The First Step

1

Outcomes:

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

- *explain the meaning of the terms* ***software***, ***program***, ***source code, program code***;
- *distinguish between* ***application software*** *and* ***system software***;
- *explain how Java programs are compiled and run*;
- *provide examples of different types of java applications*;
- *write Java programs that display text on the screen*;
- *join messages in output commands by using the* ***concatenation*** *(+) operator*;
- *add comments to programs*.

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 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 we need to make sure that you understand what we mean by important terms such as *program*, *software, code* and *programming languages*.

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_1

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**. Programs that the computer can use can be stored on electronic chips that form part of the computer, or can be stored on devices like hard disks, CDs, DVDs, and USB drives (sometimes called memory sticks), and can often be downloaded via the Internet.

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**. This is the name given to useful programs that a user might need; for example, word-processors, spreadsheets, accounts programs, games and so on. Such programs are often referred to simply as **applications**.
- **System software**. This 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).

Of course software is not restricted simply to computers themselves. Many of today's devices—from mobile phones to microwave ovens to games consoles—rely on computer programs that are built into the device. Such software is referred to as **embedded software**.

Both application and system software are built by writing a set of instructions for the computer to obey. **Programming**, or **coding**, is the task of writing these instructions. These instructions have to be written in a language specially designed for this purpose. These **programming languages** include C++, Visual Basic, Python and many more. The language we are going to use in this book is Java. Java is an example of an **object-oriented** programming language. Right now, that phrase might not mean anything to you, but you will find out all about its meaning as we progress through this book.

1.3 Compiling Programs

Like most modern programming languages, the Java language consists of instructions that look a bit like English. For example, words such as **while** and **if** are part of the Java language. The set of instructions written in a programming language is called the **program code** or **source code**.

Ultimately these instructions have to be translated into a language that can be understood by the computer. The computer understands only **binary** instructions—that means instructions written as a series of 0s and 1s. So, for example, the machine might understand 01100111 to mean add. The language of the computer is often referred to as **machine code**. A special piece of system software called a **compiler** translates the instructions written in a programming language into

machine instructions consisting of 0s and 1s. This process is known as **compiling**. Figure 1.1 illustrates how this process works for many programming languages.

Programming languages have a very strict set of rules that you must follow. Just as with natural languages, this set of rules is called the **syntax** of the language. A program containing syntax errors will not compile. You will see when you start writing programs that the sort of things that can cause compiler errors are the incorrect use of special Java keywords, missing brackets or semi-colons, and many others. If, however, the source code is free of such errors the compiler will successfully produce a machine code program that can be run on a computer, as illustrated.

Once a program has been compiled and the machine code program saved, it can be run on the target machine as many times as necessary. When you buy a piece of software such as a game or a word processor, it is this machine code program that you are buying.

1.4 Programming in Java

Before the advent of Java, most programs were compiled as illustrated in Fig. 1.1. The only problem with this approach is that the final 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 a Mac™ or a UNIX™ machine.

But this is not the case with Java. Java—and nowadays many other languages—is **platform-independent**. A Java program will run on any type of computer.

How is this achieved? The answer lies in the fact that any Java program requires the computer it is running on to also be running 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.

For example, you can get a JVM for a PC running Windows™; there is a JVM for a MAC™, and one for a Unix™ or Linux™ box. There is a special kind of JVM for mobile phones; and there are JVMs built into machines where the embedded software is written in Java.

We saw earlier that conventional compilers translate our program code into machine code. This machine code would contain the particular instructions appropriate to the type of 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**. Java byte code, which, like machine code, consists of 0s and 1s, contains instructions that are exactly the same irrespective of the type of computer—it is *universal*, whereas machine code is specific to a particular type of computer. The job of the JVM is to translate each byte code instruction for the computer it is running on, before the instruction is performed. See Fig. 1.2.

There are various ways in which a JVM can be installed on a computer. In the case of some operating systems a JVM comes packaged with the system, along with the Java **libraries**, or **packages**, (pre-compiled Java modules that can be integrated

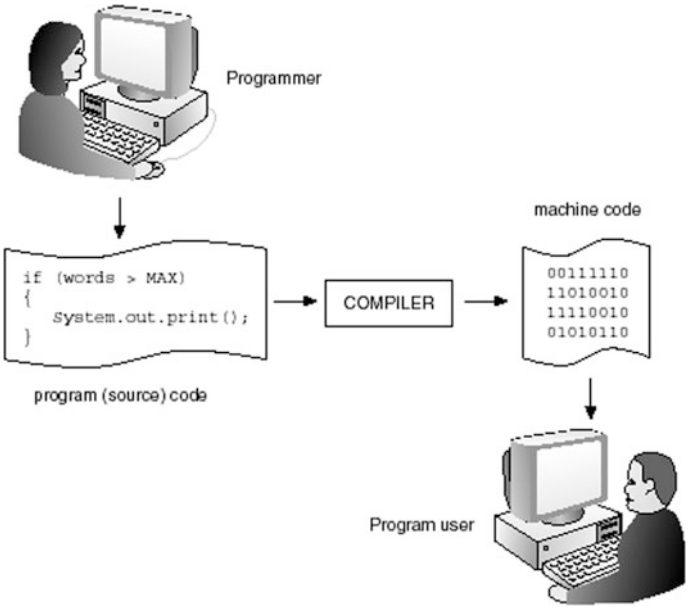

Fig. 1.1 The compilation process

with the programs you create) and a compiler. Together the JVM and the libraries are known as the **Java Runtime Environment (JRE)**. If you do not have a JRE on your computer (as will be the case with any Windows™ operating system), then the entire Java Development Kit (JDK), comprising the JRE, compiler and other tools, can be downloaded from Oracle™, the owners of the Java platform.[1]

1.5 Integrated Development Environments (IDEs)

It is very common to compile and run your programs by using a special program called an **Integrated Development Environment** or **IDE**. An IDE provides 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.

Not only does an IDE do all these things, it also lets you run your programs as soon as you have compiled them. Depending on the IDE you are using, your screen will look something like that in Fig. 1.3.

[1]The original developers of Java were Sun Microsystems™. This company was acquired by Oracle™ in 2010.

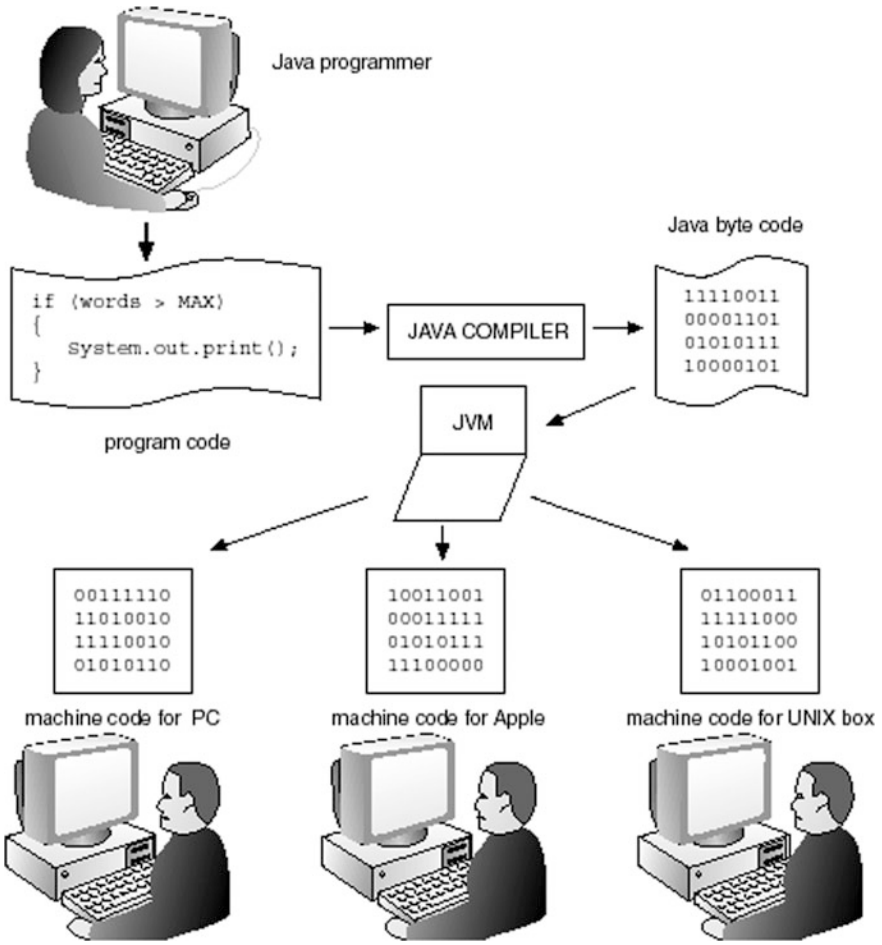

Fig. 1.2 Compiling Java programs

The IDE shown in Fig. 1.3 is NetBeans™, a very commonly used compiler for Java—another widely used IDE is Eclipse™. Instructions for installing and using an IDE are on the website (see preface for details).

It is perfectly possible to compile and run Java programs without the use of an IDE—but not nearly so convenient. You would do this from a command line in a console window. The source code that you write is saved in the form of a simple text file which has a `.java` extension. The compiler that comes as part of the JDK is called `javac.exe`, and to compile a file called, for example, `MyProgram.java`, you would write at the command prompt:

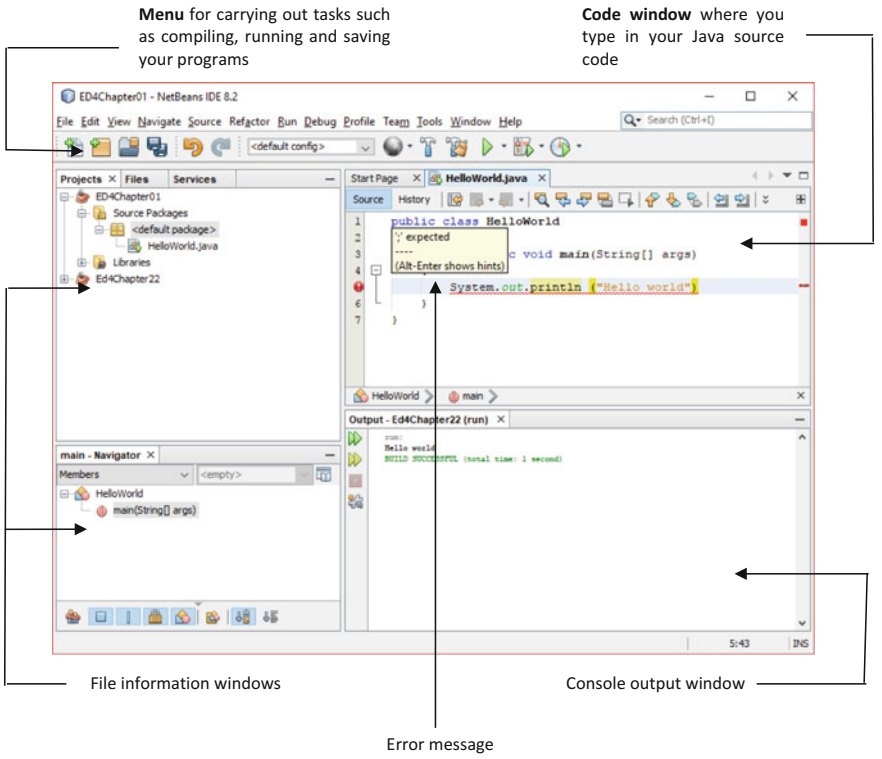

Fig. 1.3 A typical Java IDE screen

javac MyProgram.java

This would create a file called `MyProgram.class`, which is the compiled file in Java byte code. The name of the JVM is `java.exe` and to run the program you would type:

java MyProgram

To start off with however, we strongly recommend that you use an IDE such as NetBeans™ or Eclipse™.

1.6 Java Applications

As we explained in Sect. 1.2, Java applications can run on a computer, on such devices as mobile phones and games consoles, or sometimes can be embedded into an electronic device. In the last case you would probably be unaware of the fact that the software is running at all, whereas in the former cases you would be seeing

output from your program on a screen and providing information to your program via a keyboard and mouse, via a touch screen, or via a joystick or game controller.

The screen that provides output from your program, and prompts you to enter information, is known as the **user interface**. There are two principal types of user interface:

- **text** based;
- **graphics** based.

With text based user interfaces, information is displayed simply as text—with no pictures. Text based programs make use of the keyboard for user input. Text based programs are known as **console applications**. If you are using an IDE, the console window is usually integrated into the IDE as you saw in Fig. 1.3. However, if you are running a program from the command prompt you will see a window similar to that shown in Fig. 1.4.

You are probably more accustomed to running programs that have a **graphical user interface (GUI)**. Such interfaces allow for pictures and shapes to be drawn on the screen (such as text boxes and buttons) and make use of the mouse as well as the keyboard to collect user input. An example of a GUI is given in Fig. 1.5.

Eventually we want all your programs to have graphical interfaces, but these obviously require a lot more programming effort to create than simple console applications. So, for most of the first semester, while we are teaching you the fundamentals of programming in Java, we are going to concentrate on getting the program logic right and we will be sticking to console style applications. Once you have mastered these fundamentals, however, you will be ready to create attractive graphical interfaces before the end of this very first semester.

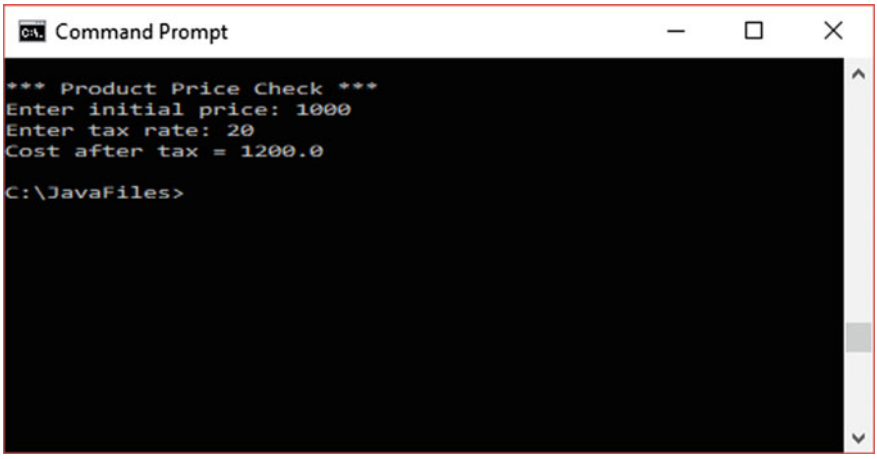

Fig. 1.4 A Java console application

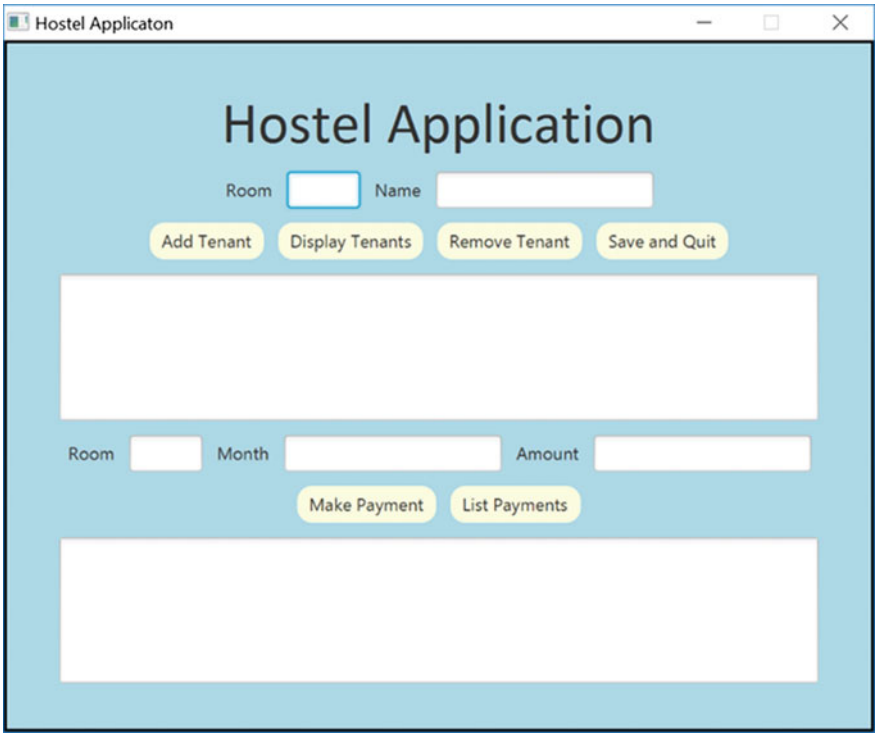

Fig. 1.5 A graphical application

1.7 Your First Program

Now 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 see the words "Hello world" displayed. The type of window in which this is displayed will vary according to the particular operating system you are running, and the particular compiler you are using.

The code for the "Hello world" program is written out for you below.

HelloWorld

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

1.7.1 Analysis of the "Hello World" Program

Let's start with the really important bit—the line of code that represents the instruction *display "Hello world" on the screen*. The line that does this looks like this:

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

This is the way we are always going to get stuff printed on a simple text screen; we use `System.out.println` (or sometimes `System.out.print`, as explained below) and put whatever we want to be displayed in the brackets. The `println` is short for "print line" by the way. 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. 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.

Now we can consider the meaning of the rest of the program. The first line, which we call the header, looks like this:

```
public class HelloWorld
```

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 Chap. 7—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**. The simple programs that we are starting off with will contain only one class (although they will interact with other classes from the "built-in" Java libraries). We always have to give a name to a class and in this case we have simply called our class `HelloWorld`.

When choosing a name for a class, you can choose any name as long as:

- the name is not already a keyword in the Java language (such as **`static`**, **`void`**);
- the name has no spaces in it;
- the name does not include operators or mathematical symbols such as + and −;
- the name starts either with a letter, an underscore (_), or a dollar sign ($).

So, the first line tells the Java compiler that we are writing a class with the name `HelloWorld`. However, 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 and to other classes—so, until we learn about specific ways of restricting access (in the second semester) we will always include this word in the header. A **public** class should always be saved in a file with the same name as the class itself—so in this case it should be saved as a file with the name `HelloWorld.java`.

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

There is one important thing that we must emphasize 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 application we write is going to contain one class with this line in it. In Chap. 7 you will find out that this basic unit called a class is made up of, among other things, a number of **methods**. You will find out a lot more about methods in Chap. 5, but for now it is good enough for you to know that a method contains a particular set of instructions that the computer must carry out. Our `HelloWorld` class contains just one method and this line introduces that method. In fact it is a very special method called a `main` method. Applications in Java 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`.[2]

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 **public static void** in front of `main`, and the bit in the brackets afterwards (`String[] args`)[3]—we will just accept that they always have to be there; you will begin to understand their significance as you learn more about

[2]In Chap. 10 you will learn to create graphics programs with a package called JavaFX, and in the case of JavaFX applications you will see that in some instances it is possible to run a JavaFX application without a `main` method.

[3]In fact, if you left out the words in brackets your program would still compile—but it wouldn't do what you wanted it to do!

programming concepts. The top line of a method is referred to as the method **header** and words such as **`public`** and **`static`**, that are part of the Java language, are referred to as **keywords**.[4]

As we have said, we place instructions inside a method by surrounding them with opening and closing curly brackets. In Java, curly brackets mark the beginning and end of a group of instructions. In this case we have only one instruction inside the curly brackets but, as you will soon see, we can have many instructions inside these braces.

By the way, you should be aware that the compiler is not concerned about the layout of your code, just that your code meets the rules of the language. So we could have typed the method header, the curly brackets and the `println` command all on one line if we wished! Obviously this would look very messy, and it is always important to lay out your code in a way that makes it easy to read and to follow. So throughout this book we will lay out our code in a neat easy-to-read format, lining up opening and closing braces.

1.7.2 Adding Comments to a Program

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 different 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 approach. 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. The program 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 the original program.

[4]You will notice that we are using bold courier font for Java keywords.

***HelloWorld* – with comments**

```
// this is a short comment, so we use the first method

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

/* 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 */

}
```

In Chap. 11 you will learn about a special tool called **Javadoc** for documenting your programs. In that chapter you will see that in order to use this tool you must comment your classes in the Javadoc style—as you will see, Javadoc comments must begin with /** and end with */.

1.8 Output in Java

As you have already seen when writing your first program, to output a message on to the screen in Java we use the following command:

`System.out.println(`*`message to be printed on screen`*`);`

For example, we have already seen:

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

This prints the message "Hello world" onto the screen. There is in fact an alternative form of the `System.out.println` statement, which uses `System.out.`**`print`**. As we said before, `println` is short for *print line* and the effect of this statement is to start a new line after displaying whatever is in the brackets. You can see the effect of this below—we have adapted our program by adding an additional line.

***HelloWorld* – with an additional line**

```
public class HelloWorld
{
     public static void main(String[] args)
     {
          System.out.println("Hello world"); // notice the use of println
          System.out.println("Hello world again!");
     }
}
```

When we run this program, the output looks like this:

```
Hello world
Hello world again!
```

Now let's change the first `System.out.println` to `System.out.print`:

HelloWorld* – adapted to show the effect of using *print* instead of *println

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        System.out.print("Hello world"); // notice the use of 'print'
        System.out.println("Hello world again!");
    }
}
```

Now our output looks like this:

```
Hello worldHello world again!
```

You can see that the output following the `System.out.print` statement doesn't start on a new line, but follows straight on from the previous line.

By the way, if you want a blank line in the program, then you can simply use `println` with empty brackets:

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

Messages such as "Hello world" are in fact what we call **strings** (collections of characters). In Java, literal strings like this are always enclosed in speech marks. We shall explore strings in depth in Chap. 7. However, it is useful to know now how several strings can be printed on the screen using a single output command.

In Java, two strings can be joined together with the plus symbol (+). When using this symbol for this purpose it is known as the **concatenation operator**. For example, instead of printing the single string "Hello world", we could have joined two strings, "Hello" and "world", for output using the following command:

```
System.out.println("Hello " + "world");
```

Note that spaces are printed by including them within the speech marks ("`Hello `"), not by adding spaces around the concatenation operator (which has no effect at all).

1.9 Self-test Questions

1. Explain the meaning of the following terms:

   - program;
   - software;
   - application software;
   - system software;
   - machine code;
   - source code;.
   - embedded software;
   - compilation;
   - Java byte code;
   - Java virtual machine;
   - integrated development environment;

2. Explain how Java programs are compiled and run.

3. Describe two different ways of adding comments to a Java program.

4. What is the difference between using `System.out.println` and `System.out.print` to produce output in Java?

5. What, precisely, would be the output of the following programs?

(a)

```
public class Question5A
{
       public static void main(String[] args)
       {
           System.out.print("Hello, how are you? ");
           System.out.println("Fine thanks.");
       }
}
```

(b)

```
public class Question5B
{
       public static void main(String[] args)
       {
           System.out.println("Hello, how are you? ");
           System.out.println("Fine thanks.");
       }
}
```

(c)

```
public class Question5C
{
       public static void main(String[] args)
       {
              System.out.println("1 + 2 " + "+ 3" + " = 6");
       }
}
```

6. Identify the syntax errors in the following program:

```
public class
{
       public Static void main(String[] args)
       {
           system.out.println( I want this program to compile)
       }
```

1.10 Programming Exercises

1. If you do not have access to a Java IDE go to the accompanying website and follow the instructions for installing an IDE. You will also find instructions on the website for compiling and running programs.
2. Type and compile the *Hello World* program. If you make any syntax errors, the compiler will indicate where to find them. Correct them and re-compile your program. Keep doing this until you no longer have any errors. You can then run your program.
3. Make the changes to the *Hello World* program that are made in this chapter, then each time re-compile and run the program again.
4. Type and compile the program given in self test question 6 above. This program contained compiler errors that you should have identified in your answer to that question. Take a look at how the compiler reports on these errors then fix them so that the program can compile and run successfully.
5. Write a program that displays your name, address and telephone number, each on separate lines.
6. Adapt the above program to include a blank line between your address and telephone number.

7. Write a program that displays your initials in big letters made of asterisks. For example:

Do this by using a series of `println` commands, each printing one row of asterisks.