



# THE FIRST STEP

## LEARNING OBJECTIVES

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

- explain the meaning of the word **software**;
- describe the way in which software is produced in industry;
- explain the need for high-level programming languages;
- describe the way in which programs are compiled and run;
- distinguish between **compilation** and **interpretation**;
- explain how Java programs are compiled and run;
- write Java programs that display text on the screen.

### 1.1 *Introduction*

Like any student starting out on a first programming module, you will be itching to do just one thing – get started on your first program. We can well understand that, and you won't be disappointed, because you will be writing programs in this very first chapter. Designing and writing computer programs can be one of the most enjoyable and satisfying things you can do, although sometimes it can seem a little daunting at first because it is like nothing else you have ever done. But with a bit of perseverance you will not only start to get a real taste for it but you may well find yourself sitting up till two o'clock in the morning trying to solve a problem. And just when you have given up and you are dropping off to sleep the answer pops into your head and you are at the computer again until you notice it is getting light outside! So if this is happening to you, then don't worry – it's normal!

However, before you start writing programs you need some background; so the first part of this chapter is devoted to some fundamental issues that you need to get to grips with before you can understand what programming itself is all about. This chapter therefore starts off with

## 4 THE FIRST STEP

an explanation of some of the terms you will come across, particularly the notion of *software*. It goes on to give a little bit of background as to how software is produced in industry, and the way in which this has changed in recent times. This is followed by a general look at programming languages and the way that programs are written and developed. After this we discuss the Java language and the way in which developing Java programs differs from that of conventional programming languages. Once we have done all this, we will show you how to write and adapt your first program.

### 1.2 *Software*

A computer is not very useful unless we give it some instructions that tell it what to do. This set of instructions is called a **program**. The word **software** is the name given to a single program or a set of programs.

There are two main kinds of software. **Application software** is the name given to useful programs that a user might need; for example, word-processors, spreadsheets, accounts programs and so on. **System software** is the name given to special programs that help the computer to do its job; for example, operating systems such as UNIX or Windows (which help us to use the computer) and network software (which helps computers to communicate with each other).

### 1.3 *Developing software*

We will now take a brief look at how software is developed in industry. This has begun to change over the last past decade or so – until quite recently the method was for software developers to go through a number of phases and complete each of these before moving on to the next. It was then necessary to go back one, two or more phases to make corrections. The first phase consisted of **analysis and specification**, the process of determining what the system was required to do (analysis) and writing it down in a clear and unambiguous manner (specification). The next phase was **design**; this phase consisted of making decisions about how the system would be built in order to meet the specification. After this came **implementation**, at which point the design was turned into an actual program. This was followed by the **testing** phase. When testing was complete the system would be **installed** and a period of **operation and maintenance** followed, whereby the system was improved, and if necessary changed to meet changing requirements. This approach to software development (often called the **waterfall model**) is summarized in figure 1.1.

One problem with this approach was that it often meant that customers had to wait a very long time before they actually saw the product. Another problem was that in practice software produced in this way was actually very difficult to adapt to changing needs (consider, for example, the Y2K problem).

Nowadays it is therefore common to use a RAD approach; this stands for **rapid application development**. The RAD approach involves doing the activities described above a “little bit at a

time”; in this way we build **prototypes** of the product and the potential user can be actively involved in testing them out and re-building until we eventually end up with the best possible product in the most satisfactory time period. Several programming tools such as Visual Basic and JBuilder exist to facilitate this process. The RAD process is summarized in figure 1.2.

Crucial to the RAD approach has been the advent of **object-oriented** methods of developing software. This term, *object-oriented*, is going to play a very important role in this book; however, we are not going to explore its meaning just yet as there are a few other concepts that we need to understand first. Nonetheless, we will start to get an idea of its meaning later in this chapter, and we will really start to get to grips with it from chapter 4 onwards. But right now it is time to start exploring the way in which programs are actually written.

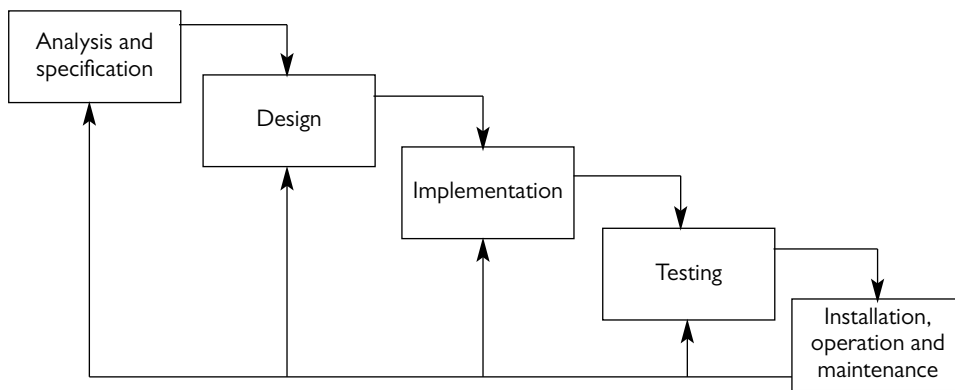


Fig 1.1 The traditional approach to software development (the waterfall model)

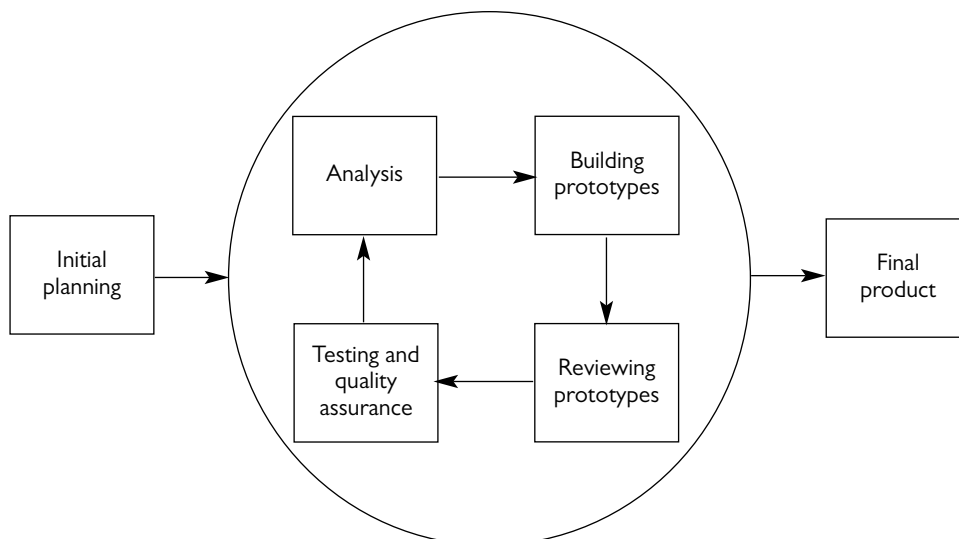


Fig. 1.2 The modern approach to software development (RAD)

### 1.3.1 Programming languages

A program is written by a developer in a special computer language; these include C++, Java, BASIC, Pascal and many more. It is then translated by a special program (a **compiler**) into something called **machine code**, or, in the case of Java, to **Java byte code**. Machine code and Java byte code consist of binary numbers (0s and 1s) which the computer can understand. For example, 01010011 might mean ADD whereas 01010100 could mean SUBTRACT.

It is of course very difficult to write even the shortest of programs in machine code (although this is what had to be done in the early days of computing) and it is for this reason that programming languages such as Java have been developed. The first, and most basic, of these languages was known as **assembly language**. In assembly language each one of the separate instructions that the computer understands is represented by a word that is a bit like an English word; for example MOV (for “move”), JMP (for “jump”), SUB (for “subtract”) and so on. A special piece of software (an **assembler**) is then used to turn what we have written into machine code. Assembly language is called a **low-level language**.

Most programs require fairly sophisticated instructions for control and data-manipulation, and writing such programs in assembly language is a very tedious business. It was for this reason that **high-level** languages – such as C, COBOL, Pascal and Basic – were created. Such languages (also known as **third generation languages** or **3GLs**) combine several basic computer instructions into a single statement or set of statements such as **if . . . else**, or **do . . . while**. So, when you write program instructions in these languages you write lines that are a bit like English. The set of instructions that you write in a programming language is called the **program code** or **source code**. In the case of high-level languages the translation of our source code into machine code is known as **compilation**, and the software that we use to perform this translation is called a **compiler**. The code that is produced by a compiler is often referred to as **object code**.<sup>1</sup>

Java, the language that you will be studying, is an **object-oriented programming language**; object-oriented languages represent the most recent development in the history of programming languages. As we have pointed out earlier, it will not be long before the meaning of this term starts to become clear.

At this stage it is worth mentioning that Java programs are compiled and run in a rather different way to programs written in other programming languages such as C, Pascal or C++ (which is another object-oriented language). In order to appreciate the difference, you need to understand how conventional compilers work, so we will take a brief look at this before exploring the Java development process.

## 1.4 Conventional compilers

The program code that we write must obey the rules of the programming language. These rules are known as the **syntax** (the grammatical structure) of the language. It is very common to

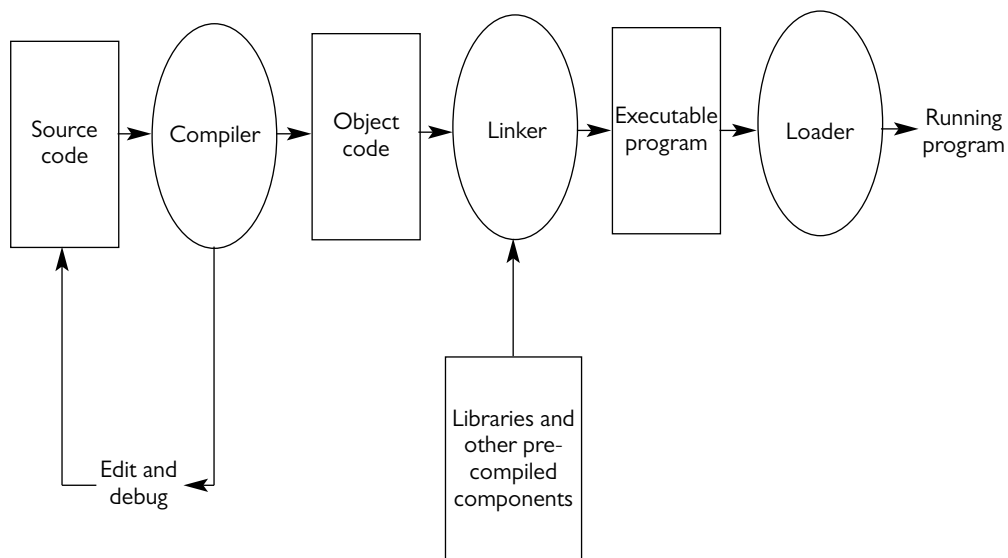
<sup>1</sup> Not to be confused with the word *object* that you will come across many times in this book, in connection with object-oriented programming.

make **syntax errors** when we write programs; programs with errors (or **bugs**) in them cannot be compiled. Therefore most compiler programs, when they try to compile what we have written, also tell us about the errors we have made and where they are in the program. So we go through a process of typing the code, trying to compile, correcting the errors (**debugging**) and trying to compile again.

These days, compilers do much more than just compile your programs. They usually provide what is called an **integrated development environment (IDE)**. This means that the one piece of software allows us to write and edit our programs, compile them into object code, and make changes in response to error messages that are generated by the compiler during compilation. In any major project, different bits of the program are developed separately and need to be integrated; this process is called **linking**, and an IDE will also perform this process. This also includes linking any necessary pre-compiled pieces of code which most IDEs provide (collections of pre-written pieces of code are known as **libraries**).

Once the program is compiled and linked it is saved as an **executable** file – that is, a file that can be loaded into the computer’s memory (by a special piece of system software called a **loader**) and run. Most IDEs will load and run the program for you without your having to exit from the IDE itself. The process of compiling and running programs in the conventional way is summarized in figure 1.3.

Compiling programs in this way is only one way of doing it – at one time it was more common to use another method of translating and running programs. This method is called **interpretation** and we will take a brief look at this now, because, as we shall see, it plays an important role in the Java development process.



**Fig 1.3 Compiling and running conventional programs**

## 1.5 *Interpreters*

At one time it was common to use an **interpreter** rather than a compiler. As we have seen, compilers operate on the entire program, providing a permanent binary file which may be **executed** (or **run**). An interpreter, on the other hand, translates and executes the program one line at a time. This of course is less efficient and execution is slower. It does have the advantage, however, of enabling a program to be changed frequently without going through the whole compilation process again. Until recently, interpreters were not used very much for real-life programs – but with the coming of Java the processes of compilation and interpretation have been cleverly combined to give us the best of both worlds.

## 1.6 *The Java development process*

As we have explained above, until recently all programs tended to be compiled and linked so that they formed a separate executable file. At first sight, this sounds like the ideal thing – compile a program so that you can take it away and run it on your computer or some other computer. But there is one major drawback with this process. A compiled program is suitable only for a particular type of computer. For example, a program that is compiled for a PC will not run on an Apple Mac or a UNIX machine. Again, until recently, this was not such a huge problem – but then along came the World Wide Web!

With the advent of the Web it became desirable for us to be able to download a program from a remote site and run it on our machine – and we want it to run in exactly the same way irrespective of whether our machine is a PC, an Apple Mac or any other machine. We need a language that is **platform-independent**.

Java is the language for this! It is designed to work within the World Wide Web of computers through special pieces of software called **browsers**, the most common of which are Netscape and Internet Explorer. Java programs that run on the Web are called **applets** (meaning little applications).

Inside a modern browser there is a special program, called a **Java Virtual Machine**, or **JVM** for short. This JVM is able to run a Java program for the particular computer on which it is running.

We saw earlier that conventional compilers translate our program code into machine code. This machine code would contain the particular instructions appropriate to the computer it was meant for. Java compilers do not translate the program into machine code – they translate it into special instructions called **Java byte code**. These instructions are then *interpreted* by the JVM for that particular machine. So whereas machine code is specific to a particular type of computer, Java byte code is universal.

So the process consists of typing in the program (source code), compiling it into byte code, correcting errors (debugging) if necessary, and then, if there are no more errors, interpreting the byte code. At this final stage the JVM loads not only your program's byte code but also that contained in any libraries that are required.

Java IDEs such as *J++* or *JBuilder* are also able to compile programs that are not just applets, but **applications**; an application is a program designed to run on your machine just like any other program. Rather than being interpreted by the browser's JVM, applications are interpreted by the IDE's built-in interpreter, or by a separate JVM built into the computer's operating system, or by a standalone program (usually called **java**).

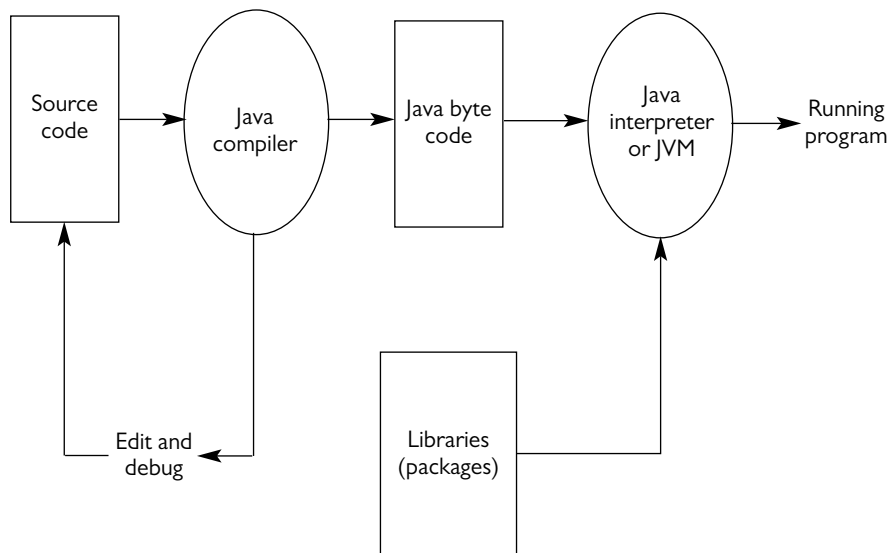
Until chapter 12 we will be creating applications only, and not worrying about applets designed for the World Wide Web. However, once you have grasped the programming principles in this book it will be an easy step to move on to writing applets; indeed in chapter 9, the graphical applications that we will write will be constructed in such a way that converting them to applets in the future will be an easy matter.

The process of compiling, interpreting and running Java programs is summarized in figure 1.4.

If you are working in a Windows or Windows-type environment, your compiler program will provide you with an easy-to-use window into which you can type your code; other windows will provide information about the files you are using; and a separate window will be provided to tell you of your errors. Your screen will probably look something like that in figure 1.5.

The source code that you write will be saved in a file with an extension **.java** (for example `myProgram.java`) and once it is compiled into byte code it will be kept in a file with an extension **.class** (for example `myProgram.class`).

If you are working in a UNIX or UNIX-type environment you will be working from a command line interface. You will therefore be using a text editor to write your code, and – if



**Fig 1.4** Compiling, interpreting and running Java programs

## 10 THE FIRST STEP

you are writing a program called `myProgram`, for example – you will probably have to write something like:

```
vi myProgram.java
```

You will then need to invoke the compiler with a line like:

```
javac myProgram.java
```

And to run your program you will need to invoke the interpreter with a line like:

```
java myProgram
```

### 1.7 *Your first program*

At last it is time to write your first program. Anyone who knows anything about programming will tell you that the first program that you write in a new language has always got to be a program that displays the words “Hello world” on the screen; so we will stick with tradition, and your first program will do exactly that!

When your program runs you will get a black and white black screen with the words “Hello world” displayed; if you are using a Windows-type environment there will probably be some additional stuff displayed before the “Hello world” line. The screen will remain like this until

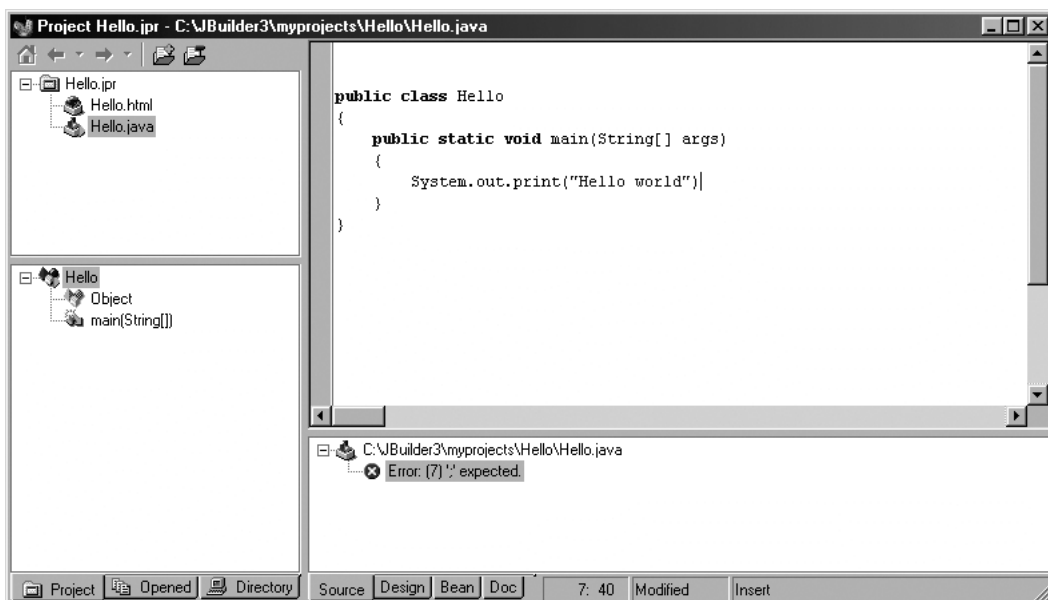


Fig 1.5 A typical Java IDE screen

you press the Enter key. Then the program will end and – in a Windows-type environment – the screen will disappear.

It sounds like a rather simple program, and so it is. However, before you type it in and compile and run it there is something we need to tell you about: Java wasn't really designed to write simple text messages on boring black and white screens! It is a language that can produce interesting and attractive graphical applications that run over the Web – but before you can begin to write such programs you need to learn the basics and you are going to be using this simple text screen right up till chapter 9; by that time you will have the fundamentals of programming firmly understood.

Now, because of the potential of the Java language, the people who developed it never really bothered to provide simple methods of getting text from the keyboard and displaying it on the screen, because it is usually done with attractive icons and mouse-clicks and so on. So getting input from the keyboard can involve lots of lines of program code that can look pretty complicated and get in the way of your learning the basic programming principles.

But don't worry! We have made it easy for you! We have created something called EasyIn. You will need to download the file called EasyIn.java from our website (or copy it from the CD-ROM). You should put this in the folder where your compiler automatically looks for files, so that EasyIn.java can be interpreted along with your program.

So now we can get on with our “Hello world” Program, which is written out for you below as program 1.1

#### PROGRAM 1.1

```
public class Hello
{
    public static void main(String[] args)
    {
        System.out.print("Hello world");
        EasyIn.pause();
    }
}
```

As we have said, when you run the program, you should get a screen with the words “Hello world” written on it. If you are using a Windows-type environment the screen will remain like this until you press the Enter key, at which point the program ends and the black screen disappears. In a UNIX-type environment you will simply be returned to your usual prompt when the program terminates.

Let's examine what is really going on here. The program we are writing is really incredibly simple. It displays the words “Hello world” on the screen, then it pauses and waits for you to press the Enter key. A simple sequence involving two actions: the program displays the message on the screen and then it pauses until the user presses Enter. If we didn't have the pause statement here, then, in a Windows environment, the screen would simply disappear before you had a chance to read what was on it.

### 1.7.1 Analysis of the “Hello world” program

We will consider the meaning of the program line by line. The first line, which we call the header, looks like this:

```
public class Hello
```

The first, and most important, thing to pay attention to is the word **class**. We noted earlier that Java is referred to as an **object-oriented programming language**. Now the true meaning of this will start to become clear in chapter 4 – but for the time being you just need to know that object-oriented languages require the program to be written in separate units called **classes**. There are two aspects to a class: the information that it holds and the things that it can do. The different bits of information (or **data**) that belong to a class are referred to as the **attributes** of the class; the operations that a class can perform are referred to as the **methods** of the class.

In forthcoming chapters we will be using the notation of the **Unified Modeling Language (UML)**<sup>2</sup> to specify and design our classes; UML is the standard method of specifying and designing object-oriented systems. In UML a class is represented diagrammatically by a rectangle divided into three sections as shown in figure 1.6.

The simple programs that we are starting off with will contain only one class (although they will interact with other classes like `EasyIn`, and the “built-in” Java libraries) – in this case we have called our class `Hello`. The first line therefore tells the Java compiler that we are writing a class with the name `Hello`. You will also have noticed the word `public` in front of the word `class`; placing this word here makes our class accessible to the outside world so we should include this word in the header of the main class of an application.

Notice that everything in the class has to be contained between two curly brackets that look like this `{ }`; these tell the compiler where the class begins and ends.

There is one important thing that we should point out here. Java is *case-sensitive* – in other words it interprets upper case and lower case characters as two completely different things – it is very important therefore to type the statements exactly as you see them here, paying attention to the case of the letters.

The next line that we come across (after the opening curly bracket) is this:

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

This looks rather strange if you are not used to programming – but you will see that every program we write in the first few weeks is going to follow on from this line. Now, we have already noted that the Java language requires that every program consists of at least one class and that a class will contain attributes and methods. Our `Hello` class in fact contains no attributes and just one method and this line introduces that method. In fact it is a very special

<sup>2</sup> Simon Bennett, Steve McRobb and Ray Farmer, *Object-Oriented Systems Analysis and Design using UML*, McGraw-Hill UK, 1999.

method called a **main** method. Applications in Java (as opposed to applets, which run in a browser) must always contain a class with a method called **main**: this is where the program begins. A program starts with the first instruction of **main**, then obeys each instruction in sequence (unless the instruction itself tells it to jump to some other place in the program). The program terminates when it has finished obeying the final instruction of **main**.

So this line that we see above introduces the **main** method; the program instructions are now written in a second set of curly brackets that show us where this **main** method begins and ends. At the moment we will not worry about the words in front of **main** (**public static void**) and the bit in the brackets (`String[] args`) – we will just accept that they always have to be there; you will begin to understand their significance as you learn more about programming concepts.

Now, at last we are in a position to examine the important bits – the two lines of code that represent our instructions, display “Hello world” and pause.

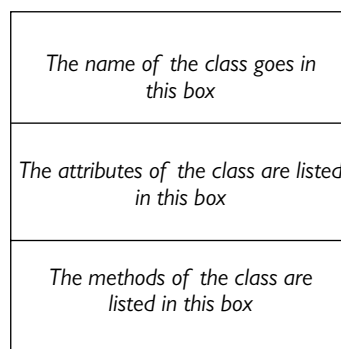
The line that gets “Hello world” displayed on the screen is this one:

```
System.out.print("Hello world");
```

This is the way we are always going to get stuff printed on the screen; we use `System.out.print` (or sometimes `System.out.println`, as explained below) and put whatever we want to be displayed in the brackets. You won’t understand at this stage why it has to be in this precise form (with each word separated by a full stop, and the actual phrase in double quotes), but do make sure that you type it exactly as you see it here, with an upper case S at the beginning and all the rest in lower case. Also, you should notice the semi-colon at the end of the statement. This is important; every Java instruction has to end with a semi-colon.

The next line is the *pause* line:

```
EasyIn.pause();
```



**Fig 1.6** A UML class template

## 14 THE FIRST STEP

As you can see, we are already starting to make use of the `EasyIn` class; we will do this every time we want to get input from the keyboard. In this case we are simply waiting for the user to press the `Enter` key, and to do this we use the `pause` method of `EasyIn`. In the next chapter you will see how to use other `EasyIn` methods to get other types of input.

### 1.7.2 Some variations to the *Hello world* program

As we mentioned above, there is an alternative form of the `System.out.print` statement, which uses `System.out.println`. The `println` is short for “print line” and the effect of using this statement is to start a new line after displaying whatever is in the brackets. We can see the effect of this from program 1.2 – we have renamed our class `Hello2` to avoid confusion; the change to the code is in bold:

#### PROGRAM 1.2

```
public class Hello2
{
    public static void main(String[] args)
    {
        System.out.println("Hello world");
        EasyIn.pause();
    }
}
```

At first glance the output from this program might not look very different – but notice that the cursor is now flashing at the start of the line following the “Hello world” line, instead of at the end of the “Hello world” line!

So now you could add extra lines if you wanted, as, for example, in program 1.3:

#### PROGRAM 1.3

```
public class Hello3
{
    public static void main(String[] args)
    {
        System.out.println("Hello world");
        System.out.print("This is my third Java Program");
        EasyIn.pause();
    }
}
```

Finally, there is another way in which we can use the `pause` method of `EasyIn`. We can get a message printed on the screen by placing text in the brackets. This is demonstrated in program 1.4.

**PROGRAM 1.4**

```
public class Hello4
{
    public static void main(String[] args)
    {
        System.out.println("Hello world");
        EasyIn.pause("Press <Enter> to quit");
    }
}
```

When you try this out you will see the words “Press <Enter> to quit” displayed beneath “Hello world”. In future we are going to end all our text-based programs with a line like this.

Before we finish there is one more thing to tell you. When we write program code, we will often want to include some comments to help remind us what we were doing when we look at our code a few weeks later, or to help other people to understand what we have done.

Of course we want the compiler to ignore these comments when the code is being compiled. There are two ways of doing this. For short comments we place two slashes (//) at the beginning of the line – everything after these slashes, up to the end of the line, is then ignored by the compiler.

For longer comments (that is, ones that run over more than a single line) we usually use another method. The comment is enclosed between two special symbols; the opening symbol is a slash followed by a star (/\*) and the closing symbol is a star followed by a slash (\*//). Everything between these two symbols is ignored by the compiler. Program 1.5 below shows examples of both types of comment; when you compile and run this program you will see that the comments have no effect on the code, and the output is exactly the same as that of program 1.4.

**PROGRAM 1.5**

```
// this is a short comment, so we use the first method
public class Hello5
{
    public static void main(String[] args)
    {
        System.out.println("Hello world");
        EasyIn.pause("Press <Enter> to quit");
    }
/* this is the second method of including comments - it is more convenient
to use this method here, because the comment is longer and goes over more
than one line */
}
```

**Tutorial exercises**

1. Explain what is meant by each of the following terms:
  - machine code;
  - assembly language;
  - high-level (third generation) language;
  - source code;
  - Java byte code;
  - library.
2. Explain the difference between *compilation* and *interpretation* of programs.
3. Explain the difference between a Java compiler and a conventional compiler.

**Practical work**

1. Type, compile and run programs 1.1 to 1.5 from this chapter.
2. Write a program that displays your name, address and telephone number, each on separate lines.
3. Adapt the above program to include a blank line between your address and telephone number.
4. Write a program that displays your initials in big letters made of asterisks. For example:

```

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

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

```

or

```

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

```