {
        System.err.println(arithmeticException +
           ": second number must be nonzero");
      }
    }

    System.out.print("Quotient is " + quotient);
  }
}
```

```
> java DivideIntegersHandled
First number? 23
Second number? 0
java.lang.ArithmeticException: / by zero:
second number must be nonzero
First number? 23
Second number? 2
Quotient is 11
>
```

4 Packaging

Write code once: use it many times

You have now written many programs, but most of these have been very short illustrations of a particular language feature. It is now time to start learning to write substantial *applications* — large programs with lots of different sub-parts, possibly controlled by a Graphical User Interface (GUI).

4.1 The tar-pit of complexity

One of the most famous books written on software engineering is Fred Brooks' *The Mythical Man Month* [**?**]. In it, he likened large scale development to the La Brea Tar Pits in Los Angeles; a famous source of fossils. For about the last 40,000 years, tar has seeped up to the surface there, making sticky pools in which animals become entrapped and ultimately fossilised. Working on big software artefacts can require us to understand myriad interconnections and interdependencies, and even good programmers can rapidly become overwhelmed by the sheer volume of information.

We need techniques that, when coupled with disciplined programming, allow us to compose large applications from 'human-sized' chunks of code. We don't want to end up paralysed and fossilised in the tar-pit of complexity. We keep ourselves safe by trying to make small parcels of code that operate rather independently, and which can be re-used in different applications.

4.2 Portability and re-use

We talked quite a lot in the first chapter about the drive to produce *portable* programs that could give the same results on different machines. General purpose programming languages developed between the 1950's and 1980's went a long way to achieving that: Java's extra contribution to portability is the Java Virtual Machine, a standardised pseudo-computer with well defined behaviour that we can efficiently interpret on different real computers.

So, portability is largely solved at the level of individual programs. We can write applications which run with similar look and feel across Linux, Windows PC's, Mac's and even on your mobile phones. There is another kind of portability though: the portability of pieces of code from one project to another. We would like programmers to be able to write rather generic code which can be picked up and re-used later as part of another application. For instance, writing

a program to fetch an audio file from a remote networked computer is actually a long and complicated process. It would be very discouraging, and expensive, if every time we needed to do this we had to write everything from scratch.

Now, very early on in the development of programming languages the need for a set of standardised facilities was recognised. Each language came with a *library* of useful functions. However, most of these libraries were specified at the time the language was designed, and it was very hard for them to keep up with changes in technology. For instance, hand held intelligent phones like the iPhone simply didn't exist when Java and other languages were designed. Any attempt to imagine what the future might bring is pretty much doomed to failure: there are always surprises. What we need is a way to gracefully extend our libraries as technology moves on.

Another aspect of library design was that usually the source code was not available. Library functions were just black boxes which you could not see inside. As long as the library function did *just* what you wanted, no more, no less, this was fine. However, in reality we often want to add extra capability to some core library function, or even change the way it does something. Think about a library function to print out a variable, for instance. It would be great if every time we made a new user-defined data type we could arrange for the system print function to be extended to allow variables of that type to be handled. This isn't possible using traditional languages like Pascal and C.

So, true re-use means that we need to be able to customise existing code in a way that doesn't disturb the way it works inside, but allows us to generalise it to handle new style of data. Re-use became a holy grail for software engineering — after all, electronic engineers make their systems out of large numbers of essentially standardised parts: why can't we make software that way too.

Practical re-use is provided by *object oriented programming,* a style of programming language with its roots in the simulation of real world systems. The idea was to package up both code and user-defined data types into bundles called *classes*. More importantly, we could define a new class which used as its starting point an old class, but in which we could change or extend the functionality. In Java, we say that we can *extend* a *parent* class to make a new *child* class. The parent class is called the *superclass* of the child class.

4.3 Types and classes

In Chapter 2 we met the basic Java data types. Each type allows a certain range of values, and there are built in operators to manipulate those values. We can add user-defined data types: the simplest is the `enum` type, but we can also compose primitives (and indeed user defined types) into aggregates such as a *coordinate pair* (representing a single point in a graphical display) which is an ordered pair of doubles or a *telephone book record* (comprising a name and telephone number).

Here's a first attempt at defining a Point data type in Java.

```
class Point{
    double x, y;
}

class Simple {
  public static void main(String[] args) {
    Point p = new Point();

    p.x = 17.3;
    p.y = 12.1;

    System.out.println("(" + p.x + ", " + p.y + ")");
  }
}
```

The new class `Simple` contains two *fields* which represent the x and y parts of a coordinate pair. We can access the fields using dot notation: `p.x` refers to the `x` field of `p`.

Now, although we have created a new data type which combines two doubles into a single `Point`, the way we declare a `Point` is different to the way we declare a `double`. Classes are descriptions of *objects* not simple data types. This fundamental difference between the built-in basic types and user defined types made with classes can be confusing.

When we write something like

```
int i = 3;
```

we are asking the computer to reserve space for a variable called `i` of type `int`, and to put the value `3` into it.

However, in class `Simple`, the line

```
Point p = new Point();
```

means the following.

1. Reserve memory space for a variable called `p` of type *reference to object instance of class* `Point`;

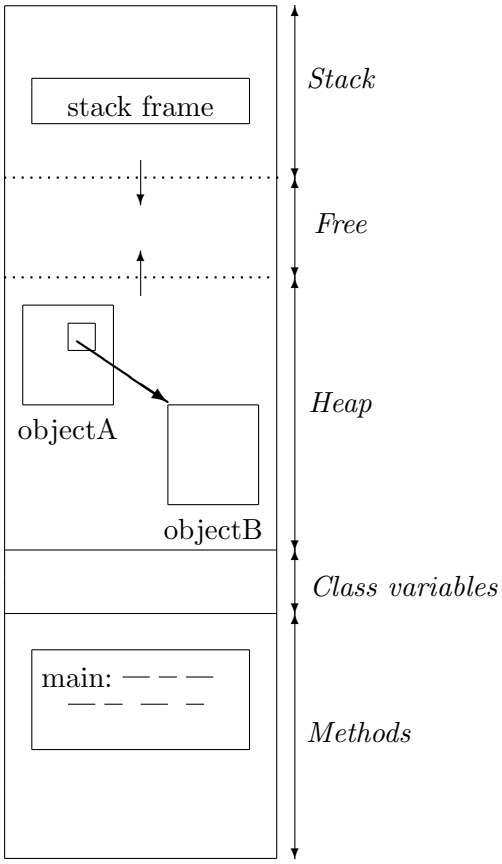
2. While the program is running, find a space big enough to hold an unnamed object of class `Point` and reserve it;

3. Put the address of that unnamed object into the variable `p`.

You might reasonably feel that this is a lot of complication, but there are good reasons for it, and as Computer Scientists it is important for you to really understand the detail.

4.4 Pigeonholes revisited: run time layout

To help you understand what is going on inside the computer during program execution, here's a map of memory usage for a typical Java system. We'll refer back to this picture as we think about various Java features.



The *Methods* section holds the JVM instructions that the compiler made from each of the methods that you write. This region has fixed size, and its contents do not change while a program is running. We've shown just one method `main` but in a real application there would be all the methods from all the classes together in this region.

The *Class variables* section holds data that is fixed during the lifetime of a program. The region is of fixed size, but the values in the variables might change as a program executes.

The *Heap* is the region in which the computer finds space for objects. The heap can grow and shrink in size during program execution as new objects are created and as old ones are cleared away after use. As you can see, objects themselves have variables in them which contain references to other objects: in this example object A contains a variable that references object B.

The *Stack* is another region of memory that grows and shrinks during program execution. It is used for allocating things which follow a strict *Last In — First Out* rule, by which mean that the order in which we destroy items is the reverse of the order in which we create them. It turns out that parameters

to methods and local variables follow this rule, but objects themselves do not, which is why objects are created on the heap.

The *Free* region sits between the heap and the stack. The heap grows up, and the stack grows down. If they ever meet, then the free region has vanished and the JVM has run out of memory.

4.5 Encapsulation

Successful re-use requires good hygiene. The `Point` class used in our `Simple` program is actually an example of *very* bad hygiene. Don't write your classes like this!

Good hygiene mostly means controlling access to an object's data fields very carefully. Allowing other classes to simply change the data fields, as we have done here, is a recipe for disaster.

Here's a much better (although still imperfect) version.

```
class Point{
  private double x, y;

  public void init(double iX, double iY) { x= iX; y= iY; }

  public double getX() { return x; }
  public double getY() { return y; }
}

class Coordinates {
    public static void main(String[] args) {
    Point p = new Point();

    p.init(13.2, 17.3);
  }
}
```

The most important change is that we have added the word `private` to the declaration of fields `x` and `y`. Once we do this, we are no longer allowed to refer to the fields as `p.x` and `p.y`. How, then, should we access the fields. Well, we use small methods called *getters* and *setters*. A *setter* takes values as parameters and puts them into the fields: they set values. *getters* get values from fields and return them to the caller. We make use of these methods by calling them using dot notation:

```
p.init(13.2, 17.3)
```

which means *call the* `init` *method of object* `p` *passing parameter values* `13.2` *and* `17.3`.

Why do we go to all the trouble of using private fields, getters and setters? Well it enables us to change things later inside class `Point` without having

to change every class that uses `Point`. As a trivial example, imagine that for some strange reason we needed to change the names of the `x` and `y` fields to `xCoordinate` and `yCoordinate`. We could do that here, because actually classes that use `Point` do not even need to know what those fields are called.

This principle of each class having a few public methods which control and supervise access to the class's data is absolutely fundamental to good software engineering. It enables us to write largely independent pieces of code that can be fitted together with only very small interdependencies. The alternative is to have the entrails of every data type open to view, and to need to go and rework large parts of an application every time we make major changes to a data type.

4.6 Constructors

One of the greatest sources of programming error is forgetting to initialise a variable. In the version of `Point` used in the `Coordinates` class above we have to declare the `Point P` and then separately initialise it with a call to method `p.init()`. If we forget to call `p.init()` then trouble may ensue.

A *constructor* is a special method that is automatically called when a new object is made. The name of the constructor method is the same as the name of the class. Parameter values that match the constructor's parameter list must be added to the declaration. Here's a simple example.

```
class Point
{
  private double x, y;

  Point(double iX, double iY)
  { x= iX; y= iY;}

  public double getX() { return x; }
  public double getY() { return y; }
}

class CoordinatesWithConstructor
{
  public static void main(String[] args)
  {
    Point p = new Point(13.0, 17.0);
  }
}
```

Here, we've removed the `init()` method and replaced it with a constructor method called `Point()` which has the same parameter list and sets the private data fields in the same way. Now, we can write

```
Point p = new Point(13.0, 17.0);
```

which means the following.

1. Reserve memory space for a variable called `p` of type *reference to object instance of class* `Point`;
2. While the program is running, find a space big enough to hold an unnamed object of class `Point` and reserve it;
3. Put the address of that unnamed object into the variable `p`.
4. Look in class `Point` for a method called `Point` whose parameter list matches in type and number the parameter list in the declaration; if found, call it.

4.7 Multiple constructors and overloading

There can be several constructors for each class: Java decides which one to use by matching the type and number of parameters. Let's extend our `Point` type to include a field for a `String` label which might be printed out when we plot a `Point` on the screen. This is a new feature of our class, but we don't want to have to disturb old applications which make use of the previous version, so we keep the old constructor (merely adding a line to put an empty string in the label) and add a new constructor which allows all three fields to be set.

```
class Point
{
  private double x, y;
  private String label;

  Point(double iX, double iY, String iLabel)
  { x= iX; y= iY; label = iLabel; }

  Point(double iX, double iY)
  { x= iX; y= iY; label = "";}

  public double getX() { return x; }
  public double getY() { return y; }
  public String getLabel() { return label; }
}

class CoordinatesWithConstructor
  {
    public static void main(String[] args)
    {
      Point p = new Point(13.0, 17.0, "Test point");
    }
}
```

This way, we maintain *backwards compatibility* for users of the old version of `Point`.

4.8 Overriding

We've just seen how we can add functionality to a class without disturbing existing code that uses that class. We've also seen how important it is to use functions to access data members: if we had allowed direct access to `Point`'s data fields then we would have had to change usage of a `Point` to correctly set the new `label` field. Instead, all we had to do was to add one line to the constructor.

Now we need to think about how to *change* the behaviour of methods, again without disturbing existing code that uses the old version. We're going to achieve this by making new classes that *extend* `Point` without actually changing `Point` itself. These child classes start off by effectively having a copy of all `Point`'s fields and methods which we can then *overload* with new methods.

Before showing you the full mechanism, we want you see a very powerful and useful trick. At the moment, if we try to print out the contents of a variable of class `Point` the results are disappointing.

```
class Point
{
  private double x, y;
  private String label;

  Point(double iX, double iY, String iLabel)
  { x= iX; y= iY; label = iLabel; }

  Point(double iX, double iY)
  { x= iX; y= iY; label = "";}

  public double getX() { return x; }
  public double getY() { return y; }
  public String getLabel() { return label; }

}

class CoordinatesDefaultPrint
{
  public static void main(String[] args)
  {
    Point p = new Point(13.0, 17.0, "Test point");

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

When we run this piece of code, we get something like this:

```
:-) java CoordinatesDefaultPrint
Point@19821f
```

The system is telling us that it has been asked to print out a variable of type *reference to object of class* `Point`, and that the object lives on the heap at memory address 0x19821F. This isn't very useful! What we would like is for objects of class `Point` to know how to print themselves out.

Now, in detail it turns out that when we call `System.out.println()` It expects to see a single string as its parameter. If it finds something else (such a `p`) it attempts to call a method called `toString()` to convert the object to a string. There is a default `toString()` method that simply prints out the class and the object's address, and that's what gives the disappointing message.

If we add a `toString()` method to `Point`, then we can get the results we want.

```
class Point{
  private double x, y;
  private String label;

  Point(double iX, double iY, String iLabel)
  { x= iX; y= iY; label = iLabel; }

  Point(double iX, double iY)
  { x= iX; y= iY; label = "";}

  public double getX() { return x; }
  public double getY() { return y; }
  public String getLabel() { return label; }

  public String toString()
    { return "(" + x + ", " + y + ") '" + label + "'"; }
}

class CoordinatesCustomPrint
{
  public static void main(String[] args)
  {
    Point p = new Point(13.0, 17.0, "Test point");

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

The output is

```
:-) java CoordinatesCustomPrint
(13.0, 17.0) 'Test point'
```

4.9 Classes that use other classes

The whole point of a class is to create a bundle of data and methods that can be used by other applications. We *try* to keep the number of public methods small, and in general we do not allow any of the data to be public. That way we maintain good hygiene.

Here's a version of our `Point` class that has a fuller range of getters and setters. We can set just the coordinate, or the coordinate and the label. We are going to use this class to develop a new class that describes polygons.

```
class Point{
  private double x, y;
  private String label;

  Point(double iX, double iY)
  { x= iX; y= iY; label = "";}

  Point(double iX, double iY, String iLabel)
  { x= iX; y= iY; label = iLabel; }

  public void set(double iX, double iY) { x = iX; y = iY; }
  public void set(double iX, double iY, String iLabel)
  { x = iX; y = iY; label = iLabel;}
  public double getX() { return x; }
  public double getY() { return y; }
  public String getLabel() { return label; }

  public String toString()
    { return "(" + x + ", " + y + ") '" + label + "'"; }
}
```

In computer graphics, a polygon is a sequence of points. When we *render* a polygon (that is, paint it onto a display screen) we go through the sequence of points joining pairs. We distinguish between *complex polygons* in which the lines may cross, and *simple polygons* in which the lines may not cross. In mathematics, polygons are almost always simple: we would view a complex polygon as a collection of individual simple polygons.

We are going to write a class that represents complex polygons, and then derive another class from it that rearranges the ordering of the points to guarantee

that the polygon is simple.

```
class ComplexPolygon{
  private Point[] points;
  private int maxPoints;
  private int nextFreePoint = 0;

  ComplexPolygon(int vertexCount)
  {
    points = new Point[vertexCount]; maxPoints = vertexCount;

    for (int i = 0; i < maxPoints; i++)
      points[i] = new Point(0,0);
  }

  void addPoint(double x, double y)
  { points[nextFreePoint++].set(x, y); }

  void addPoint(double x, double y, String label)
  { points[nextFreePoint++].set(x, y, label); }

  Point getPoint(int index)
  { return points[index]; }

  int getMaxPoints() { return maxPoints; }
}
```

We have two independent classes now: `Point` which represents a single point in the graphics plane, and `ComplexPolygon` which represents a (possibly self-intersecting) polygon as a sequence of `Point` objects. `ComplexPolygon` *uses* `Point`.

In detail, class `ComplexPolygon` has three data fields: an array of `Point` objects whose size is unspecified; an `int` called `MaxPoints` specifying the maximum number of points that we can hold in a particular polygon; and another `int` called `nextFreePoint` which keeps track of how many points the current polygon actually has (as opposed to `MaxPoints` which specifies an upper bound).

The idea is that when we create a new object of class `ComplexPolygon` we specify the upper bound on the number of vertices as a parameter to the constructor `ComplexPolygon(int vertexCount)` The constructor method then creates an appropriately sized array. We shall study arrays in more detail in Chapter 5. Here we note that simply creating an array of class `Point[]` does not actually create the individual `Point` objects themselves, so our constructor includes a `for` loop that runs down the array constructing them explicitly.

Our two `addpoint()` methods are a bit like setter methods, except that they put the information into the next free `Point` object in the array, updating `nextFreePoint` accordingly.

4.9.1 Things we could do better

We've tried to make this class easy to understand. When you've learnt a little more, you might come back to this section and wonder whether we could have made a better version. Well the answer is yes: firstly it turns out that since arrays carry their length around with them it isn't necessary for us to use the `maxPoints` variable; and in addition it would make sense to only make a `Point` object when we actually load some data into the array. That way, small polygons with less than `maxPoints` vertices would take less space. Finally, we could use a dynamic structure like a *linked list* to hold the points, thus doing away with the need for a fixed upper bound. We'll revisit these issues in later chapters but for now note that we should really add code that checks to ensure that users of our class are not trying to add more points than we can cope with.

4.10 Classes that extend other classes

So far, we've used classes to encapsulate data and to bundle data and relevant methods together into 'black box' chunks of code. These techniques are very important to the maintenance of good data hygiene.

The real power of this style of programming, though, comes from our ability to *extend* existing classes. We start with a base class. In this example our base class will be *ComplexPolygon*. We then make an extended class *SimplePolygon* that *inherits* all of the data fields, constructors and methods from the base class. We say that `ComplexPolygon` is the *superclass* of `SimplePolygon`.

The basic syntax for extending classes is straightforward.

```
class SimplePolygon extends ComplexPolygon {
}
```

This empty declaration creates a new class `SimplePolygon` which is essentially just a synonym for `ComplexPolygon`. It has all the same methods, constructors and data fields.

We can now add extra data fields, and we can also add new methods and constructors either with brand new names or by overloading existing names: that is, adding methods with existing names but different parameter to the methods in the base class. Most interestingly of all, we can *override* methods in the base class by declaring new methods in the extended class that have the *same* name and parameter list as a method in the base class.

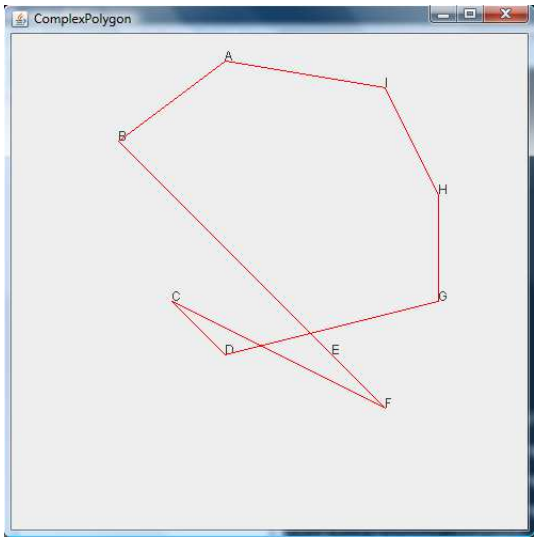
We can use these techniques to create classes that are specialised versions of other classes. In this example, we are going to make a class that represents *simple* polygons, based on our existing ComplexPolygon class. In a `ComplexPolygon`, points are stored in the order in which they are entered. In a `SimplePolygon`, the points are sorted counter clockwise, so that we can never have two lines crossing each other.

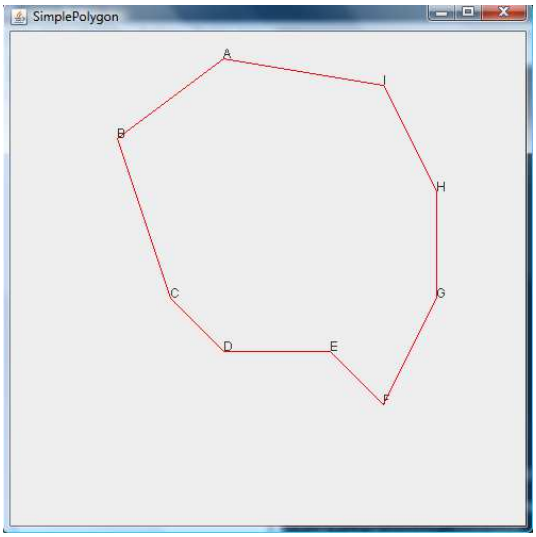
We use the classes as shown below. We have two declarations for object `shape` — one as a `ComplexPolygon` and the other as a `SimplePolygon`. If we 'comment out' the `ComplexPolygon` declaration and use the `SimplePolygon`

declaration then the points are sorted into reverse clock order after each call to `addPoint`. If we use `ComplexPolygon` instead, then the order in which the `addPoint()` calls are made dictates the order of the points around the polygon.

```
SimplePolygon shape = new SimplePolygon(20);
//ComplexPolygon shape = new ComplexPolygon(20);

shape.addPoint(2, 2, "B");
shape.addPoint(6, 6, "E");
shape.addPoint(7, 7, "F");
shape.addPoint(3, 5, "C");
shape.addPoint(4, 6, "D");
shape.addPoint(8, 5, "G");
shape.addPoint(8, 3, "H");
shape.addPoint(7, 1, "I");
shape.addPoint(4, 0.5, "A");
```

We've labelled the nine points so that they read in counter clockwise order. Here you can see the results of using the two different classes.





We see that `ComplexPolygon` retains the B-E-F-C-... construction order. `SimplePolygon` has sorted vertices.

4.11 Using `super` to enrich base methods and constructors

It's quite usual for a method in an extended class to need to call the method in the superclass that it is replacing. For instance, we might need to change a method so that it stores some information in data fields belonging to the extended class, but then carry on to do whatever it was the original method did.

Now, we can't simply call the method by name, because that would be indistinguishable from a recursive call. Instead, Java provides a pseudo-variable

called `super`. If you write `super.aMethod()` then `aMethod()` in the *superclass* of the current class is activated. Typically we use this to enrich a method from the superclass with extra actions.

Here are the definitions of the `SimplePolygon` constructor and the two `addPoint()` methods from our extended class.

```
class SimplePolygon extends ComplexPolygon {
  SimplePolygon(int vertexCount) {super(vertexCount); }

  void addPoint(double x, double y)
    { super.addPoint(x, y); makeSimple(); }

  void addPoint(double x, double y, String label)
    { super.addPoint(x, y, label); makeSimple(); }

}
```

The constructor is particularly simple: it simply calls the superclass's constructor with the same parameter. Really, we are just chaining the constructors together. Java will automatically perform this chaining for simple (parameterless) constructors. Actually, we think it is good programming practice to put in explicit calls to the superconstructor in any case, but others differ.

The `addPoint()` methods also simple chain up to their superclass equivalents. However, on return from the superclass method they each make a call to `makeSimple()` which sorts the points in the array into counter-clockwise order.

4.11.1 `this` and `super`

The pseudo-variable `super` gives us access to the methods in our super class. (Only one level is allowed, by the way! You are not allowed to write `super.super.field` to get to the class two levels up). Occasionally we also need to be able to explicitly specify a field in our own object, and we do that with the pseudo-variable `this`.

The simplest example is when we define a setter method which uses the name of the field to be updated as a parameter name. This is quite a common idiom, but the obvious implementation won't work:

```
class Point{
  private double x, y;

  public void setBroken(double x, double y) { x = x; y = y; }
}
```

Here we have a fragment of our `Point` class with a setter called `setBroken` which has parameters that have the same name as the class fields that we are trying to set. We try using the expression `x = x`, but all that's going to do is update the local value of parameter `x`. The solution is to use the `this` pseudo-variable

to access the current object:

```
class Point{
  private double x, y;

  public void set(double x, double y) { this.x = x; this.y = y; }
}
```

A more interesting use of `this` arises when we are implementing dynamic data structures, which we will look at in the next chapter. Briefly, a dynamic data structure has a base object which points to other objects, and sometimes we need to be able to step along one of these chains. We use a variable that *points* to each element of the chain in turn and which is initialised to `this`, the current object.

4.11.2 Sorting the points

The main work in `SimplePolygon()` is done by the method `makeSimple()`. It works as follows.

1. Find a point x with minimal y-coordinate.

2. Move that point to the beginning of the array by swapping.

3. Make an array of doubles that can hold the angle of the line joining x to each of the other vertices in the polygon.

4. Sort the angle array and the points array together so that the points are put into counter clockwise order.

In this example, we are going to use a bubble sort, but please remember that bubble sort is an $O(n^2)$ algorithm and that more efficient $O(nlog_2(n))$ alternatives are available.

Finding a minimal point Here is a method `findMin()` which scans the array of points, remembering the index of the first point with the lowest y-coordinate seen so far. As a small efficiency saving, we directly remember the corresponding

minimal point, thus saving an indexing on each comparison.

```
private int findMin()
{
  Point minPoint = points[0];
  int minPointIndex = 0;

  for (int i = 1; i < nextFreePoint; i++)
    if (points[i].getY() < minPoint.getY())
    {
      minPoint = points[i];
      minPointIndex = i;
    }

  return minPointIndex;
}
```

There might be many minimal points, of course: we just remember the first we encounter. If we changed the test from

```
if (points[i].getY() < minPoint.getY())
```

to

```
if (points[i].getY() <= minPoint.getY())
```

then we would remember the *last* minimal point seen. The class would still operate correctly, but it would be slightly less efficient because we would update `minPoint` every time we saw a minimal point, instead of just once. This is a small saving, but good programmers keep an eye open for these opportunities.

Computing the vertex line angle This small method returns the angle (in radians) between the x-axis and the line between two points. We have to take a little care for the case when the line is vertical. We're using an arctan function to find the angle using the formula

$$\theta = \arctan \Delta x / \Delta y$$

where Δx is the difference in the x-coordinates and Δy the difference in the y-coordinates.

```
private double angle(Point p1, Point p2) {
  if (p2.getX() == p1.getX())
    return (double) (3.1412 / 2);
  else return
    Math.atan2(p2.getY()-p1.getY(), p2.getX()-p1.getX());
}
```

Making a complex polygon simple Here's the `makeSimple()` method itself. First, we check to make sure that we have more than two points in the polygon. If not, there's nothing to do! Then we find a minimal point, swap it to the beginning of the array; construct the angles array and populate and then finally bubble sort the `angles` and `points` arrays into order.

```
public void makeSimple() {
  if (nextFreePoint < 2)
    return;

  int minPointIndex = findMin();

  // Swap minimum point to start of array
  Point swapPoint = points[0];
  points[0] = points[minPointIndex];
  points[minPointIndex] = swapPoint;

  double[] angles = new double[nextFreePoint];

  for (int i = 1; i < nextFreePoint; i++)
    angles[i] = angle(points[0], points[i]);

  for (int pass = 1; pass < nextFreePoint; pass++)
    for (int test = 1; test < nextFreePoint - pass; test++)
      if (angles[test] > angles[test+1])
      {
        double swap = angles[test];
        swapPoint = points[test];
        angles[test] = angles[test + 1];
        points[test] = points[test + 1];
        angles[test + 1] = swap;
        points[test + 1] = swapPoint;
      }
}
```

5 Structures

A data structure is a container for data elements, possibly including other data structures

In Chapter 2 we learnt about Java's primitive datatypes. Real programs need more than a sprinkling of simple integers and strings to do their work: usually we are collecting large amounts of data and we need to directly represent interrelationships between elements of those collections.

A *data structure* is an arrangement of data items inside the computer's pigeon-hole memory that enables us to store and retrieve information efficiently. Really, a data structure is a more sophisticated and abstract way of thinking about memory.

We sometimes use data structures in the real world too: a telephone directory is a list of ordered triples (*name*, *address*, *number*) which has been sorted into name order. This enables us to rapidly look up the telephone number belonging to somebody whose name we have. If, on the other hand, we were given an address and asked to find the number, we would have no alternative but to search linearly through the whole directory until we either found the address, or reach the end of the directory in which case the result is **not in directory**.

5.1 The impact of data structures on performance

Choosing data structures wisely can significantly impact performance. Let's think in more detail about our telephone directory example.

If we don't know anything about the order of the records, we have to look through them all in order. If we were to double the number of records, we would need twice as much time to search them. We say that *linear search* executes in time $O(n)$ which we read as *Order-n* or sometimes *Big-Oh-n*.

Since the names are sorted, we can use a simple algorithm called *binary search* to find any name in $\lceil \log_2(n) \rceil$ time where n is the number of entries. We read this as *ceiling of Order Log n*. $\lceil x \rceil$ (ceiling of x) means the smallest integer greater than or equal to x.

Here is an *informal* description of the algorithm. Quite often we are given slightly vague specifications that we have to turn into code: this sort of thing is typical.

At each stage, we check a *region* of the directory for a *target* name. The region is initially the entire directory.

if the region is empty then return **not in directory**.

find the middle name in the region;

if the middle name is the target, then return the associated telephone number;

if the middle name is less than the target, then set the region to be the top half of this region and try again; else

set the region to be the bottom half of this region and try again.

There are two senses in which this specification is vague: firstly it doesn't tell us how to represent a region or even a name. Secondly, some details are missing — how does termination work for instance? With a bit of thought, we can see that when the region is only a single name long, then either the top 'half' or the bottom 'half' will be empty... so we can sort-of see what the specification means. In a real program, though, we are going to need to be a lot more specific.

Now, the key feature of this algorithm is that it repeatedly subdivides the region in half, and so the number of times we need to 'try again' is the number of times it takes to reduce the size of the region to one by dividing it in half. For 1,000 records, that only need ten passes. For 2,000, we only need eleven. That is, doubling the size of the directory only adds one extra pass! If we can find an $O(\log_2(n))$ algorithm to solve our problem then we are usually pretty pleased.

5.2 The three data structuring mechanisms

In Java, we have three separate mechanisms by which we can build data structures

objects whose instance variables act like a one dimensional table of (potentially different type) variables, with each field named;

arrays of variables which act like one dimensional tables of variables all of which have the same type, and which can be accessed via integer *index expressions*; and

linked structures of objects that contain references to other objects. The structure comprises a series of elements, each of which have one or more links to other elements.

5.2.1 Flexibility

These mechanisms offer varying levels of flexibility, but more flexibility incurs greater run-time performance costs.

objects The fields of an object must be specified in its class, and so must be known at the time the compiler is run (which we call *compile-time*). Only the values held in an object's fields can change whilst the program is running (or at *run-time*).

arrays The *type* of an array must be known at compile time, but its *extent* (the number of elements it contains) only has to be specified once the program is running. Once created, though, an array cannot change its extent. Only the values in the array's elements may change.

linked structures The extent of a linked data structure is flexible. Only the number of links from each element is fixed, and with skilful design we can even overcome that restriction.

5.2.2 The performance penalties of data structuring

Flexibility comes at a price. The class, array and linked structures are increasingly flexible, but also increasingly expensive in terms of run-time computation.

objects The compiler knows exactly where each field in an object is relative to the object's base address, so we can calculate at compile-time the offset of each variable from that base.

arrays With an array, things may be more complicated. If the index expression is a constant, then it may be evaluated by the compiler and a fixed offset generated, so in this case accessing an array element is as fast as accessing an object field. Usually, however, the index expression has some terms which are only known at run-time, so the running program has to compute the position of the variable, which takes a small constant-time overhead.

linked structures With linked structures, in general we have to walk down a chain of links so the amount of time taken by the running program to access elements is variable. This can have unpleasant consequences if we do not take care.

5.3 Arrays in Java

In Java, an array is a one-dimensional collection of data items, all of the same type or class. Each element of the array is given a sequential number called its *index* starting at zero. Arrays are of fixed size: you specify the number of elements you want when the array is created and after that the array cannot expand or contract.

Java implicitly creates a special array class for every class you make, as well as for the primitive types. The name of this special class is the name of the base class with square brackets appended: `MyClass[]`. When you declare an array, you are really declaring an object of this implicit array class.

5.3.1 Declaring arrays

Like all objects, we have to distinguish the *declaration* of a *reference to object* variable and the creation of the actual object itself on the heap.

Array declarations can take one of two forms:

```
int [] intArrayA, intArrayB, intArrayC;
```

and

```
int intArrayA[], intArrayB[], intArrayC[];
```

are equivalent.

As a point of style, the first version, in which the array designator `[]` is part of the class is the preferred Java idiom: it reinforces the reality that `intArrayA` is of class *array of* `int`. The later version is allowed for historical reasons: the C language on which much of Java's syntax is based works this way.

Be very careful about mixing idioms! What does this mean?

```
int[] intArrayX, intArrayY[];
```

This is a legal piece of Java that declares `intArrayX` to be of class *array of* `int` and `intArrayY` to be of class *array* of array *of* `int`!

5.3.2 Creating arrays

Once we have declared an array object reference, we can create an actual array for it to reference. There are two ways to make an array: with the `new` operator as is usual for objects and with the `{ ... }` curly brackets initialiser.

```
// Create an int array with indices 0..19; values set to zero
intArrayA = new int[20];

// Create an Int array withy indices 0..7;
//values set to the first eight prime numbers
intArrayB = {2, 3, 5, 7, 11, 13, 17, 19}
```

After initialisation, `int ArrayB` looks like this: eight cells indexed 0...7 containing the first eight prime numbers.

0	1	2	3	4	5	6	7
2	3	5	7	11	13	17	19

It is only at the point of creation that the size of the array must be specified, either explicitly with `new` or implicitly with the initialiser. Java then allocates enough memory for the array: since the adjacent pieces of memory will in general be used for other things, the size of an array is fixed for the duration of the program. If you need to extend an array, then you must create a new array object of the required size and copy values from the old version into it, as we shall show you in Section 5.3.8. As long as you do not continue to use the old array, it will be cleaned up and its memory reclaimed by the *garbage collector*.

The length of an array is made available *via* a `public`, `final` field called `length`, so `intArrayA.length` would contain 20 and `intArrayB.length` 8.

5.3.3 Iterating over arrays with `for` statements

Often, we want to process the elements of an array in some way: print them all out, or add them up, for instance. We can do this by using a `for` loop to step over all the elements. For instance, we can find the sum of the first eight primes by adding up the elements of `intArrayB`.

```
int sum = 0;

for (int i = 0; i < intArrayB.length; i++)
  sum += intArrayB[i];
```

Occasionally, we want to scan an array in reverse order. This needs a little care because we need to remember that the highest available index is one less than the length. Here's how to print the first eight primes in reverse order.

```
for (int i = intArrayB.length - 1; i >= 0; i--)
  System.out.println(intArrayB[i]);
```

5.3.4 The for-each statement

In both of the above `for` loops, the indexing integer (`i`) is called the *induction* variable. There is another form of the `for` loop called the *for-each* loop that does away with the need for an induction variable. The general form is

`for (` *type var* `:` *array*`)` *statement* `;`

The *array* must be of the same base type as the variable `var`. The loop works as follows: the body of the loop is entered exactly once for each element of the array, with *var* set to the value of the element. The `for` loop does not provide the index of the element is, although the elements are delivered in ascending index order. As we'll see later on, the for-each loop can also be applied to system-supplied datastructures called *collections* which may have no explicit ordering.

This, then, is an alternative way to print out the first eight primes in ascending order.

```
for (int value: intArrayB)
  System.out.println(value);
```

5.3.5 Reading in arrays from the keyboard

Let's now put everything together into a program that (i) asks the user for an array's extent; (ii) creates an appropriately sized `int` array, and then (iii) loads its contents from the keyboard before printing the array using all three styles of `for` loop.

We begin by writing two utility routines which prompt the user, and then read integers from the keyboard. Method `getInteger()` prints a user-defined prompt and then reads a single integer from the supplied `Scanner` object. Method `getIntegers()` reads an entire array of integers, using the array's `length` field to specify the number of integers to read.

```
import java.util.Scanner;

class LoadIntegerArray
{
  static int getInteger(Scanner sc, String prompt)
  {
      System.out.print(prompt);
      return sc.nextInt();
  }

  static void getIntegers(Scanner sc, int [] arr)
  {
    for (int i = 0; i < arr.length; i++)
      arr[i] = getInteger(sc, "Element " + i + ": ");
  }
```

In `main()`, we read an `int` extent and then load `testArray` with integers before

printing them out.

```
  public static void main(String args[])
  {
    Scanner keyboard = new Scanner(System.in);

    int extent = getInteger(keyboard, "Size of array? ");

    int[] testArray = new int[extent];

    getIntegers(keyboard, testArray);

    System.out.println("\nForwards: ");
    for (int i = 0; i < testArray.length; i++)
      System.out.println("testArray[" +i+ "]: " + testArray[i]);

    System.out.println("\nBackwards: ");
    for (int i = testArray.length - 1; i >= 0; i--)
      System.out.println("testArray[" +i+ "]: " + testArray[i]);

    System.out.println("\nFor-each: ");
    for (int value: testArray)
      System.out.println(value);
  }
}
```

Here's the result of on erun of this program.

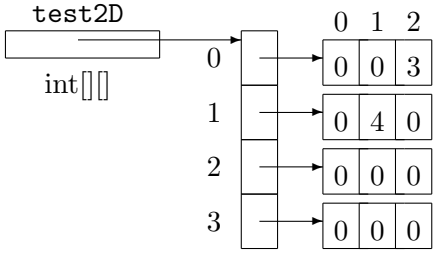
```
:-) java LoadIntegerArray
Size of array? 5
Element 0: 2
Element 1: 3
Element 2: 5
Element 3: 6
Element 4: 7

Forwards:
testArray[0]: 2
testArray[1]: 3
testArray[2]: 5
testArray[3]: 6
testArray[4]: 7

Backwards:
testArray[4]: 7
testArray[3]: 6
testArray[2]: 5
testArray[1]: 3
testArray[0]: 2

For-each:
2
3
5
6
7

:-)
```

5.3.6 Arrays of arrays

All arrays in Java are one dimensional, but since we can have arrays of objects, we can have arrays of arrays.

```
    int[][] test2D = new int[4][3];

    test2D[0][2] = 3;
    test2D[1][1] = 4;;
```

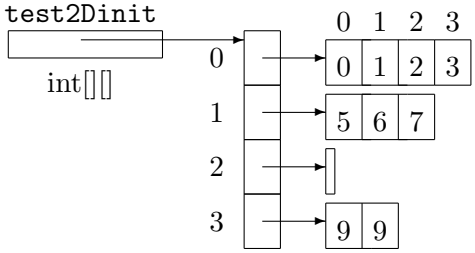
This code actually creates *five* separate arrays: four *row* arrays of length 3 whose addresses are then referenced by a *column* array of length 4.



There's no particular reason why a two-dimensional array should be rectangular. We can make each row a different length, especially if we want to save memory. Since every array carries its size around with it, we can iterate over the array using the usual nested `for` loop idiom.

```
int[][] test2Dinit = new int[4][];

test2Dinit[0] = new int[] {0, 1, 2, 3};
test2Dinit[1] = new int[] {5, 6, 7};
test2Dinit[2] = new int[0];
test2Dinit[3] = new int[] {9, 9};

for (int i = 0; i < test2Dinit.length; i++)
  for (int j = 0; j < test2Dinit[i].length; j++)
    System.out.println("[" + i + "][" + j + "]: " +
                       test2Dinit[i][j]);
```

Here, instead of using `new[][]` to create both the column and row arrays simultaneously, we initially just create the column array. Then we 'manually' create four different rows, one of which is zero length.



The nested `for` loops then print the contents. Here's the output from this fragment

```
[0][0]: 0
[0][1]: 1
[0][2]: 2
[0][3]: 3
[1][0]: 5
[1][1]: 6
[1][2]: 7
[3][0]: 9
[3][1]: 9
```

5.3.7 Anonymous arrays in parameter lists

Sometimes we would like to be able to pass a *variable* number of arguments to a method. We can use arrays in conjunction with array initialisers to achieve this effect.

```
class MultiArgs
{
  static void printMulti(String tag, int[] numbers)
  {
    System.out.print("\n" + tag + ": ");
    for (int val: numbers)
      System.out.print(val + " ");
  }

  public static void main(String args[])
  {
    printMulti("First set", new int[] {3, 4, 5});
    printMulti("Second set", new int[] {9, 8, 7});
  }
}
```

Here, we are creating an anonymous array 'on-the-fly' and passing it to the `printMulti` method. Here's the output of the program.

```
:-) java MultiArgs

First set: 3 4 5
Second set: 9 8 7
:-)
```

5.3.8 Resizing arrays manually

The method `System.arraycopy()` may be used to copy one array to another. We do not necessarily know what the internal details are of the methods and classes that we use. The convention is that we list the name of the method and its parameters, with some explanation as to what each does. The body of the method remains unspecified.

```
public static void arraycopy(Object src, int srcPos, Object dest, int
 destPos, int length)
```

This method will copy `length` entries from array `src` starting at position `srcPos` to array `dest` starting at `destPos`. The method is smart enough to notice if `src` and `dst` are the same object and ensures that the effect of the copy is as if the `src` elements were copied to a temporary array before copying back into the original array.

In the following example, we declare and create an array `arr` of `int` containing three elements, then make a new array `doubleArr` which is twice the length of `arr`. Finally, we copy the contents of `arr` into the first half of `doubleArr`.

```
int[] arr = {1,2,3};
int [] doubleArr= new int[2 * arr.length];

System.arraycopy(arr, 0, doubleArr, 0, arr.length);
```

5.3.9 Passing in parameters from the command line

Our `main()` methods are usually declared as

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

When we run the JVM from the command line, the array of `String` called `args` is filled in with any information that comes after the `java` *`myclass`* command itself. The command line is broken up into individual strings corresponding to each space-delimited portion. We call each substring a *command line parameter*.

The following program prints out the command line parameters.

```
class CommandArgs
{
  public static void main(String args[])
  {
    for (int i = 0; i < args.length; i++)
      System.out.println("args[" + i + "]:" + args[i]);
  }
}
```

Here's the screen output from one run of the program.

```
:-) java CommandArgs a b c
args[0]:a
args[1]:b
args[2]:c
```

5.3.10 Circular buffers

5.4 Dynamic structures

Arrays of arrays are simple examples of *linked* structures. In a two dimensional array, each element of the column array simply holds a reference (a link) to a row array. The real data, the stuff we actually want to manipulate, is held in the row arrays.

Now, even arrays of arrays are of fixed size once we have created them. However, many real programs have to manage datasets that are growing and

reducing during a program's execution. These *dynamically sized* datasets need dynamically sized data structures to hold them.

Dynamic data structures are constructed from elements (called *nodes*) that have one or more links, and a separate *payload* which contains the data itself. Each link can connect to another node of the same class. Here is a simple example.

```
class IntegerDynamicNode
{
  int payload;
  IntegerDynamicNode next;
}
```

This is quite a curious looking class. Each object contains one integer (the payload) and then a variable which references another object of the same class. because of this self-referential quality, data structures made from these dynamic nodes are sometimes called *recursive* data structures.

5.4.1 The value `null`

At first glance, it looks as though structures of class `IntegerDynamicNode` might be infinitely large: each node points to another node, which points to another node... The solution to this conundrum is to have a special value which means 'I am not pointing to anything'. This value is called `null` and it is a member of the set of values for every reference class. When a variable that references an object is first created, it is initialised to `null`.

5.4.2 Chains of nodes — the linked list

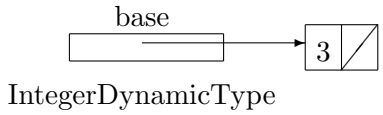
The simplest dynamic data structure is a one-dimensional chain of nodes, each of which has one link to the next node. We call this a *linked list*. It is the dynamic analogue of an array: one dimensional, but because it is dynamic the number of nodes can expand and shrink during the execution of a program. The downside is that the time taken to retrieve an arbitrary node is proportional to the length of the list.

It is a good idea to get into the habit of drawing pictures of dynamic data structures whilst you're learning how to use them. We represent each node as a box that has one box inside it for each field. Payload fields are labelled with their contents, and link fields either have a bar across them (representing `null`) or have an arrow leaving them for the next node. This code fragment creates a

single node that has `int 3` as its payload and links to nothing.

```
class IntegerDynamicNode
{
  int payload;
  IntegerDynamicNode next;
}

class SimpleNodes
{
  public static void main(String args[])
  {
    IntegerDynamicNode base;

    base = new IntegerDynamicNode();
    base.payload = 3;
    base.next = null;
  }
}
```

We draw this as



IntegerDynamicType

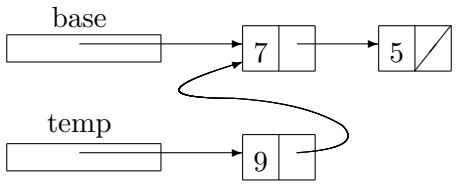
Now let's make a list of nodes. We are going to use a technique called *head insertion* in which we make a new node and squeeze it onto the front of the list. We begin as before by declaring a variable called `base` which will act as an anchor for our list of nodes. `base` is initialised to `null` which indicates that the list has no members: it is *empty*.

Each time we want to add an node to the list, we use a temporary variable `temp` to create the new node. Then we fill in the payload, and set the node's `next` field to link to the same node that `base` points to. This has the effect of linking `next` at the 'rest of the list'. We can then set `base` to link to this new node after which the list update operation is complete. This approach works if the list is empty, or if it has 1,000 nodes. Here's a version of the program that

creates a three node list using the same actions for each update.

```
class IntegerDynamicNode
{
  int payload;
  IntegerDynamicNode next;
}

class SimpleNodes
{
  public static void main(String args[])
  {
    IntegerDynamicNode base = null;
    IntegerDynamicNode temp;

    //Create the first node
    temp = new IntegerDynamicNode();
    temp.payload = 5;
    temp.next = base;
    base = temp;

    //Head insert a new node
    temp = new IntegerDynamicNode();
    temp.payload = 7;
    temp.next = base;
    base = temp;

    //Insert a third node
    temp = new IntegerDynamicNode();
    temp.payload = 9;
    temp.next = base;
    base = temp;
  }
}
```

Here's a diagram showing the third node being added to the list after the line `temp.next = base` has been executed, immediately before `base` is assigned the value of `temp`.



5.5 Singly linked lists

We know that allowing direct access to object fields is bad programming practice: we've only done it here to make the program logic easier to understand. As good software engineers, we really want to produce an *encapsulated* version of our dynamic data structure.

```
class IntegerLinkedList
{
  private IntegerLinkedList next;
  private int payload;

  IntegerLinkedList(int payload, IntegerLinkedList next)
  { this.payload = payload; this.next = next; }

  IntegerLinkedList getNext() { return next; }
  int getPayload() { return payload; }

  void setNext(IntegerLinkedList next) { this.next = next; }
  void setPayload(int payload) { this.payload = payload; }
}
```

We have made the `next` and `payload` fields private, and created appropriate setter and getter methods. We've also written a constructor that takes an `int` payload and a next object reference and fills in the new nodes fields accordingly.

We can use this constructor to simplify the process of head insertion. The basic idiom is illustrated by this fragment.

```
  IntegerLinkedList aList = null;

  IntegerLinkedList temp =  new IntegerLinkedList(value, aList);
  aList = temp;
```

We declare a base variable for the list, called `aList` in this example, and initialise it to `null`. This makes an empty list.

We then declare a temporary variable, and use the constructor to make a list node that carries the desired payload and whose next field is loaded with `aList`, which points to the 'rest of the list'. All that remains is for `aList` to be updated so that the new node is the first node of the list.

Now we can write a program to load a linked list from the keyboard. This is analogous to the array loading program from Section 5.3.5. Since the linked list can expand we no longer need to prompt the user for an extent. Instead, we just read integers until the user types a negative number.

We use a `while(true)` loop which will iterate indefinitely until the `break` statement is executed after a negative number is read. Each time round the loop we use the `IntegerLinkedList()` constructor to add an node to the head

of the list.

```
class LoadIntegerList
{
  static int getInteger(Scanner sc, String prompt)
  {
      System.out.print(prompt);
      return sc.nextInt();
  }

  public static void main(String args[])
  {
    Scanner keyboard = new Scanner(System.in);

    IntegerLinkedList aList = null;

    System.out.println("Negative integer terminates input");

    while (true)
    {
      int value = getInteger(keyboard, "Value: ");

      if (value < 0)
        break;

      IntegerLinkedList temp =
          new IntegerLinkedList(value, aList);
      aList = temp;
    }


    for(IntegerLinkedList temp = aList;
        temp != null;
        temp = temp.getNext())
      System.out.println(temp.getPayload());
  }
}
```

After the loop exits, we use a `for` loop to step over the list nodes. Remember that the general syntax of the `for` loop is

`for (`*initExpression* `;` *predicate* `;` *stepExpression*`)` *statement*`;`

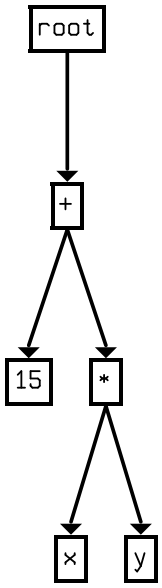
When scanning a list, we declare and initialise a temporary variable of class *reference to object* and assign it the current value of the base of the list. The *predicate* returns true when we get to the end of the list: when `temp` contains `null`. At each iteration, we advance to the next node by setting `temp` to the value of `temp`'s `next` field.

5.5.1 Doubly linked lists

5.5.2 Queues

5.6 Trees

Hierarchies are everywhere in computer science. Files are stored within folders which are stored within folders. We manage the complexity of large programs by using hierarchies of classes and methods. Even expressions form hierarchies because of the different priorities of the operators. This diagram represents the execution order for the expression `15 + x * y`:



We read this left-to-right, depth first, giving `( 15 + ( x * y ) )`.

These kinds of diagrams are called *tree diagrams* and the data structure equivalents are called *trees*. It is a slightly mysterious fact that in computer science, trees always grow downwards.

In a tree, each node has exactly one incoming link, apart from a single node called the *root* node which has no incoming links at all and from which there is a path every other node. As a side-effect of this restriction, the tree always fans out. There can never be a loop in the tree, and two paths in the tree cannot converge. Since non-root nodes have only one incoming link, they must have only one parent. Nodes with no children are called *leaf nodes*. An empty tree has no nodes. In a tree with exactly one node, that node is both the root node and a leaf node. A linked list is a special case of a tree: all linked lists are valid trees.

5.6.1 Binary trees

A *binary* tree is a tree in which the maximum number of links leaving a node is two. The expression tree above is a binary tree.

We can represent binary trees inside the computer using a dynamic data structure with two links per node. In this example, the payload is a `String` variable.

```
class StringTree
{
  private StringTree leftChild;
  private StringTree rightChild;
  private String payload;
}
```

We take a slightly different approach to the construction of tree nodes. We use the constructor to build a node with no links set which would be suitable for use as a leaf node. We then provide methods `addLeftChild()` and `addRightChild` to grow the tree *via* the left or right links.

```
import java.io.*;
class StringTree
{
  private StringTree leftChild;
  private StringTree rightChild;
  private String payload;

  StringTree (String payload) { this.payload = payload; }

  StringTree getLeftChild() { return leftChild; }
  StringTree getRightChild() { return rightChild; }
  String getPayload() { return payload; }

  void setLeftChild() { this.leftChild = leftChild; }
  void setRightChild() { this.rightChild = rightChild; }
  void setPayload(int payload) { this.payload =  payload; }

  StringTree addLeftChild(String payload)
    { return leftChild = new StringTree(payload); }

  StringTree addRightChild(String payload)
    { return rightChild = new StringTree(payload); }
}
```

Now we have all the machinery we need to construct the expression tree. We build the tree from the root downwards, first using the constructor to make

a root node labelled `root` and then adding left and right children appropriately.

```
class LoadStringTree
{
  public static void main(String args[])
  {
    StringTree aTree = new StringTree("root"), temp = aTree;

    temp = temp.addLeftChild("+");
    temp.addLeftChild("15");
    temp = temp.addRightChild("*");
    temp.addLeftChild("x");
    temp.addRightChild("y");
  }
}
```

5.6.2 Tree traversal

Traversing a linked list is easy: we simply hop along from node to node using a simple `for` loop. Trees are harder because we need to potentially take two paths out of each node. What we need is a way of going to a node, remembering where we are; then traversing the left path down to a leaf and returning so that we can traverse the right path. This sort of thing lends itself naturally to recursion: a method like

```
  void traverse()
  {
    if (leftChild != null) leftChild.traverse();
    if (rightChild != null) rightChild.traverse();
  }
```

when called on the root node will systematically visit every node in the tree exactly once. Note the way that the recursion 'bottoms out' at leaf nodes because the recursive call is only made if a link is non-null.

5.6.3 Preorder, inorder and postorder traversals

In real applications, we do not merely want to traverse a tree, we want to take some action at each node. It turns out that we get rather different effects depending on whether we perform the action before visiting the children (a *preorder* traversal), after visiting the children (a *postorder* traversal) or between visiting the children (an *inorder* traversal). The sequencing of the node related actions changes quite radically with these three possibilities.

We'll illustrate the effects by adding traversal functions to the `StringTree`

class that simply print the node labels in pre-, post- and inorder.

```
  void printInorder()
  {
    if (leftChild != null) leftChild.printInorder();
    System.out.println(payload);
    if (rightChild != null) rightChild.printInorder();
  }

  void printPreorder()
  {
    System.out.println(payload);
    if (leftChild != null) leftChild.printPreorder();
    if (rightChild != null) rightChild.printPreorder();
  }

  void printPostorder()
  {
    if (leftChild != null) leftChild.printPostorder();
    if (rightChild != null) rightChild.printPostorder();
    System.out.println(payload);
  }
}
```

If we add this fragment to the `main()` method

```
    System.out.println("Preorder traversal:");
    aTree.printPreorder();
    System.out.println("Inorder traversal:");
    aTree.printInorder();
    System.out.println("Postorder traversal:");
    aTree.printPostorder();
```

we get these outputs. You should work through the traversal functions manually, and confirm to yourself that the results are as you would expect.

```
Preorder traversal:
root
+
15
*
x
y
```

```
Inorder traversal:
15
+
x
*
y
root
```

```
Postorder traversal:
15
x
y
*
+
root
```

5.6.4 Multiway trees

5.7 The Java collections framework

6 Interaction

Most of the time, our programs are busy shuffling and manipulating data in the computer's Random Access Memory (RAM). *Interesting* programs have to interact with their environment, even if all they do is print out some results. We can identify four kinds of interaction:

console writing characters to the console window and reading data from the keyboard;

disk reading and writing text and raw data to and from disk files;

network exchanging data with applications and services running independently of our current program *via* network protocols; and

GUI interacting with users *via* Graphical User Interfaces.

So far we have used the simple console facilities: `System.out.print()` and `System.out.println()` to print strings; and various `Scanner` methods to read from the keyboard.

Java provides classes to handle a multitude of interesting interactions, and the scale and richness of these libraries can be quite overwhelming. In this chapter we're going to look at the basic landscape of these libraries, showing you just enough to get started. You can go and learn more for yourself with the help of some of the online resources listed on the course web page.

6.1 Computers and the real world

In Chapter 2 we presented a simplified picture of a computer as a pigeon hole memory, a disk and a central processing unit. Figure 6.1 shows an expanded version that includes most of the devices associated with a modern laptop, along with some attached devices such as a games controller and a Lego robot. Our programs need to be able to control and communicate with all these different systems. How are we to manage this complexity?

6.1.1 Early attempts

Very early computers were provided with a fixed set of *peripheral devices* such as card readers and line printers. Since there were only a few possibilities, it was quite normal for specific primitive instructions to be provided that accessed

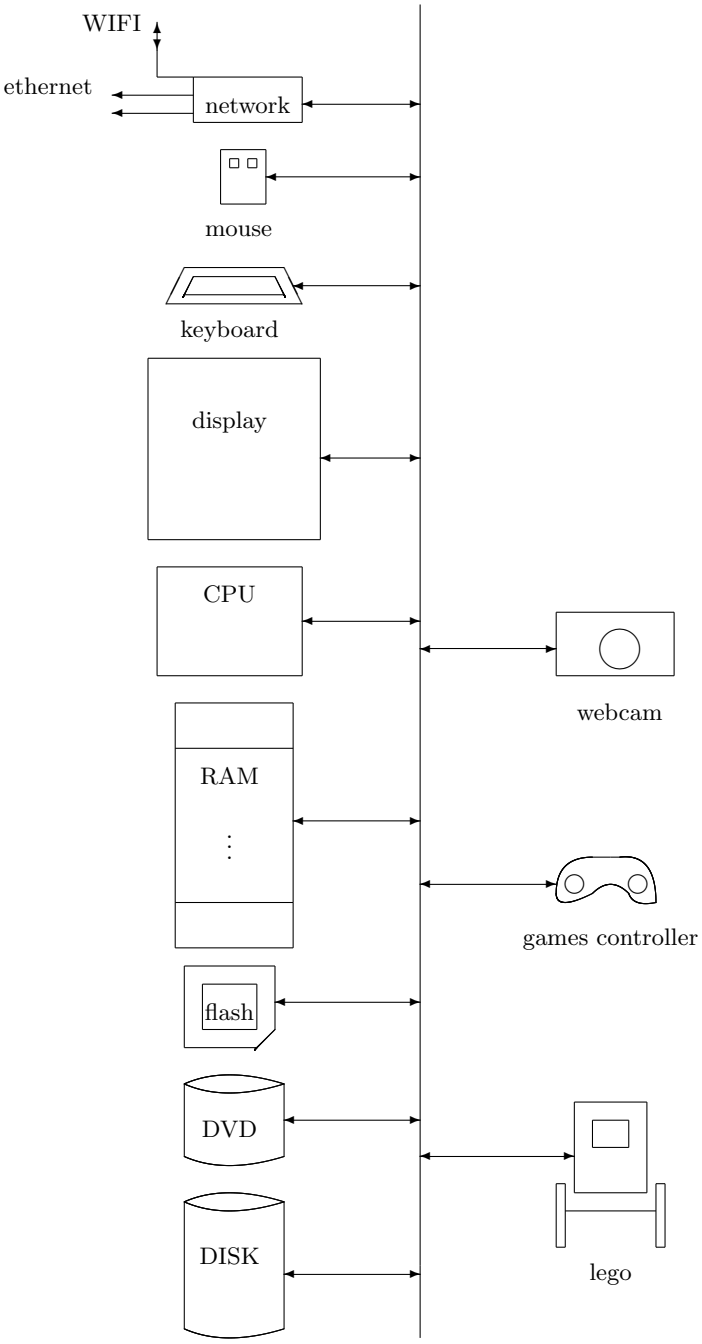


Figure 6.1 Real world computers

each device. In a very real sense, the peripherals were hardwired into the design of the main computer.

Of course, this approach was very limiting. Over the lifetime of a computer, technology would improve, and it might be impossible to take advantage of new, faster, peripherals without rewiring the computer's central processor to allow new kinds of operations. Apart from being very expensive, machines that had been enhanced ran programs that were no longer backwards compatible with the original system, so program portability was sacrificed.

6.1.2 Device drivers

A better way of handling peripherals was needed; one that made programs independent of the particular combination of devices in a system. The solution was to standardise the software conventions for sending and receiving data and to use small independent special purpose programs to translate between the standard convention and the details of a real device. These small programs are called *device drivers*, and today it is usually the responsibility of the manufacturer of a device to supply the drivers.

6.1.3 The speed gap

Peripherals are often mechanical devices, and may require tens of milliseconds to complete an operation. In one millisecond, a modern processor can execute a million instructions, and it would thus be very inefficient to make the central processing unit wait around whilst a printer actually outputs a line of text. As a result, many input/output operations (including writing to a disk, sending data via a network and printing) need to run *asynchronously*. Our programs start the operation, but then the device driver runs independently at its own speed in parallel with our program. Each time the peripheral device needs the next piece of data, our program is paused or *interrupted* and the computer switches over to running the device driver. When the device driver has fed the next piece of data to the printer, disk or network adapter it passes control back to our program. In this way, we can send a document to the printer and keep working whilst the printer slowly processes its data stream.

6.1.4 Operating systems

Once we have the capability to interrupt programs whilst device drivers run, we could also interrupt programs to allow other, normal programs to run. Pretty much every general purpose computer designed since the late 1950's is designed to allow this way of working. In the early days, the emphasis was on allowing lots of different people to access one machine — this is how we use *Zeno*, the main undergraduate server. Later, as desktop computers became affordable, the goal was to have multiple programs running to support one individual: a word processor, multiple web browsers and an email program that periodically wakes up and checks for new mail, for instance.

This way of working presents a new challenge, though. What happens if more than one user, or more than one program, wants to access a peripheral at the same time? We need some way of arbitrating between programs and ensuring that input/output requests are satisfied in an orderly fashion. The way we do this is to have a special privileged program called the *operating system* that schedules all requests, and also loads and runs the user programs. On most computers, the operating system is automatically given control of the computer every 20 milliseconds or so, by generating interrupts from a 50kHz clock. The operating system can, for instance, switch between user programs on each clock tick. In this way two programs can be run in alternate 20 millisecond slots, and to the human users, it appears that both programs are running simultaneously. We call these independently running programs *processes*, and our computers can run hundreds of processes apparently simultaneously. In reality, a computer's processor core is at any one instant running exactly one program — user program, device driver or operating system — but the switching is so fast that it all seems like they are all running all of the time. Modern desktop and laptop computers do in fact have multiple cores in them, but each core runs multiple processes. It turns out to be quite difficult to write applications that make good use of multiple cores.

Well known operating systems include Linux, Windows, and Mac OS (which, like Linux, is a variant of Unix). You may also use Google's Android operating system for mobile phones.

When users think of operating systems, they tend to think about the windowing interface style and the applications that come with it. Really, though, it is the scheduling and device support that characterises an operating system to Computer Scientists. It's a bit like cars: the Seat Ibiza and Volkswagen Polo use essentially the same underpinnings, with some tweaks to make cars that look quite different. To an engineer, it is the chassis that is interesting, not the shape of the windows.

6.2 The Java approach

A perennial problem in the design of portable programs is to ensure that the interaction facilities are independent of the host operating system. Of the four classes of interaction that we noted above, console and disk I/O have long been standardised, but Java constitutes the first serious attempt to provide operating system independent networking and GUI facilities, and in fact the support for console and disk I/O is much more flexible and comprehensive than in previous programming languages.

All of this capability comes at a price though: the Java libraries that support interaction are extremely extensive. It is quite impossible in a first programming course to give you more than the basics. However, a rather compact set of features can serve to implement most of what you need for simple applications.

Of the four classes of interaction (console, disk, network and GUI) the first three are managed with a common set of abstractions and classes which describe *files* to from which *streams* of binary data may be directed, sometimes under

the control of character level *readers* and *writers*. GUI interactions were originally handled by a large library called the *Abstract Windowing Toolkit* (AWT) which presented to the programmer a uniform set of facilities which were translated whilst a program was running into GUI operations for the particular host operating system. AWT was superseded by the *Swing* libraries which directly implement a windowing system in Java itself, with only the basic operations of window creation and mouse input being left to the host operating system. Swing uses the core of AWT, so rather confusingly, programmers need to know about some of the older-style AWT-style facilities to make full use of Swing.

We'll consider the file/stream/reader/writer style interactions first, and then look at simple GUI programming using Swing.

6.3 Files, streams, readers and writers

In early operating systems, a *file* was simply a region of a rotating magnetic disk which contained some document or dataset, with a name by which it could be accessed. A terminal, or a printer was a completely different kind of resource, and different programs and operating system features were used to manage terminals, printers and disk files.

Around 1970, the Unix operating system first introduced the notion of a unified representation for disk files and peripheral devices. Most operating systems now use this convention. In Java programs, we use a three level abstraction to handle general data, character data and file based data.

A stream is a unidirectional channel between a program and either (i) an output device such as a printer or writable disk file or (ii) an input device such as a keyboard or readable disk file. Streams carry binary data in the form of individual bytes.

A reader or writer is a unidirectional channel that carries 16-bit character data. Readers and writers are high level abstractions that wrap round the basic binary streams.

A file is an entry in a disk's directory corresponding to a region of the disk. We can look up existing files, or create new ones and associate them with streams and by extension readers and writers.

The basic I/O mechanism in Java is the *stream*. We can associate a stream with a disk file by constructing a `File` object that represents it, and we can enhance a stream so that it comfortably handles Unicode character data by constructing a `Reader` or `Writer` object that wraps some special methods into the stream object.

6.3.1 Predefined streams

Java provides three important predefined streams that by default connect to the console, but which can be redirected from the command line to files.

`System.in` the keyboard for the console that started this program;

`System.out` the screen for the console that started this program;

`System.err` another stream that sends messages to the console screen, but intended for error messages rather than standard output.

The `err` and `out` streams are separated so that error messages can be redirected by the user to a log file without having to plough through non-error output.

6.3.2 Accessing a named file

In Java, we connect to or create a file by creating an object of class `File`. The most commonly-used constructor takes a filename (possibly containing a directory path) as a string.

```
String filename = "test.txt";

File file = new File(fileName);
```

Some of the higher level objects for controlling interaction automatically create `File` objects as necessary: sometimes all we have to do is specify the filename string.

6.4 The class `Scanner`

We've already made extensive use of the `Scanner` class that is included in recent versions of Java. `Scanner` objects can read characters from the keyboard and group them into *tokens* according to a variety of (programmable) rules. We can generate some very complicated effects with `Scanner` objects, but the most common application is simply to read numbers and strings from the keyboard or a file. The `Scanner` class includes constructors that will create a scanner reading from a `File`, an `InputStream` and even from a `String`.

6.4.1 Echoing a file

As a first example, here's a program that reads tokens from a file and simply echoes them to the screen. The name of the file is supplied as a command line

argument.

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class SimpleScanner
{
  private static void readFile(String fileName) throws
                      FileNotFoundException
  {
    File file = new File(fileName);
    Scanner scanner = new Scanner(file);

    while (scanner.hasNext())
    {
      System.out.println(scanner.next());
    }
    scanner.close();
  }

  public static void main(String[] args) throws
                     FileNotFoundException
  {
    if (args.length != 1)
    {
      System.err.println("usage: java FileScanner <file>");
      System.exit(0);
    }
    readFile(args[0]);
  }
}
```

The `main()` method checks that a filename has been supplied, issuing a short help message if not. Then we pass the first command line argument `args[0]` to method `readFile()`, which creates `File` and `Scanner` objects before reading through the file.

The `Scanner` class provides methods `hasNext()` and `next()` which test to see if there is anything left to scan, and scan the next token respectively. We use `hasNext()` to control the while loop so that the program terminates tidily

when the end of the file is reached. If we run this program on the following file

```
adrian
was
here 123
32.0

456 , 789
```

we get this output

```
adrian
was
here
123
32.0
456
,
789
```

Notice how the scanner absorbs arbitrary amounts of whitespace, including line breaks, between tokens. It turns out that the definition of whitespace is itself a `Pattern` object, and we can change it. That way we could make line ends visible as their own special token, for instance, if we wanted to count blank lines.

6.4.2 Conditional scanning

The real power of the `Scanner` class derives from its ability to *look ahead* in the input stream, so we can test to see what is coming next.

```
import java.io.File; import java.io.FileNotFoundException; import
 java.util.Scanner; import java.util.regex.Pattern;

public class SelectScanner
{
  private static void readFile(String fileName) throws
                      FileNotFoundException
  {
    File file = new File(fileName);
    Scanner scanner = new Scanner(file);

    while (scanner.hasNext())
    {
      int scannedInt;
      double scannedDouble;
      String scannedString;
      Pattern comma = Pattern.compile(",");

      if (scanner.hasNextInt())
      {
        scannedInt = scanner.nextInt();
        System.out.println("Scanned int " + scannedInt);
      }
      else if (scanner.hasNextDouble())
      {
        scannedDouble = scanner.nextDouble();
        System.out.println("Scanned double " + scannedDouble);
      }
      else if (scanner.hasNext(comma))
      {
        scanner.next(comma);
        System.out.println("Scanned a comma");
      }
      else
      {
        scannedString = scanner.next();
        System.out.println("Scanned other " + scannedString);
      }
    }
    scanner.close();
  }
}
```

This program is essentially the same as the file-echoing program, except that we have added a chain of `if` statements to the body of the `while` loop which test to see what kind of token is coming next. There are two ways of doing this: using variants of `hasNext()` such as `hasNextInt` and `hasNextDouble` or by supplying a `Pattern` object as a parameter to the `hasNext()` method.

A pattern is a precompiled *regular expression* that describes the 'shape' of a text string. We can do powerful things with regular expressions, but this simple example just makes a pattern corresponding to a comma. That way, we can read comma delimited strings of numbers.

The general approach with these kinds of scanning programs is to loop over the file testing for each individual case we are interested in (integers, doubles and commas in this example) with a final catch-all case that simply grabs the next token as a `String`.

In this example we're just printing the tokens out with a message to say what flavour of token they are. In a real application, we would have processing conditional on the token flavour.

6.5 Formatted output

Recent versions of the Java libraries include *formatted output* methods such as `printf()` which take a format string with embedded format placeholders and a list of variables and constants to be printed. The idea is that the string is printed, and as we work along we match the placeholders with the next variable or constant in the list, formatting it accordingly.

By carefully specifying the format, we can alter the number of decimal places printed for a `double`, change the width of a field so that numbers always appear right justified and print integers as hexadecimal numbers.

Format placeholders are introduced by a percent sign `%`. The most common ones are `%d` for a decimal integer, `%f` for a decimal floating point number, `%X`

for a hexadecimal integer and %% for a percent sign. Here's an example.

```
import java.lang.Math;
import java.io.PrintWriter;
import java.io.FileNotFoundException;

public class FormattedOutput
{
  public static void main(String[] args)
                throws FileNotFoundException
  {
    System.out.printf("pi to five digits: %.5f\n", Math.PI);
    System.out.printf("pi to four digits: %.4f\n", Math.PI);

    int temp = 1020;
    System.out.printf("Temp in decimal: %d\n", temp);
    System.out.printf("Temp in hex: %x\n", temp);
    System.out.printf("Temp in HEX: %X\n", temp);
    System.out.printf
      ("Temp as a zero padded four digit field %04X\n", temp);

    PrintWriter textfile = new PrintWriter("test_out.txt");

    textfile.printf
      ("Temp to file as a zero padded four digit field %04X\n",
       temp);

    textfile.close();
  }
}
```

Most of the output is produced with called so `System.out.printf()`. Here is what appears on the screen when the program runs.

```
pi to five digits: 3.14159
pi to four digits: 3.1416
Temp in decimal: 1020
Temp in hex: 3fc
Temp in HEX: 3FC
Temp as a zero padded four digit field 03FC
```

As you can see, we can control the number of decimal places printed for a floating point number by using a placeholder like `%.5f`. The `%x` placeholder prints hexadecimal integers using lower case whereas the `%X` version uses upper case.

This program also creates a `PrintWriter` object, which is an extension of the `Writer` class that supports formatted printing. We have used it here to

open a new text file for writing, and then send this single line of output to it before closing the file.

```
Temp to file as a zero padded four digit field 03FC
```

6.6 Graphical user interfaces

Graphical User Interfaces (which used to be known as Window-Icon-Mouse-Pointer or WIMP interfaces) have a long history, and are now ubiquitous on desktop computers. They bring special challenges to the programmer, because a GUI is inherently *event driven*. Typically, a GUI based program initialises its data structures, opens one or more windows and then settles down to wait for user input. This is very different to the typical batch mode program which takes its instructions from the command line, then goes off and performs its task before terminating. Some GUI programs never explicitly terminate: instead the operating system terminates them when the last window closes.

Apart from being event-driven, GUI programs are much more dependent on the host operating system than command line driven programs. The details of opening a window and drawing within it can be very complicated. Even using a device driver to shield the programmer from nasty details is not enough because GUI's often need to directly access the graphics subsystem so as to ensure that windows are responsive to user input. The best way to handle all this complexity is to allow the operating system itself to take care of window positioning and management, but that means we have to make our program subordinate to the operating system, rather than our program being in charge (as it usually is for command line programs).

The usual way this is handled is for us to write our GUI programs as a collection of objects and methods which fall into these categories:

- ◇ setup code run which creates graphical object that correspond to onscreen effects;
- ◇ listener methods which responds to mouse and other events when called by the operating system;
- ◇ action methods which perform program functions, usually called by listener methods.

The setup code creates the initial windows and other GUI components and ensures that the operating system knows the names of the listener methods, and to which GUI component they apply. We say the methods are *registered* with the operating system.

The operating system must be in overall charge of a GUI program, but it typically causes our program to hibernate, waiting for input events or repaint operations, at which point it calls the program's appropriate methods. Event listeners can call action methods which might open new windows and register new methods. The program only terminates in response to a specific action.

6.6.1 The Java approach

Java uses object oriented techniques to make GUI implementation easier, although not perhaps easy. There have been three main waves of GUI support in Java. The original system was called the *Abstract Window Toolkit* or AWT. The idea was that a set of standard wrapper classes would be provided for the host operating systems GUI operations, so that a single application could run on, say, Apple systems with Apple look-and-feel and on Windows systems with Windows' look-and-feel. AWT turns out to be quite cumbersome, because some of the processing needed to convert the general calls into system-specific calls is very intense.

AWT was superseded in 1998 by the *Swing* graphical libraries (although early versions were in use before that). Swing is built on top of AWT which it uses to handle the top level interaction with the operating system, such as opening a window and tracking the mouse. However, the *contents* of a Swing window is controlled not by the native operating system features but by code written in Java. So as to allow Java programs to look like other programs, Swing also provides a look-and-feel feature that enables the Swing graphical components to adjust themselves to look like, say, Window-XP components or Mac components.

In the last few years an alternative approach called Java FX has been developed, and from 2014 with the Java 8 release Java FX is the recommended way to build GUI applications. We are going to look at Java FX programming here; from section 6.6.5 onwards you will find a discussion of how to build the same program in Swing in case you need to build or maintain legacy applications that were based on these earlier libraries.

A major distinction between JavaFX and the earlier systems is the way that screen updates are handled. In Swing, there are explicit repaint methods which redraw window contents when called by the operating system. In JavaFX, we give overall control to the JavaFX API, and it looks after repaints and some other book keeping for us. This makes GUI programming much easier, but we have to change the way we build our applications. Instead of having a top level class which contains a `main()` method, our primary application class extends a special JavaFX class called `Application` which contains its own `main()` method. Our program begins with a method called `start()` instead.

6.6.2 Java FX

The main innovation in JavaFX over earlier Java UI toolkits is the emphasis on *scene graphs*. This idea really comes into its own when building 3D graphics systems of the sort you need for realistic games programs, but it turns out to be very useful even for simple user interface tasks.

We are already familiar with the idea of a directed tree in Computer Science as a branching structure with a root node that has children, which may have children and so on. The overriding property of a tree is that each node is the child of at most one other node; only the root has no parents, and the nodes with no children are called leaves.

In JavaFX we describe the picture we want to appear on te screen as a tree of objects. For instance, we might want the window divided into four separate pans, andthe top left one might need a button in it. The panes would be top level children, and the top left pane would have a button as a child.

In general, we want to be able to construct scene graphs independently of updating the screen. For instance, when displaying an animation we might want to display the first scene, then construct the next scene invisibly and then 'reveal' the scene all in one go. This makes sure that users do not see flickering or other artefacts whilst scenes are being put together.

The metaphor that JavaFX uses to support this is called a *stage*. Essentially, a stage corresponds to an open, visible window on the screen, and scene's are projected onto stages when they are ready to be displayed.

6.6.3 The click application in Java FX

In this section we shall examine a very small drawing program which exercises most of the basic features of JAvaFX. However, this is really only the start because JavaFX contains many, many features which we cannot cover within the time available. You'll find many online resources that can help you develop your understanding if you want to go further.

The user is presented with a blank window containing a single button labelled Clear. The window represents a grid of squares ten pixels on a side. When the mouse is clicked on a square it changes colour: in fact the colours cycle through red, green, cyan, blue, black and back to white again. If the Clear button is pressed, the window is reset to all white.

Here is a screen shot of the `click` application running:

This GUI application is closely related to the GUI component of the second assignment (Conway's game of Life). Try to running this before you attempt part **??** of the assignment.

6.6.4 Click application code for JavaFX

Now, be warned, this code is going to be incantation-heavy, by which I mean that some parts will be unfamiliar when you encounter them. Remember that the best way to learn about new Java API methods and classes is to look them up in the online Java pages.

If you cut and paste the code boxes in this section into a single file called Click.java, you will have a complete application that will compile and run if you have a Java 8 system.

We begin with a large set of imports. Systems such as Eclipse can pull these in automatically.

```
package clickFX;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.canvas.Canvas;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.control.Button;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.HBox;
import javafx.scene.layout.VBox;
import javafx.scene.paint.Color;
import javafx.stage.Stage;
```

Recall that our JavaFX GUI application does not have a `main()` method — instead we have a class which has the same name as our application and which extends JavaFX's own `Application` class.

Now, some Java systems do not yet support JavaFX. So as to provide a failsafe startup, most JavaFX applications conventionally do provide a `main()` method which in normal use is never called. Treat this as an incantation.

We then declare some data. The `GraphicsContext gc` and `Stage primaryStage` are used to hold information given to us by the JavaFX runtime system. The other items are part of our application: `ClickData clickdata` is the object that looks after the data backing our array of colour squares; `int squareSize` is the size of an onscreen square in pixels, and the `Color colours[]` array

defines the sequence of colours that we shall cycle through on mouse clicks.

```
public class Click extends Application {

  public static void main(String[] args) {
    launch(args);
  }

  GraphicsContext gc;
  ClickData clickData;
  int squareSize;
  Stage primaryStage;
  Color colours[] = {Color.WHITE, Color.RED, Color.GREEN, Color.CYAN,
      Color.BLUE, Color.BLACK};
```

We now proceed to the `start()` method which is passed a `Stage` parameter which is the main stage for this application. We can open more if we want further onscreen windows.

We begin by initialising the `clickData` object and setting `squareSize` to 16.

The rest of this method sets up the scene graph. We make an `HBox` — a box whose elements are laid out horizontally called `Buttons`. This is going to be the container for our clear and close buttons. We create appropriate button objects, and then by calling `buttons.getChildren().addAll()` we associate the buttons with the `Hbox`. What is really going on here is that we are defining the bottom layer of the scene graph. The buttons are the leaves of the tree, and the `Hbox` is the parent of the buttons.

We then make a `Canvas` object which is how JavaFX provides us with a drawing surface. We add a mouse handler to the canvas so that when we click within the canvas we can perform some action.

We also make a `Vbox` called root which is going to act as the root of our scene graph, and we add the canvas and the buttons box to it as children.

In the last two lines, we set the scene on our primary stage, and call

`primaryStage.show()` which causes the window to display.

```
public void start(Stage primaryStage) {
  squareSize = 16;
  int x = 20, y = 30;
  clickData = new ClickData(x, y);

  VBox root = new VBox(8);

  HBox buttons = new HBox(5);
  buttons.setAlignment(Pos.CENTER);

  Button clearButton = new Button("Clear");
  clearButton.setOnAction(new clearButtonHandler());

  Button closeButton = new Button("Close");
  closeButton.setOnAction(new closeButtonHandler());

  buttons.getChildren().addAll(clearButton, closeButton);

  Canvas canvas = new Canvas(x * squareSize, y * squareSize);
  gc = canvas.getGraphicsContext2D();
  canvas.addEventHandler(MouseEvent.MOUSE_CLICKED,
      new mouseClickHandler());

  this.primaryStage = primaryStage;
  primaryStage.setTitle("Click me");
  root.getChildren().addAll(canvas, buttons);

  primaryStage.setScene(new Scene(root));
  primaryStage.show();

}
```

Now that we have set up the graphical elements, we can turn our attention to the button and mouse handlers.

There are some more incantations here. In particular, you will see instances of `implements <SomeThing>` which is new syntax that you have not previously encountered. In a later section of the course you will learn that as well as extending classes, we can associate classes with *interfaces*. For now, just follow the text.

The `clearButtonHandler()` method is called whenever we click on the clear button. It overwrites the canvas with a white rectangle, and clears the data array back to its initial value of all zeroes.

The `closeButtonHandler()` is called when the close button is clicked; it closes the primary stage. When the primary stage of a JavaFX application closes, the application is shut down.

The `mouseClickHandler()` method is called whenever a user clicks within the canvas area. It is passed a paramater of class `MouseEvent` which we can use to find out what kind of mouse event occured; for instance one of the buttons may have been pressed or released. We can also use the paramater to recover the coordinates of the event. In our application we are ignoring the kind of event, but we are using the coordinates. We output a message, and increment the value of the associated cell. We then set the fill colour and draw a rectangle.

```
class clearButtonHandler implements EventHandler<ActionEvent> {
  public void handle(ActionEvent e) {
    clickData.clear();
    gc.setFill(Color.WHITE);
    gc.fillRect(0, 0, clickData.getX() * squareSize, clickData.getY()
        * squareSize);
  }
}

class closeButtonHandler implements EventHandler<ActionEvent> {
  public void handle(ActionEvent e) {
    primaryStage.close();
  }
}

class mouseClickHandler implements EventHandler<MouseEvent> {
  public void handle(MouseEvent e) {
    double x = e.getX(), y = e.getY();
    int cx = (int) x / squareSize, cy = (int) y / squareSize;

    System.out.println("Click (" + x + ", " + y + ")");

    clickData.setCell(cx, cy, clickData.getCell(cx, cy) + 1);

    gc.setFill(colours[clickData.getCell(cx, cy) % colours.length]);
    gc.fillRect(cx * squareSize, cy * squareSize, squareSize,
        squareSize);
  }
}
```

In this final section, we make a class that encapsulates a two dimensional array of integers with appropriate getters and setters, and with a clear method

which returns the array to its initial state.

```
  class ClickData {
    int[][] grid;

    ClickData(int x, int y) {
      grid = new int[y][x];
    }

    public int getX() {
      return grid[0].length;
    }
    public int getY() {
      return grid.length;
    }
    public int getCell(int x, int y) {
      return grid[y][x];
    }
    public void setCell(int x, int y, int v) {
      grid[y][x] = v;
    }

    public void clear() {
      for (int y = 0; y < getY(); y++)
        for (int x = 0; x < getX(); x++)
          setCell(x, y, 0);
    }
  }
}
```

6.6.5 Click FX in Java swing

So as to support pre-Java 8 systems, here is a description of how to build the click application using Swing. This is not examinable material; you are expected to use JavaFX for new code development. The idea for putting this material in the book is to give you a starting point if you ever need to maintain a legacy application.

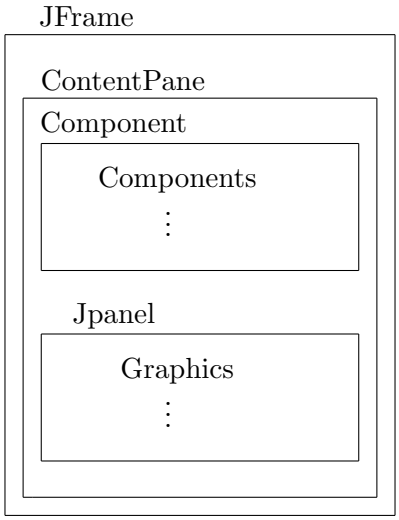
6.6.6 Containers and components

If we look at a running GUI program we see some (possibly overlapping) windows which contain panels with buttons, menu bars and other controls. In Java Swing, we abstract these relationships by using *container* objects which can nest: containers hold components which may themselves be containers. The specific spatial relationships between components as drawn on the screen is mostly left to a *layout manager* which decides whether to place components

left-to-right, or in a ring, or in some more complicated layout. This frees the programmer from a great deal of clerical work.

At the top level, standalone programs have a class based on `Jframe`. Creating an object derived from `JFrame` causes a window to open on the screen. These frames have several graphical areas called *panes* within them corresponding to the menu bar, the main content area and a special pane which covers the whole window area. Usually we add containers to the content pane, and then put Swing components or AWT graphics into those containers.

Here's a diagram showing a typical (simple) Swing window.



6.6.7 Callbacks for painting and listening

The prototypical components and containers include methods that are designed to be called by the operating system. We call these methods *call backs*. When objects of classes extended from the prototypical classes are created they register themselves with the operating system and Java Swing runtime environment.

The most important call-back is `void paintComponent(Graphics g)` which is called whenever a component needs to be redrawn. So, for instance, if our application window is moved, or de-iconised or revealed because another window has moved, the operating system automatically calls the `paintComponent()` method with a particular graphics canvas `g` onto which we can draw.

You should never call `paintComponent()` yourself. Instead, each component has a method `void repaint()` which you can call to indirectly retrigger a repaint. The Java Swing runtime environment schedules a repaint operation and at some time in the future the object's `paintComponent()` method will be called.

There are a whole set of *listener* callbacks that are used to trigger code execution when mouse events occur within one of our windows. By default, components do not have mouse listeners. Instead, a special kind of class called an *interface* lists the necessary methods that you must write. You add mouse capability to a component by appending `implements MouseInputListener` to

the class description, which specifies that your class must implement *all* of the listener methods.

6.7 A first example

This is a *Hello World* example for Java Swing. We first have to import both the old-style AWT library and the Swing library. Our `main()` method then creates a new `JFrame` object which creates an operating system window (although it does not become visible yet).

```
import java.awt.*;
import javax.swing.*;

class FirstGUI
{
  public static void main(String[] args)
  {
    JFrame f = new JFrame("FirstGUI");
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    f.setSize(200, 300);
    f.setLocation(100,100);

    Container content = f.getContentPane();
    content.add(new JLabel("Hello World!"));
    f.setVisible(true);
  }
}
```

We then set the size and location of the window, and specify that when the window closes, the application should terminate.

Now we need to display something in our window. The method `getContentPane()` return to us a container that corresponds to the user area of the window. We can add any kind of Swing component we like, and the layout manager will take care of the details of its positioning and so on, even if the window changes size subsequently. In this example, we add a `JLabel` component, which displays a text string in the window.

Finally, we make the window visible with `setVisible(true)`. If this were a non-GUI program, it would now terminate because we have reached the end of the `main()` method, but by creating a `Jframe` object we have ceded overall control now to the operating system which will keep it running until the user closes the window using the host operating system's standard window close button.

6.8 The `Click` application

Real Swing applications can be very large programs: Swing provides an extraordinary depth of features. In this section we shall examine a very small drawing program which exercises most of the basic features of Swing.

The user is presented with a blank window containing a single button labelled Clear. The window represents a grid of squares ten pixels on a side. When the mouse is clicked on a square it changes colour: in fact the colours cycle through red, green, cyan, blue, black and back to white again. If the Clear button is pressed, the window is reset to all white.

We begin by importing the main Swing and AWT libraries along with their `event` libraries which define objects used to tell the listener methods in our program which events to respond to. We also import some colour definitions (watch out for American spelling!).

```
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.Color;
import java.io.*;
```

The array of squares is backed up by a data structure called `ClickData` which has: a constructor allowing arbitrarily sized grids to be made; three getter methods to return the dimensions of the grid and the value of an individual element; and a setter method to set the value of an element.

```
class ClickData
{
  private int[][] grid;

  ClickData(int x, int y) {grid = new int[y][x];}

  public int getX() {return grid[0].length; }
  public int getY() {return grid.length; }
  public int getCell(int x, int y) {return grid[y][x];}
  public void setCell(int x, int y, int v) {grid[y][x] = v;}
}
```

The `main()` method defines the dimensions of the grid before creating the data array, a frame, a component to hold the array of coloured squares and a button. The component and the button are added to the frame with layout controls that place them with the button at the bottom of the window. We then call the frame's `pack()` method to adjust the window size so that it snaps

around the button and the component.

```
class Click
{
  public static void main(String[] args)
    throws IOException, FileNotFoundException
  {
    int squareSize = 10, x = 30, y = 40;

    ClickData click = new ClickData(x, y);
    ClickFrame frame = new ClickFrame();
    ClickComponent component =
      new ClickComponent(click, squareSize);

    JButton button = new JButton("Clear");
    button.addActionListener(new ClearListener(click, frame));

    frame.add(component, BorderLayout.NORTH);
    frame.add(button, BorderLayout.SOUTH);
    frame.pack();
  }
}
```

The button needs some code that will be called by the operating system when it is pressed. We make a class that implements the `ActionListener` interface with a public method `void actionPerformed(ActionEvent e)`. Inside this method, we place the code to clear the array.

```
class ClearListener implements ActionListener
{
  ClickFrame f;
  ClickData c;

  ClearListener(ClickData c, ClickFrame f)
  { this.c = c; this.f = f; }

  public void actionPerformed(ActionEvent e)
  {
    for (int y = 0; y < c.getY(); y++)
      for (int x = 0; x < c.getX(); x++)
        c.setCell(x, y, 0);
    f.repaint();
  }
}
```

Our frame is a small extension of the standard one that we used in the previous example: we change the title and the close operation within the constructor

instead of in the `main()` method.

```
class ClickFrame extends JFrame
{
  ClickFrame()
  {
    setTitle("Click me");
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setVisible(true);
  }
}
```

Finally, we have the main component. This extends, and overrides some of the methods of, `JComponent`. It also connects to the mouse by implementing the `MouseInputListener` interface.

The class looks quite complicated, but it breaks down into the following pieces.

- ◇ A constructor `ClickComponent` which sets the preferred size of the component and registers the mouse listener methods.

- ◇ The `paintComponent()` method which uses the `map[]` array to decide which colour to paint each square in the window.

- ◇ Seven `mouse...()` listener methods which will be called by the operating system whenever the corresponding mouse event occurs with the piece of screen real estate corresponding to this component. Six of the seven are unused. The `mouseClicked()` listener increments the value of the array element corresponding to the decoded screen coordinates of the click, before calling `repaint()` to schedule a call to `paintComponent()`.

This is quite an interesting class because it actually has no methods which are ever called from within other user classes: the constructor is automatically executed when a new object of class `ClickComponent` is made; `paintComponent()` is only ever called by the Swing runtime environment and the mouse listener methods are only ever called by the operating system.

This style is typical of Swing programming. We make graphical components (sometimes called *widgets*) and put them into containers. We rely almost

completely on the operating system to schedule their execution.

```
class ClickComponent extends JComponent
                     implements MouseInputListener
{
  private ClickData c;
  private int squareSize;
  private Color map[] = {Color.WHITE, Color.RED, Color.GREEN,
    Color.CYAN, Color.BLUE, Color.BLACK};

  ClickComponent(ClickData c, int squareSize)
  {
    this.c = c; this.squareSize = squareSize;

    setPreferredSize(new Dimension(c.getX() * squareSize,
                                   c.getY() * squareSize));
    addMouseMotionListener(this);
    addMouseListener(this);
  }

  protected void paintComponent(Graphics g)
  {
    super.paintComponent(g);

    for (int y = 0; y < c.getY(); y++)
      for (int x = 0; x < c.getX(); x++)
      {
        g.setColor(map[c.getCell(x,y) % map.length]);
        g.fillRect(x * squareSize,y * squareSize,
                   squareSize, squareSize);
      }
  }

  public void mousePressed(MouseEvent e) {}
  public void mouseDragged(MouseEvent e) {}
  public void mouseReleased(MouseEvent e) {}
  public void mouseClicked(MouseEvent e)
  {
    int x = e.getX(), y = e.getY();

    System.out.println("Click (" + x + ", " + y + ")");
    x /= squareSize; y /= squareSize;
    c.setCell(x, y, c.getCell(x, y) + 1);
    this.repaint();
  }
  public void mouseEntered(MouseEvent e) {}
  public void mouseExited(MouseEvent e) {}
  public void mouseMoved(MouseEvent e) {}
}
```

A Assignment one – breaking the Caesar cypher

Please see the 'Coursework requirements and deadlines' section of our website for submission deadlines, and for the contribution to your final course mark.

Assessment criteria and feedback

The parts of this assignment are not all of the same weight: the marks for each question are shown below. Some marks will be given for a correct approach even when the overall solution is incomplete or incorrect.

This assignment will be marked by the course lecturer and feedback given; please note that the feedback grades are provisional and subject to moderation by our external examiners during the examination process.

The primary purpose of this assignment is *formative*: it provides practice in basic Java programming and gives you an opportunity to assess for yourself the degree to which you have understood the material

You should try all parts and hand in what you have done, even if you are not able to complete some of the questions. Solutions to the assignment will be discussed in a lecture slot after the assignments have been marked.

If you do not know how to do part of the assignment, begin by reading the corresponding text in the notes and working through related examples discussed in lectures. Please feel free to make an appointment to talk to me about anything you don't understand, and remember solutions *must be the submitter's own work*.

If someone asks you for help you may discuss the material and examples given in class but you should not give any form of help that you would not expect the course lecturer to give; in particular do not share code. It is an offence to submit somebody elese's work under your own name, and it is also an offence to give your work to somebody else.

A.1 Introduction

The Caesar cypher is a particularly simple (i.e. not very effective) way of encrypting text. You can read about it at

`https://en.wikipedia.org/wiki/Caesar_cipher`

The reason that the cypher is insecure is that in most human languages, letter frequencies are not uniform. For instance, in English the letter E is often the most common and the letters Q and Z are quite rare. In addition, in English the letter Q is invariably followed by the letter U.

In this assignment you will perform letter frequency analysis of a piece of text to test these claims. Your program could also be used to decrypt text with an unknown Caesar shift by looking for th emost common letter and assumming that is an E.

A.2 Preparing the input data [10 marks]

◇ Create a directory `ass1` in your `javaProgramming` directory and begin editing a program file containing the class `jc.txt`

```
teaching$ cd javaProgramming
teaching$ mkdir ass1
teaching$ cd ass1
teaching$ kate jc.txt &
```

Of course, you should use whatever kind of editor pleases you.

◇ Copy and paste the contents of page 10 if the course book *Java, Concisely* into your `jc.txt` file.

Copy and paste the above column of figures into it.

◇ Add your own name to this file, inserting it on a line of its own at the end.

A.3 Reading the file [30 marks]

◇ Begin editing a Java programme called `Ass1.java`

```
teaching$ kate Ass1.java &
```

◇ Create a suitable class with `main` method.

◇ Inside the `main` method, add code to open the text file `jc.txt` and read in each line, keeping a count of the number of lines as you go. When the end of the file is reached, print out a message of the form

```
jc.txt contains x lines
```

where `x` is the number of lines in the file.

- ◇ Manually verify that the number of lines read in is indeed the number of line in the file.

A.4 Counting letter frequencies [30 marks]

We are now going to read through the file, counting letter frequencies. Note that we are treating lowercase characters such as `a` as equivalent to their uppercase counterparts (`A`, in this case).

- ◇ Create an array of integers called `counters` with 27 positions, one for each letter of the alphabet, plus an extra slot for non-alphabetic characters.

  We shall use the first slot to store the count of the number of `a` characters, the next to count the number of `b` characters and so on. The last slot will hold the number of non-alphabetic characters.

- ◇ Read the lines of the file, and for each line iterate over its characters incrementing the appropriate element of the `counters` array for each character.

- ◇ After reading the whole file, iterate over the array and for each element print out the letter it represents and the count within the file. In my test solution, there were 2 Q characters, four X characters and no Z characters. Your results might differ, depending on your name.

A.5 Visualising the output [30 marks]

Produce a diagram showing a histogram of the output. There are a variety of ways you might do this: you could print one out on the screen by using a column of * characters for each slot. You could output the data in Excel's CSV format and get Excel to plot the histogram. You could use one of the Java graphics subsystems such as Swing or JavaFX to directly display the result on the screen. Be creative.

B Assignment two

Please see the 'Coursework requirements and deadlines' section of our website for submission deadlines, and for the contribution to your final course mark.

Assessment criteria and feedback

The parts of this assignment are not all of the same weight: the marks for each question are shown below. Some marks will be given for a correct approach even when the overall solution is incomplete or incorrect.

This assignment will be marked by the course lecturer and feedback given; please note that the feedback grades are provisional and subject to moderation by our external examiners during the examination process.

The primary purpose of this assignment is *formative*: it provides practice in basic Java programming and gives you an opportunity to assess for yourself the degree to which you have understood the material

You should try all parts and hand in what you have done, even if you are not able to complete some of the questions. Solutions to the assignment will be discussed in a lecture slot after the assignments have been marked.

If you do not know how to do part of the assignment, begin by reading the corresponding text in the notes and working through related examples discussed in lectures. Please feel free to make an appointment to talk to me about anything you don't understand, and remember solutions *must be the submitter's own work*.

If someone asks you for help you may discuss the material and examples given in class but you should not give any form of help that you would not expect the course lecturer to give; in particular do not share code. It is an offence to submit somebody else's work under your own name, and it is also an offence to give your work to somebody else.

B.1 Hunting hedgehogs

In 1994, one of us was approached by a world authority on hedgehogs to help write a program for his student labs. The idea was to find out the *range* hedgehogs cover in the course of a night. The experiment involved releasing a hedgehog into the wild after it had been fitted with a small radio transmitter. Readings of the hedgehog's position were taken at regular intervals as coordinate pairs. Back in the lab, the students plotted the points on graph paper, and then drew the enclosing convex polygon. The area of the polygon was taken as the range for that particular hedgehog.

In computational geometry this problem is called the *Convex Hull* problem, and there are a variety of algorithms available to solve it. We are going to introduce you to the simplest one, which takes time cubic in the number of observations to run. We call it an $O(n^3)$ solution.

In Parts 1-3 we are concentrating on reading the list of coordinate pairs into the computer. In Part 4 you will calculate the convex hull of the points.

B.2 Setting up

Create a directory `ass2` in your `javaProgramming` directory and begin editing a program file containing the class `Ass2`

```
teaching$ cd javaProgramming
teaching$ mkdir ass2
teaching$ cd ass2
teaching$ emacs Ass2.java &
```

B.3 Preparing the data [10 marks]

Edit a new Java source file which will hold class `ConvexHull`.

```
teaching$ emacs ConvexHull.java &
```

In the class `ConvexHull` write a method `loadPoints()` which takes two arrays `xVal` and `yVal` of `doubles` and an integer `maxPoints` and returns an integer, `numPoints`.

- ◇ `loadPoints` should read values from the input line alternately into `xVal` and `yVal` until a negative value is read or until greater than 2∗`maxPoints` are read. In the latter case, print out a message warning the user that the maximum capacity of the array has been reached.
- ◇ The method should return the number of points read into `yVal`.

Here is a skeleton starting point for the required class `ConvexHull.java`.

```
import java.util.Scanner;

class ConvexHull
```

```
{
    static int loadPoints(
            int maxPoints, double[] xVal, double[] yVal)
    {
      /* put your code in here */
    }
}
```

When you have finished, make sure that your class compiles without errors.

Test your `loadPoints` method by writing a `main` method in the class `ConvexHull`. The main method should

- ◇ Declare an integer variable `maxPoints` and initialise it to 70.
- ◇ Declare two arrays `xVal` and `yVal` of size `maxPoints` which contain `doubles`.
- ◇ Call `loadPoints(maxPoints, xVal, yVal)` and write the value it returns to an integer variable `pointCount` and then print out this value.
- ◇ Print out the points entered as pairs in the form $(xVal[i], yVal[i])$.

Compile and run your program and enter the values 1 5 6 8 3 -4

B.4 Finding a convex hull

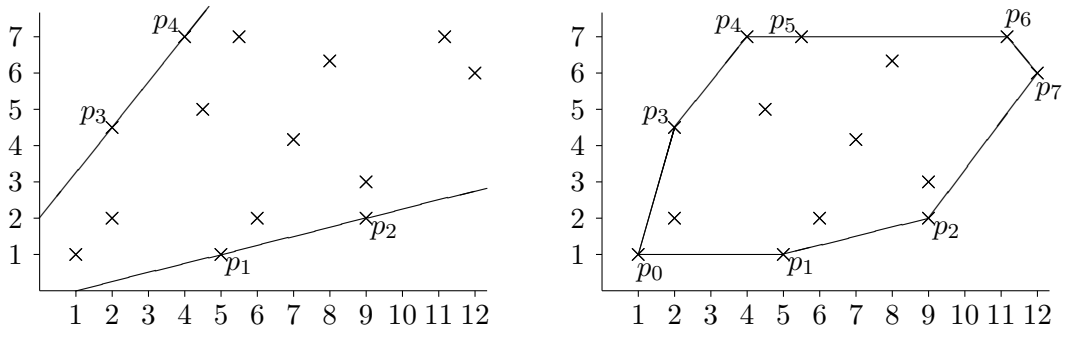
Recall that every pair, (p_1, p_2) of distinct points in the plane lies on a unique line, which has *slope* $m = \frac{(y_1 - y_2)}{(x_1 - x_2)}$.

If $p = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ then the equation of this line is either

1. $x = c$, where $c = x_1$, if $x_1 = x_2$, or
2. $y = mx + c$, where $c = y_1 - mx_1$, if $x_1 \neq x_2$.

For the purposes of this assignment, the Convex Hull of a set P of points in the plane is the set of edges (p_1, p_2) between pairs of points in P such that either none of the points in P lie above the line defined by p_1 and p_2, or none of the points lie below the line defined by p_1 and p_2.

For example, for the following set of points, no points lie below the line defined by p_1 and p_2, and no points lie above the line defined by p_3 and p_4.



For this example the convex hull is the set of edges

$$(p_0, p_3), (p_3, p_4), (p_4, p_5), (p_5, p_6), (p_4, p_6), (p_6, p_7), (p_2, p_7), (p_1, p_2), (p_0, p_1)$$

Note that by our definition of convex hull, *any* line segment on the hull is part of the hull. When we have three points in a line (such as p_4, p_5 and p_6) above, we expect three lines to be listed ($(p_4, p_5), (p_5, p_6), (p_4, p_6)$ in this case). If we wanted a minimal hull, we would like just the line (p_4, p_6) to be listed, but for this exercise we are content with the non-minimal listing.

To calculate whether a point $p = (a, b)$ lies above or below a line we use the equation of the line.

For the line $x = c$, p is above the line if $a > c$ and below the line if $a < c$.

For the line $y = mx + c$, if $b > ma + c$ then p is above the line and if $b < ma + c$ then p is below the line.

For this assignment you will write a method that computes the convex hull of any sequence of distinct points in the plane by looking at each pair of points and deciding whether or not there are points both above and below the line they define. We will extend the class `ConvexHull` that you wrote in Part 3 of this assignment, so open this file for editing.

```
teaching$ cd javaProgramming
teaching$ cd Ass1
teaching$ emacs ConvexHull.java &
```

B.5 The `checkDuplicates` method [20 marks]

It turns out that a polygon in which two points overlay each other (or equivalently, a polygon with a repeated point) can cause some nasty problems. For instance, we are going to find the gradient of a line from some (x_1, y_1) to (x_2, y_2). What would that mean if they have the same coordinates? Clearly, it's important to go through the data the user has given us, and check for duplicates.

Your class `ConvexHull` should already have a method `loadPoints` and a `main` method.

Add a method, `checkDuplicates` that tests to see whether the same point was read in twice by `loadPoints`. A skeleton for the method is given below.

The method should take as input an integer `pointCount` and two arrays of doubles, `xVal` and `yVal`, and it should return a boolean.

The method should check whether there is a pair of integers i, j, that lie between 0 and `pointCount`, such that $i \neq j$ and $xVal[i] = xVal[j]$ and $yVal[i] = yVal[j]$. As soon as it finds such a pair the method should print out an error message, return `true` and terminate. Otherwise, if there are no duplicates, the method should return `false`.

```
static boolean checkDuplicates(int pointCount, double xVal[], double yVal[])
{
   for (int i=pointCount; i>=1; i--) {
      for (int j=0; j<i; j++) {
```

```
            /* put your code in here */

          }
      }
      return false;
  }
```

Test your method by adding the following `if` statement to the end of your `main` method. (The statement terminates the program if a duplicate input entry is found.)

```
if ( checkDuplicates(pointCount, xVal, yVal) ) return;
```

Run the program with input which contains duplicates and with input which does not contain duplicates.

B.6 The `computeConvexHull` method [40 marks]

Write a method

```
static void computeConvexHull(int pointCount, double xVal[], double yVal[])
{
  double m, c;
  /* put your code here */
}
```

which begins by declaring two variables `m` and `c` of type `double`.

1. The method should then consider each pair of points $p_i = (xVal[i], yVal[i])$, $p_j = (xVal[j], yVal[j])$, for $0 \leq i < j <$ `pointCount`, using two nested `for` statements like those used in your `checkDuplicates` method. For each pair, p_i, p_j, the method should proceed as follows.

2. Declare two variables `above` and `below` of type `int`, both of which are initialised to 0.

3. Set `m` to be the slope of the line defined by p_i and p_j.

4. The method should consider two possibilities:

   (a) a near vertical line, for which the value of `m` is `Double.POSITIVE_INFINITY` or `Double.NEGATIVE_INFINITY`; and

   (b) other lines for which `m` a conventional finite numeric value.

5. In both cases the method should test all the points $p_k = (xVal[k], yVal[k])$, for $0 \leq k <$ `pointCount` and increase `above` by 1 if p_k is above the line defined by p_i and p_j, and increase `below` by 1 if p_k is below the line defined by p_i and p_j.

6. When all the points p_k have been considered, if only one of `above` and `below` is greater than 0 then the edge (p_i, p_j) is on the convex hull and a message reporting this should be printed out.

Now add a final line to your `main` method which calls the `computeConvexHull` method as follows.

```
computeConvexHull(pointCount, xVal, yVal);
```

Compile, run and test your program.

B.7 Visualisation [30 marks]

Using Java FX, draw a map of the hedgehog's path and the range, as defined by the convex hull of the path.

C Concise topic list

This is a short summary of the main topics covered within Java Concisely, as an aid to term 1 revision. The list is not comprehensive, but might form a useful basis upon which you can construct your own, more comprehensive list. Start by locating the page references for these topics. Hint: use an electronic copy of the book and the search facilities of your reader.

1. Declaration of variables with simple types: int, double, boolean, char
2. Declaration of variables with object types: String and classes declared by you.
3. Expressions: assignment, basic arithmetic operators, relational operators, logic operators, if-then-else operator.
4. Expression subtleties: side effect of `++` and `--` operators; short circuit behaviour of some logical operators.
5. Use of `int` and `double` data types (use `int` where you can, and `double` whenever you need a real number such as the mean of a sequence of data); correct use of casts to ensure real number arithmetic where needed.
6. Control flow: sequence, if-then-else, switch, while-do; do-while; for; for over collections; break; continue; call and return; recursion.
7. Class definition: variable members; static class variables; method definition; signatures; constructors; overloading; extending classes; overriding; super and this.
8. Structures: arrays of primitive types; arrays of object types; array initialisation; multi-dimensional arrays (arrays of arrays); dynamic data structures: the null value; lists; trees: preorder, inorder and postorder traversal.
9. Printing to the console and a file; reading from the console and a file using a scanner; formated output (printf); simple JavaFX display.
10. Algorithms: sieve of Eratosthenes; recursive Fibonacci sequence; complex to simple polygon conversion, bubble sort.