CHAPTER: 02

Building blocks

Objectives:

By the end of this chapter you should be able to:

- *distinguish between the eight built-in **primitive types** of Java;*
- ***declare** and **assign** values to **variables**;*
- *create **constant** values with the keyword* `final`*;*
- *use the input methods of the* `Scanner` *class to get data from the keyboard;*
- *design the functionality of a method using **pseudocode**.*

2.1 Introduction

The "Hello world" program that we developed in chapter 1 is of course very simple indeed. One way in which this program is very limited is that it has no *data* to work on. All interesting programs will have to store data in order to give interesting results; what use would a calculator be without the numbers the user types in to add and multiply? For this reason, one of the first questions you should ask when learning any programming language is “what types of data does this language allow me to store in my programs?”

2.2 Simple data types in Java

We begin this topic by taking a look at the basic types available in the Java language. The types of value used within a program are referred to as **data types**. If you wish to record the *price* of a cinema ticket in a program, for example, this value would probably need to be kept in the form of a **real number** (a number with a decimal point in it). However, if you wished to record *how many* tickets have been sold you would probably need to keep this in the form of an **integer** (whole number). It is necessary to know whether suitable types exist in the programming language to keep these bits of data.

In Java there are a few simple data types that programmers can use. These simple types are often referred to as the **primitive types** of Java; they are also referred to as the **scalar types**, as they relate to a single piece of information (a single real number, a single character etc).

Table 2.1 lists the names of these types in the Java language, the kinds of value they represent, and the exact range of these values.

Table 2.1 The primitive types of Java		
Java type	**Allows for**	**Range of values**
`byte`	very small integers	–128 to 127
`short`	small integers	–32 768 to 32 767
`int`	big integers	–2 147 483 648 to 2 147 483 647
`long`	very big integers	–9 223 372 036 854 775 808 to 9 223 372 036 854 775 807
`float`	real numbers	+/– $1.4 * 10^{-45}$ to $3.4 * 10^{38}$
`double`	very big real numbers	+/– $4.9 * 10^{-324}$ to $1.8 * 10^{308}$
`char`	characters	Unicode character set
`boolean`	true or false	not applicable

As you can see, some kinds of data, namely integers and real numbers, can be kept as more than one Java type. For example, you can use the **`byte`** type, the **`short`** type or the **`int`** type to hold integers in Java. However, while each numeric Java type allows for both positive and negative numbers*, the maximum size of numbers that can be stored varies from type to type*.

For example, the type **`byte`** can represent integers ranging only from –128 to 127, whereas the type **`short`** can represent integers ranging from –32 768 to 32 767. Unlike some programming languages, these ranges are *fixed* no matter which Java compiler or operating system you are using.

The character type, **`char`**, is used to represent characters from a standard set of characters known as the **Unicode** character set. This contains nearly all the characters from most known languages. For the sake of simplicity, you can think of this type as representing any character that can be input from your keyboard.

Finally, the **`boolean`** type is used to keep only one of two possible values: **`true`** or **`false`**. This type can be useful when creating tests in programs. For example, the answer to the question "have I passed my exam?" will either be either *yes* or *no*. In Java a **`boolean`** type could be used to keep the answer to this question, with the value **`true`** being used to represent *yes* and the value **`false`** to represent *no*.

2.3 Declaring variables in Java

The data types listed in table 2.1 are used in programs to create named locations in the computer's memory that will contain values while a program is running. This process is known as **declaring**. These named locations are called **variables** because their values are allowed to *vary* over the life of the program.

For example, a program written to develop a computer game might need a piece of data to record the player's score as secret keys are found in a haunted house. The value held in this piece of data will vary as more keys are found. This piece of data would be referred to as a variable. To create a variable in your program you must:

- give that variable a name (of your choice);
- decide which data type in the language best reflects the kind of values you wish to store in the variable.

What name might you choose to record the score of the player in our computer game?

The rules for naming *variables* are the same as those we met when discussing the rules for naming *classes* in the previous chapter. However, the convention in Java programs is to begin the name of a variable with a *lower case* letter (whereas the convention is to start class names with an upper case letter). We could just pick a name like x, but it is best to pick a name that describes the purpose of the item of data; an ideal name would be `score`.

Which data type in table 2.1 should you use if you wish to record a player's score? Well, since the score would always be a whole number, an integer type would be appropriate. There are four Java data types that can be used to hold integers (**`byte`**, **`short`**, **`int`** and **`long`**). As we said before, the only difference among these types is the range of values that they can keep. Unless there is specific reason to do otherwise, however, the **`int`** type is normally chosen to store integer values in Java programs. Similarly, when it comes to storing real numbers we will choose the **`double`** type rather than the **`float`** type.

Once the name and the type have been decided upon, the variable is **declared** as follows:

```
dataType variableName;
```

where `dataType` is the chosen primitive type and `variableName` is the chosen name of the variable. So, in the case of a player's score, the variable would be declared as follows:

```
int score;
```

Figure 2.1 illustrates the effect of this instruction on the computer's memory. As you can see, a small part of the computer's memory is set aside to store this item. You can think of this reserved space in memory as being a small box, big enough to hold an integer. The name of the box will be `score`.

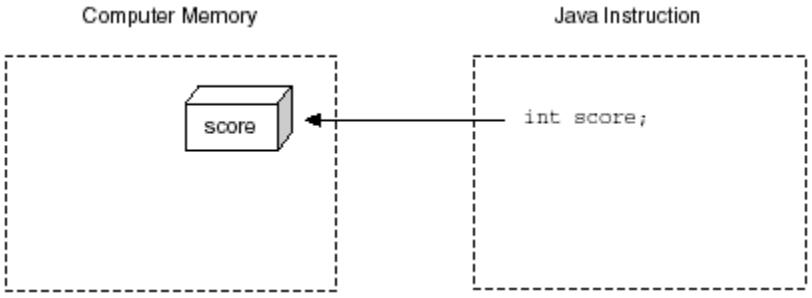


Fig 2.1. The effect of declaring a variable in Java

In this way, many variables can be declared in your programs. Let's assume that the player of a game can choose a difficulty level (A, B, or C); another variable could be declared in a similar way.

What name might you give this variable? An obvious choice would be *difficulty level* but remember names cannot have spaces in them. You could use an underscore to remove the space (*difficulty_level*) or start the second word with a capital letter to distinguish the two words (*difficultyLevel*). Both are well-established naming conventions in Java. Alternatively you could just shorten the name to, say, *level*; that is what we will do here.

Now, what data type in table 2.1 best represents the difficulty level? Since the levels are given as characters (A, B and C) the **`char`** type would be the obvious choice. At this point we have two variables declared: one to record the score and one to record the difficulty level.

```
int score;
char level;
```

Finally, several variables can be declared on a *single line* if they are *all of the same type*. For example, let's assume that there are ghosts in the house that hit out at the player; the number of times a player gets hit by a ghost can also be recorded. We can call this variable *hits*. Since the type of this variable is also an integer, it can be declared along with `score` in a single line as follows:

```
int score, hits; // two variables declared at once
char level ; // this has to be declared separately
```

Figure 2.2 illustrates the effect of these three declarations on the computer's memory.

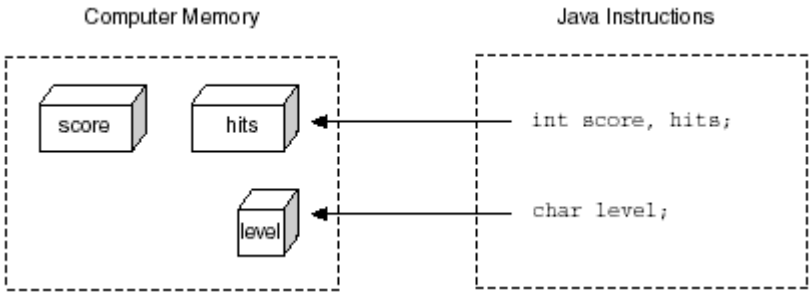


Fig 2.2. The effect of declaring many variables in Java

Notice that the character box, `level`, is half the size of the integer boxes `score` and `hits`. That is because, in Java, the **`char`** type requires half the space of the **`int`** type. You should also be aware that the **`double`** type in Java requires twice the space of the **`int`** type.

You're probably wondering: if declaring a variable is like creating a box in memory, how do I put values into this box? The answer is with assignments.

2.4 Assignments in Java

Assignments allow values to be put into variables. They are written in Java with the use of the equality symbol (=). In Java this symbol is known as the **assignment operator**. Simple assignments take the following form:

```
variableName = value;
```

For example, to put the value zero into the variable `score`, the following assignment statement could be used:

```
score = 0;
```

This is to be read as "*set* the value of `score` *to* zero" or alternatively as "`score` *becomes equal to* zero". Effectively, this puts the number zero into the box in memory we called `score`. If you wish, you may combine the assignment statement with a variable declaration to put an initial value into a variable as follows:

```
int score = 0;
```

This is equivalent to the two statements below:

```
int score;
score = 0;
```

Although in some circumstances Java will automatically put initial values into variables when they are declared, this is not always the case and it is better explicitly to initialize variables that require an initial value.

Notice that the following declaration will not compile in Java:

```
int score = 2.5 ;
```

Can you think why?

The reason is that the right-hand side of the assignment (2.5) is a *real* number. This value could not be placed into a variable such as `score`, which is declared to hold only integers, without some information loss. In Java, such information loss is not permitted, and this statement would therefore cause a compiler error.

You may be wondering if it is possible to place a whole number into a variable declared to hold real numbers. The answer is yes. The following is perfectly legal:

```
double someNumber = 1000;
```

Although the value on the right-hand side (1000) appears to be an integer, it can be placed into a

variable of type **double** because this would result in no information loss. Once this number is put into the variable of type **double**, it will be treated as the real number 1000.0.

Clearly, you need to think carefully about the best data type to choose for a particular variable. For instance, if a variable is going to be used to hold whole numbers *or* real numbers, use the **double** type as it can cope with both. If the variable is only ever going to be used to hold whole numbers, however, then although the **double** type might be adequate, use the **int** type as it is specifically designed to hold whole numbers.

When assigning a value to a character variable, you must enclose the value in single quotes. For example, to set the initial difficulty level to A, the following assignment statement could be used:

```
char level = 'A';
```

Remember: you need to declare a variable only once. You can then assign values to it as many times as you like. For example, later on in the program the difficulty level might be changed to a different value as follows:

```
char level = 'A'; // initial difficulty level
// other Java instructions
level = 'B'; // difficulty level changed
```

2.5 Creating constants

There will be occasions where data items in a program have values *that do not change*. The following are examples of such items:

- the maximum score in an exam (100);
- the number of hours in a day (24);
- the mathematical value of π (approximately 3.1416).

In these cases the values of the items do not vary. Values that remain constant throughout a program (as opposed to variable) should be named and declared as **constants**.

Constants are declared much like variables in Java except that they are preceded by the keyword **final**. Once they are given a value, then that value is fixed and cannot later be changed. Normally we fix a value when we initialize the constant. For example:

```
final int HOURS = 24;
```

Notice that the standard Java convention has been used here of naming constants in upper case. Any attempt to change this value later in the program will result in a compiler error. For example:

```
final int HOURS = 24; // create constant
HOURS = 12; // will not compile!
```

2.6 Arithmetic operators

Rather than just assign simple values (such as 24 and 2.5) to variables, it is often useful to carry out some kind of arithmetic in assignment statements. Java has the four familiar arithmetic operators, plus a remainder operator, for this purpose. These operators are listed in table 2.2.

Table 2.2 The arithmetic operators of Java	
Operation	**Java operator**
addition	+
subtraction	–
multiplication	*
division	/
remainder	%

You can use these operators in assignment statements, much like you might use a calculator. For example, consider the following instructions:

```
int x;
x = 10 + 25;
```

After these instructions the variable `x` would contain the value 35: the result of adding 10 to 25. Terms on the right-hand side of assignment operators (like 10 + 25) that have to be *worked out* before they are assigned are referred to as **expressions**. These expressions can involve more than one operator.

Let's consider a calculation to work out the price of a product after a sales tax has been added. If the initial price of the product is 500 and the rate of sales tax is 17.5%, the following calculation could be used to calculate the total cost of the product:

```
double cost;
cost = 500 * (1 + 17.5/100);
```

After this calculation the final cost of the product would be 587.5.

By the way, in case you are wondering, the order in which expressions such as these are evaluated is the same as in arithmetic: terms in brackets are calculated first, followed by division, then multiplication, then addition and finally subtraction. This means that the term in the bracket

```
(1 + 17.5/100)
```

evaluates to 1.175, not 0.185, as the division is calculated before the addition. The final operator (%) in table 2.2 returns the remainder after *integer division* (this is often referred to as the **modulus**).Table 2.3 illustrates some examples of the use of this operator together with the values returned.

Table 2.3 Examples of the modulus operator in Java	
Expression	**Value**
29 % 9	2
6 % 8	6
40 % 40	0
10 % 2	0

As an illustration of the use of both the division operator and the modulus operator, consider the following example:

A large party of 30 people is going to attend a school reunion. The function room will be furnished with a number of tables, each of which seats four people.

To calculate how many tables of four are required, and how many people will be left over, the division and modulus operators could be used as follows:

```
int tablesOfFour, peopleLeftOver;
tablesOfFour = 30/4; // number of tables
peopleLeftOver = 30%4; // number of people left over
```

After these instructions the value of `tablesOfFour` will be 7 (the result of dividing 30 by 4) and the value of `peopleLeftOver` will be 2 (the remainder after dividing 30 by 4). You may be wondering why the calculation for `tablesOfFour`

```
30/4
```

did not yield 7.5 but 7. The reason for this is that there are, in fact, two different in-built division routines in Java, one to calculate an integer answer and another to calculate the answer as a real number.

Rather than having two division operators, however, Java has a single division symbol (/) to represent *both* types of division. The division operator is said to be **overloaded**. This means that the same operator (in this case the division symbol) can behave in different ways. This makes life much easier for programmers as the decision about which routine to call is left to the Java language.

How does the Java compiler know which division routine we mean? Well, it looks at the values that are being divided. If *at least one value* is a real number (as in the product cost example), it assumes we mean the division routine that calculates an answer as a real number, otherwise it assumes we mean the division routine that calculates an answer as a whole number (as in the reunion example).[1]

[1] To force the use of one division routine over another, a technique known as **type casting** can be used. We will return to this technique in later chapters.

2.7 Expressions in Java

So far, variable names have appeared only on the left-hand side of assignment statements. However, the expression on the right-hand side of an assignment statement can itself contain variable names. If this is the case then the name does not refer to *the location*, but to *the contents of the location*. For example, the assignment to calculate the cost of the product could have been re-written as follows:

```
double price, tax, cost; // declare three variables
price = 500; // set price
tax = 17.5; // set tax rate
cost = price * (1 + tax/100); /// calculate cost
```

Here, the variables `price` and `tax` that appear in the expression

```
price * (1 + tax/100)
```

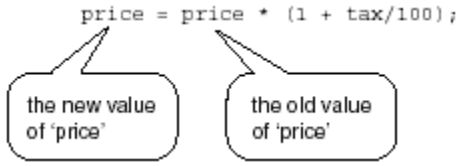
are taken to mean *the values contained in* `price` and `tax` respectively. This expression evaluates to 587.5 as before. Notice that although this price happens to be a whole number, it has been declared to be a **`double`** as generally prices are expressed as real numbers.

There is actually nothing to stop you using the name of the variable you are assigning to in the expression itself. This would just mean that the old value of the variable is being used to calculate its new value. Rather than creating a new variable, `cost`, to store the final cost of the product, the calculation could, for example, have updated *the original price* as follows:

```
price = price * (1 + tax/100);
```

Now, only two variables are required, `price` and `tax`. Let's look at this assignment a bit more closely.

When reading this instruction, the `price` in the right-hand expression is to be read as the *old value* of `price`, whereas the `price` on the left-hand side is to be read as *the new value* of `price`.



You might be wondering what would happen if we used a variable in the right hand side of an expression before it had been given a value. For example, look at this fragment of code:

```
double price = 500;
double tax;
cost = price * (1 + tax/100);
```

The answer is that you would get a compiler error telling you that you were trying to use a variable before it has been initialized.

You will find that one very common thing that we have to do in our programs is to increase (or increment) an integer by 1. For example, if a variable `x` has been declared as an **`int`**, then the instruction for incrementing `x` would be:

```
x = x + 1;
```

In fact, this is so common that there is a special shorthand for this instruction, namely:

```
x++;
```

The '++' is therefore known as the increment operator. Similarly there exists a decrement operator, '– –'.Thus:

```
x– –;
```

is shorthand for:

```
x = x – 1;
```

It is possible to use the increment and decrement operators in expressions. We will show you a couple of examples of this here, as you might easily come across them in other texts. However, we will not be using this technique in the remainder of this book, because we think it can sometimes be confusing for new programmers. If `x` and `y` are **`int`**s, the expression:

```
y = x++;
```

means assign the value of `x` to `y`, then increment x by 1.

However the expression:

```
y = ++x;
```

means increment `x` by 1, then assign this new value to `y`. The decrement operator can be used in the same way.

While we are on the subject of shortcuts, there is one more that you might come across in other places, but which, once again, we won't be using in this text:

```
y += x;
```

is shorthand for:

```
y = y + x;
```

The code fragments we have been writing so far in this chapter are, of course, not complete programs. As you already know, to create a program in Java you must write one or more classes. In program 2.1, we write a class, `FindCost`, where the `main` method calculates the price of the product.

Program 2.1

```
/* a program to calculate the cost of a product after a sales tax has been added */

public class FindCost
{
   public static void main(String[] args)
   {
       double price, tax;
       price = 500;
       tax = 17.5;
       price = price * (1 + tax/100);
   }
}
```

What would you see when you run this program? The answer is nothing! There is no instruction to display the result on to the screen. You have already seen how to display messages onto the screen. It is now time to take a closer look at the output command to see how you can also display results onto the screen.

2.8 More about output

As well as displaying messages, Java also allows any values or expressions of the primitive types that we showed you in table 2.1 to be printed on the screen using the same output commands. It does this by implicitly converting each value/expression to a string before displaying it on the screen. In this way numbers, the value of variables, or the value of expressions can be displayed on the screen. For example, the square of 10 can be displayed on the screen as follows:

```
System.out.print(10*10);
```

This instruction prints the number 100 on the screen. Since these values are converted into strings by Java they can be joined on to literal strings for output.

For example, let's return back to the party of 30 people attending their school reunion that we discussed in section 2.6. If each person is charged a fee of 7.50 for the evening, the total cost to the whole party could be displayed as follows:

```
System.out.print("cost = " + (30*7.5) );
```

Here the concatenation operator (+), is being used to join the string, `"cost = "`, on to the value of the expression, (`30*7.5`). Notice that when expressions like `30*7.5` are used in output statements it is best to enclose them in brackets. This would result in the following output:

```
cost = 225.0
```

Bear these ideas in mind and look at program 2.2, where we have re-written program 1.2 so that the output is visible.

Program 2.2

```
// a program to calculate and display the cost of a product after sales tax has been added

public class FindCost2
{
   public static void main(String[] args)
   {
      double price, tax;
      price = 500;
      tax = 17.5;
      price = price * (1 + tax/100); // calculate cost
      // display results
      System.out.println("*** Product Price Check ***");
      System.out.println("Cost after tax = " + price);
   }
}
```

This program produces the following output:

```
*** Product Price Check ***

Cost after tax = 587.5
```

Although being able to see the result of the calculation is a definite improvement, this program is still very limited. The formatting of the output can certainly be improved, but we shall not deal with such issues until later on in the book. What does concern us now is that this program can only calculate the cost of products when the sales tax rate is 17.5% and the initial price is 500!

What is required is not to fix the rate of sales tax or the price of the product but, instead, to get the *user of your program* to *input* these values as the program runs.

2.9 Input in Java: the *Scanner* class

Java provides a special class called `Scanner`, which makes it easy for us to write a program that obtains information that is typed in at the keyboard. `Scanner` is provided as part of what is known, in Java, as a **package**. A package is a collection of pre-compiled classes – lots more about that in the second semester! The `Scanner` class is part of a package called `util`. In order to access a package we use a command called **`import`**. So, to make the `Scanner` class accessible to the compiler we have to tell it to look in the `util` package, and we do this by placing the following line at the top of our program:

```
import java.util.*;
```

This asterisk means that all the classes in the particular package are made available to the compiler.

As long as the `Scanner` class is accessible, you can use all the input methods that have been defined in this class. We are going to show you how to do this now. Some of the code might look a bit mysterious to you at the moment, but don't worry about this right now. Just follow our instructions for the time being – after a few chapters, it will become clear to you exactly why we use the particular format and syntax that we are showing you.

Having imported the `util` package, you will need to write the following instruction in your program:

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

What we are doing here is creating an object, `keyboard`, of the `Scanner` class. Once again, the true meaning of the phrase *creating an object* will become clear in the next few chapters, so don't worry too much about it now. However, you should know that, in Java, `System.in` represents the keyboard, and by associating our `Scanner` object with `System.in`, we are telling it to get the input from the keyboard as opposed to a file on disk or a modem for example.

The `Scanner` class has several input methods, each one associated with a different input type, and once we have declared a `Scanner` object we can use these methods. Let's take some examples. Say we wanted a user to type in an integer at the keyboard, and we wanted this value to be assigned to an integer variable called `x`. We would use the `Scanner` method called `nextInt`; the instruction would look like this:

```
x = keyboard.nextInt();
```

In the case of a **`double`**, `y`, we would do this:

```
y = keyboard.nextDouble();
```

Notice that to access a method of a class you need to join the name of the method (`getInt` or `getDouble`) to the name of the object (`keyboard`) by using the full-stop. Also you must remember the brackets after the name of the method.

What about a character? Unfortunately this is a little bit more complicated, as there is no `nextChar` method provided. Assuming `c` had been declared as a character, we would have to do this:

```
c = keyboard.next().charAt(0);
```

You won't understand exactly why we use this format until chapter 7 – for now just accept it and use it when you need to.

Let us return to the haunted house game to illustrate this. Rather than *assigning* a difficulty level as follows:

```
char level;
level = 'A';
```

you could take a more flexible approach by asking the user of your program to *input* a difficulty level while the program runs. Since `level` is declared to be a character variable, then, after declaring a `Scanner` object, `keyboard`, you could write this line of code:

```
level = keyboard.next().charAt(0);
```

Some of you might be wondering how we would get the user to type in strings such as a name or an address. This is a bit more difficult, because a string is not a simple data types like an **`int`** or a **`char`**, but contains many characters. In Java a `String` is not a simple data type but a class - so to do this you will have to wait until chapter 7 where we will study classes and objects in depth.

Let us re-write program 2.2 so that the price of the product and the rate of sales tax are not fixed in the program, but are input from the keyboard. Since the type used to store the price and the tax is a **`double`**, the appropriate input method is `nextDouble`, as can be seen in program 2.3.

Program 2.3

```
import java.util.*; // in order to access the Scanner class

/* a program to input the initial price of a product and then calculate and display its cost after tax
has been added */

public class FindCost3
{
   public static void main(String[] args )
   {
      Scanner keyboard = new Scanner(System.in); // create Scanner object
      double price, tax;
      System.out.println("*** Product Price Check ***");
      System.out.print("Enter initial price: "); // prompt for input
      price = keyboard.nextDouble(); // input method called
      System.out.print("Enter tax rate: "); // prompt for input
      tax = keyboard.nextDouble(); // input method called
      price = price * (1 + tax/100); // perform the calculation
      System.out.println("Cost after tax = " + price);
   }
}
```

Note that, by looking at this program code alone, there is no way to determine what the final price of the product will be, as the initial price and the tax rate will be determined *only when the program is run*.

Let's assume that we run the program and the user interacts with it as follows[2]:

```
*** Product Price Check ***

Enter initial price: 1000

Enter tax rate: 12.5

Cost after tax = 1125.0
```

You should notice the following points from this test run:

- whatever the price of the computer product and the rate of tax, this program could have evaluated the final price;
- entering numeric values with additional formatting information, such as currency symbols or

[2] We have used ***bold italic*** font to represent user input

the percentage symbol, is not permitted;

- after an input method is called, the cursor always moves to the next line.

The programs we are looking at now involve input commands, output commands and assignments. Clearly, the order in which you write these instructions affects the results of your programs. For example, if the instructions to calculate the final price and then display the results were reversed as follows:

```
System.out.println("Cost after tax = " + price);
price = price * (1 + tax/100);
```

The price that would be displayed would not be the price *after* tax but the price *before* tax! In order to avoid such mistakes it makes sense *to design your code* by sketching out your instructions before you type them in.

2.10 Program design

Designing a program is the task of considering exactly *how to build* the software, whereas writing the code (the task of *actually building* the software) is referred to as *implementation*. As programs get more complex, it is important to spend time on program design, before launching into program implementation.

As we have already said, Java programs consist of one or more classes, each with one or more methods. In later chapters we will introduce you to the use of diagrams to help design such classes. The programs we have considered so far, however, have only a single class and a single method (`main`), so a class diagram would not be very useful here! We will therefore return to this design technique as we develop larger programs involving many classes.

At a lower level, it is the instructions *within* a method that determine the *behaviour* of that method. If the behaviour of a method is complex, then it will also be worthwhile spending time on designing the instructions that make up the method. When you sketch out the code for your methods, you don't want to have to worry about the finer details of the Java compiler such as declaring variables, adding semi-colons and using the right brackets. Very often a general purpose "coding language" can be used for this purpose to convey the meaning of each instruction without worrying too much about a specific language syntax.

Code expressed in this way is often referred to as **pseudocode**. The following is an example of pseudocode that could have been developed for the `main` method of program 2.3:

```
BEGIN
  DISPLAY program title
  DISPLAY prompt for price
  ENTER price
  DISPLAY prompt for tax
  ENTER tax
  SET price TO price * (1 + tax/100)
  DISPLAY new price
END
```

Note that these pseudocode instructions are not intended to be typed in and compiled as they do not meet the syntax rules of any particular programming language. So, exactly how you write

these instructions is up to you: there is no fixed syntax for them. However, each instruction conveys a well-understood programming concept and can easily be translated into a given programming language. Reading these instructions you should be able to see how each line would be coded in Java.

Wouldn't it be much easier to write your `main` method if you have pseudocode like this to follow? In future, when we present complex methods to you we will do so by presenting their logic using pseudocode.

Self-test questions

1 What would be the most appropriate Java data type to use for the following items of data?

- the maximum number of people allowed on a bus;
- the weight of a food item purchased in a supermarket;
- the grade awarded to a student (for example ‘A’, ‘B’ or ‘C’).

2 Explain which, if any, of the following lines would result in a compiler error:

```
int x = 75.5;
double y = 75;
```

3 Identify and correct the errors in the program below, which prompts for the user’s age and then attempts to work out the year in which the user was born.

```
import java.util.*;

public class SomeProg
{
  public static void main (String[] args)
  {
    Scanner keyboard = new Scanner(System.in);
    final int YEAR;
    int age, bornIn;
    System.out.print(How old are you this year? );
    age = keyboard.nextDouble();
    bornIn = YEAR – age;
    System.out.println("I think you were born in " + BornIn);
  }
}
```

4 What would be the final output from the program below if the user entered the number 10?

```
import java.util.*;

public class Calculate
{
  public static void main(String[] args )
  {
    Scanner keyboard = new Scanner(System.in);
    int num1, num2;
    num2 = 6;
    System.out.print(„Enter value „);
    num1 = keyboard.nextInt();
    num1 = num1 + 2;
    num2 = num1 / num2;
    System.out.println(“result = “ + num2);
  }
}
```

5 Use pseudocode to design a program that asks the user to enter values for the length and height of a rectangle and then displays the area and perimeter of that rectangle.

6. The program below was written in an attempt to swap the value of two variables. However it does not give the desired result:

```
/* This program attempts to swap the value of two variables - it doesn't give the
desired result however! */

import java.util.*;

public class SwapAttempt
{

   public static void main(String[] args)
   {
      // declare variables
      int x, y;
      // enter values
      System.out.print("Enter value for x ");
      x = keyboard.nextInt();
      System.out.print("Enter value for y ");
      y = keyboard.nextInt();

      // code attempting to swap two variables
      x = y;
      y = x;

      //display results
      System.out.println("x = " + x);
      System.out.println("y = " + y);
   }
}
```

a) Can you see why the program doesn't do what we hoped?

b) What would be the actual output of the program?

c) How could you modify the program above so that the values of the two variables are swapped successfully?

Programming exercises

1 Implement the programs from self-test questions 3, 4 and 6 above in order to verify your answers to those questions.

2 Implement the rectangle program that you designed in self-test question 5.

3 Some while ago, the European Union decreed that all traders in the UK sell their goods by the kilo and not by the pound (1 kilo = 2.2 pounds). The following pseudocode has been arrived for a program that carries this conversion:

```
BEGIN
  PROMPT for value in pounds
  ENTER value in pounds
  SET value to old value ÷ 2.2
  DISPLAY value in kilos
END
```

Implement this program, remembering to declare any variables that are necessary.

4 A group of students has been told to get into teams of a specific size for their coursework. Design and implement a program that prompts for the number of students in the group and the size of the teams to be formed, and displays how many teams can be formed and how many students are left without a team.

5 Design and implement a program that asks the user to enter a value for the radius of a circle; then displays the area and circumference of the circle.

Note that the area is calculated by evaluating πr^2 and the circumference by evaluating $2\pi r$. You can take the value of π to be 3.1416 - and ideally you should declare this as a constant at the start of the program[3].

[3] Of course you will not be able to use the Greek letter π as a name for a variable or constant. You will need to give it a name like PI.