Chapter 1

Introduction

CHAPTER GOALS

- To understand the activity of programming
- To learn about the architecture of computers
- To learn about machine code and high-level programming languages
- To become familiar with your computing environment and your compiler
- To compile and run your first Java program
- To recognize syntax and logic errors

CHAPTER CONTENTS

1.1 What Is a Computer?

You have probably used a computer for work or fun. Many people use computers for everyday tasks such as balancing a checkbook or writing a term paper. Computers are good for such tasks. They can handle repetitive chores, such as totaling up numbers or placing words on a page, without getting bored or exhausted. More importantly, the computer presents you with the checkbook or the term paper on the screen and lets you fix up mistakes easily. Computers also make good game machines because they can play sequences of sounds and pictures, involving the human user in the process.

A computer must be programmed to perform tasks. Different tasks require different programs.

What makes all this possible is not just the computer. The computer must be *programmed* to perform these tasks. A computer itself is a machine that stores data (numbers, words, pictures), interacts with devices (the monitor screen, the sound system, the printer), and executes programs. Programs are sequences of instructions and decisions that the computer carries out to achieve a task. One program balances checkbooks; a different program, perhaps designed and constructed by a different company, processes words; and a third program, probably from yet another company, plays a game.

A computer program executes a sequence of very basic operations in rapid succession.

Today's computer programs are so sophisticated that it is hard to believe that they are all composed of extremely primitive operations. A typical operation may be one of the following:

- Put a red dot onto this screen position.
- Send the letter A to the printer.
- Get a number from this location in memory.
- Add up these two numbers.
- If this value is negative, continue the program at that instruction.

Only because a program contains a huge number of such operations, and because the computer can execute them at great speed, does the computer user have the illusion of smooth interaction.

The flexibility of a computer is quite an amazing phenomenon. The same machine can balance your checkbook, print your term paper, and play a game. In contrast, other machines carry out a much narrower range of tasks; a car drives, and a toaster toasts. Computers can carry out a wide range of tasks because they execute different programs, each of which directs the computer to work on a specific task.

1.2 What Is Programming?

Programmers develop computer programs to make computers perform new tasks.

A computer program tells a computer, in minute detail, the sequence of steps that are needed to fulfill a task. The act of designing and implementing these programs is called computer programming. As you work through this book, you will learn how to program a computer—that is, how to direct the computer to execute tasks.

To use a computer you do not need to do any programming. When you write a term paper with a word processor, that software package has been programmed by the manufacturer and is ready for you to use. That is only to be expected—you can drive a car without being a mechanic and toast bread without being an electrician. Many people who use computers every day in their careers never need to do any programming.

Of course, a professional computer scientist or software engineer does a great deal of programming. You are reading this introductory computer science book, so your career goal may well be to become such a professional. Programming is not the only skill required of a computer scientist or software engineer; indeed, programming is not the only skill required to create successful computer programs. Nevertheless, the activity of programming is an important part of computer science. It is also a fascinating and pleasurable activity that continues to attract and motivate students. The discipline of computer science is particularly fortunate that it can make such an interesting activity the foundation of the learning path.

Writing a computer game with motion and sound effects or a word processor that supports fancy fonts and pictures is a complex task that requires a team of many highly skilled programmers. Your first programming efforts will be more mundane. The concepts and skills you learn in this book form an important foundation, and you should not be disappointed if your first programs do not rival the sophisticated software that is familiar to you. Actually, you will find that there is an immense thrill even in simple programming tasks. It is an amazing experience to see the computer precisely and quickly carry out a task that would take you hours of drudgery, to make small changes in a program that lead to immediate improvements, and to see the computer become an extension of your mental powers.

1.3 The Anatomy of a Computer

To understand the programming process, you need to have a rudimentary understanding of the building blocks that make up a computer. We will look at a personal computer. Larger computers have faster, larger, or more powerful components, but they have fundamentally the same design.

At the heart of the computer lies the central processing unit (CPU).

At the heart of the computer lies the *central processing unit* (CPU) (see Figure 1). It consists of a single *chip* (integrated circuit) or a small number of chips. A computer chip is a component with a plastic or metal housing, metal connectors, and inside wiring made principally from silicon. For a CPU chip, the inside wiring is enormously complicated. For example, the Pentium III chip (a popular CPU for personal computers at the time of this writing) contains over 28 million structural elements called *transistors*—the elements that enable electrical signals to control other electrical signals, making automatic computing possible. The CPU performs program control, arithmetic, and data movement. That is, the CPU locates and executes the program instructions; it carries out arithmetic operations such as addition, subtraction, multiplication, and division; it fetches data from external memory or devices or stores data back. All data must travel through the CPU whenever it is moved from one location to another. (There are a few technical exceptions to this rule; some devices can interact directly with memory.)

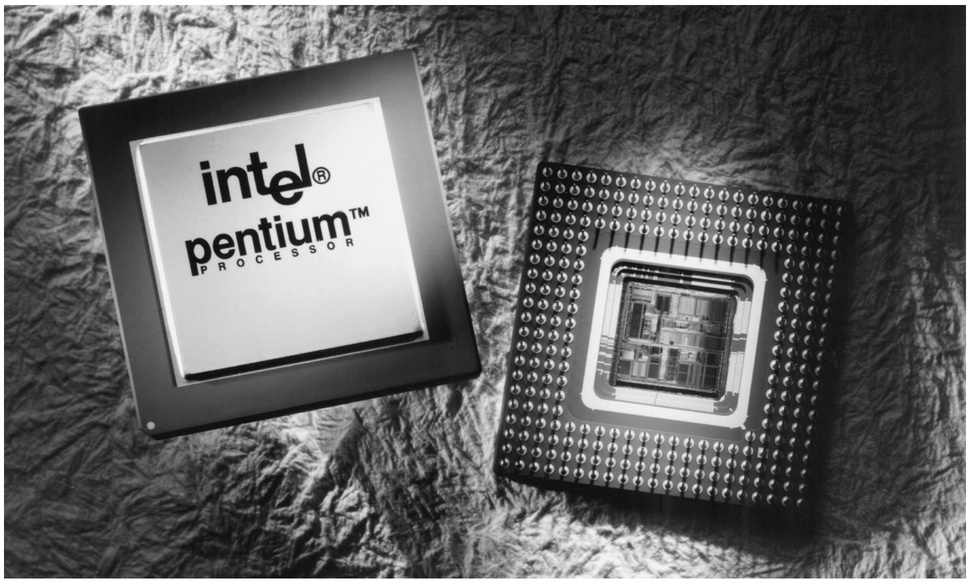


Figure 1

Central Processing Unit

Data and programs are stored in primary storage (memory) and secondary storage (such as a hard disk).

The computer keeps data and programs in *storage.* There are two kinds of storage. *Primary storage,* also called *random-access memory* (*RAM*) or simply *memory,* is fast but expensive; it is made from memory chips (see Figure 2). Primary storage has two disadvantages. It is comparatively expensive, and it loses all its data when the power is turned off. *Secondary storage,* usually a *hard disk* (see Figure 3), provides less expensive storage that persists without electricity. A hard disk consists of rotating platters, which are coated with a magnetic material, and read/write heads, which can detect and change

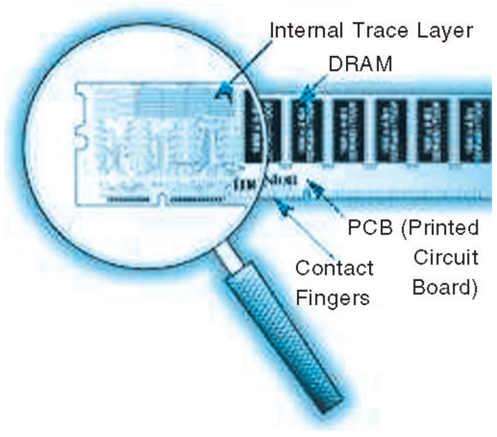


Figure 2

A Memory Module with Memory Chips

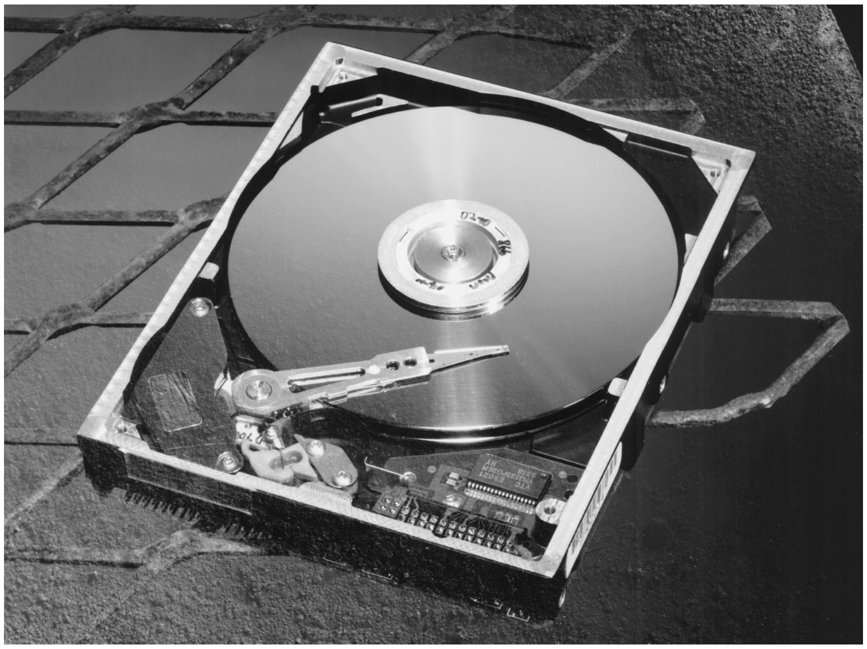


Figure 3

A Hard Disk

the patterns of varying magnetic flux on the platters. This is essentially the same recording and playback process that is used in audio or video tapes.

Some computers are self-contained units, whereas others are interconnected through *networks*. Home computers are usually intermittently connected to the Internet via a modem. The computers in your computer lab are probably permanently connected to a local area network. Through the network cabling, the computer can read programs from central storage locations or send data to other computers. For the user of a networked computer, it may not even be obvious which data reside on the computer itself and which are transmitted through the network.

Most computers have *removable storage* devices that can access data or programs on media such as floppy disks, tapes, or compact discs (CDs).

The most common use for a floppy disk is to move data from one computer to another; you can copy data from your home computer and bring the disk to school to continue working with it. Now that network connections have become commonplace, some computer manufacturers have discontinued the use of floppy disk drives, because email or a network-based file-sharing service can transport data between networked computers much more quickly and easily.

Compact discs originally served as read-only memories (CD-ROMs), which, like a commercial audio CD, could only be "played back" to bring the data on them into memory, but more and more new computers support CDs that a personal computer user can record (CD-Rs) or even overwrite with new data (CD-RWs).

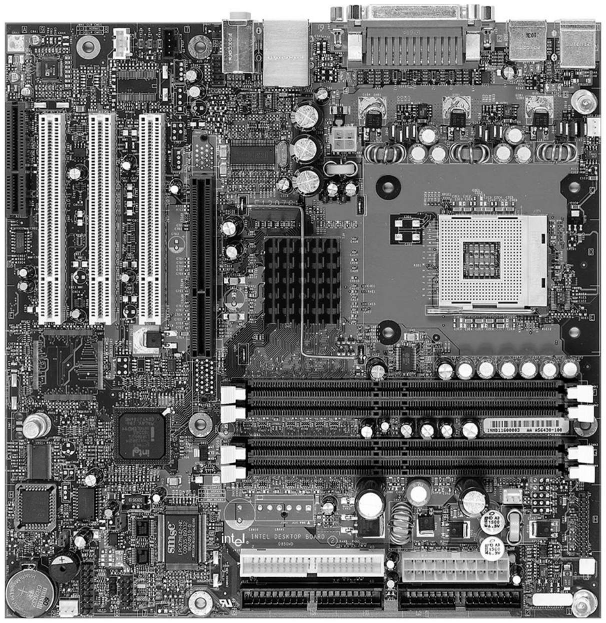


Figure 4

A Motherboard

To interact with a human user, a computer requires other peripheral devices. The computer transmits information to the user through a display screen, loudspeakers, and printers. The user can enter information and directions to the computer by using a keyboard or a pointing device such as a mouse.

The CPU, the RAM, and the electronics controlling the hard disk and other devices are interconnected through a set of electrical lines called a *bus*. Data travel along the bus from the system memory and peripheral devices to the CPU and back. Figure 4 shows a *motherboard*, which contains the CPU, the RAM, and *card slots*, through which cards that control peripheral devices connect to the bus.

Random Fact 1.1

The ENIAC and the Dawn of Computing

The ENIAC (*e*lectronic *n*umerical *i*ntegrator *a*nd *c*omputer) was the first usable electronic computer. It was designed by J. Presper Eckert and John Mauchly at the University of Pennsylvania and was completed in 1946. Instead of transistors, which were not invented until two years after it was built, the ENIAC contained about 18,000 *vacuum tubes* in many cabinets housed in a large room (see Figure 5). Vacuum tubes burned out at the rate of several tubes per day. An attendant with a shopping cart full of tubes con-

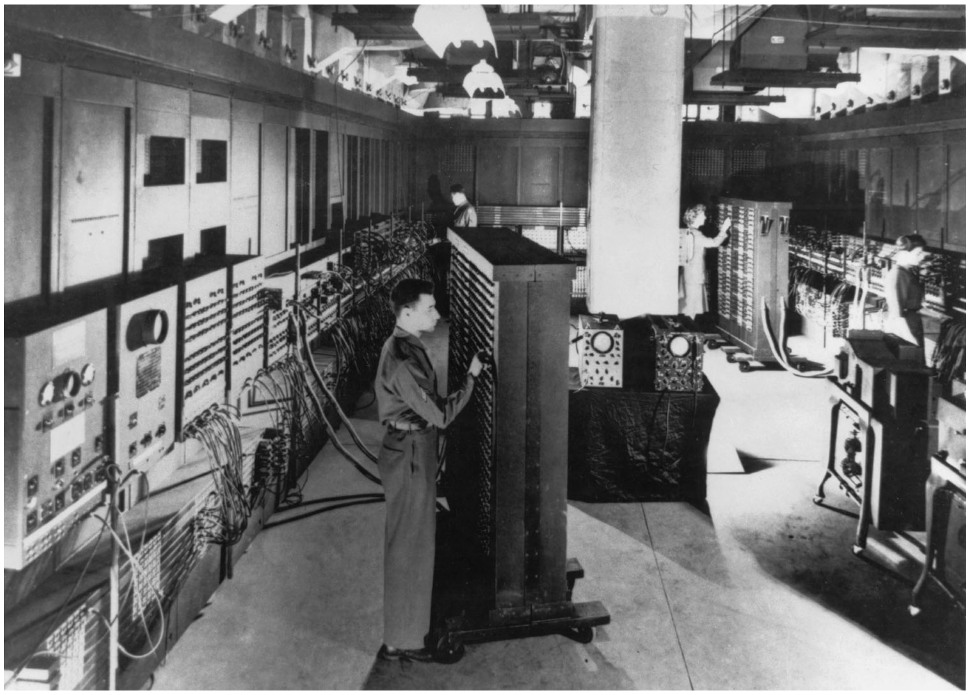


Figure 5

The ENIAC

stantly made the rounds and replaced defective ones. The computer was programmed by connecting wires on panels. Each wiring configuration would set up the computer for a particular problem. To have the computer work on a different problem, the wires had to be replugged.

Work on the ENIAC was supported by the U.S. Navy, which was interested in computations of ballistic tables that would give the trajectory of a projectile, depending on the wind resistance, initial velocity, and atmospheric conditions. To compute the trajectories, one must find the numerical solutions of certain differential equations; hence the name "numerical integrator". Before machines like ENIAC were developed, humans did this kind of work, and until the 1950s the word "computer" referred to these people. The ENIAC was later used for peaceful purposes such as the tabulation of U.S. census data.

Figure 6 gives a schematic overview of the architecture of a computer. Program instructions and data (such as text, numbers, audio, or video) are stored on the hard disk, on a CD, or elsewhere on the network. When a program is started, it is brought into memory, from which the CPU can read it. The CPU reads the program an instruction at a time. As directed by these instructions, the CPU reads data, modifies them, and writes them back to

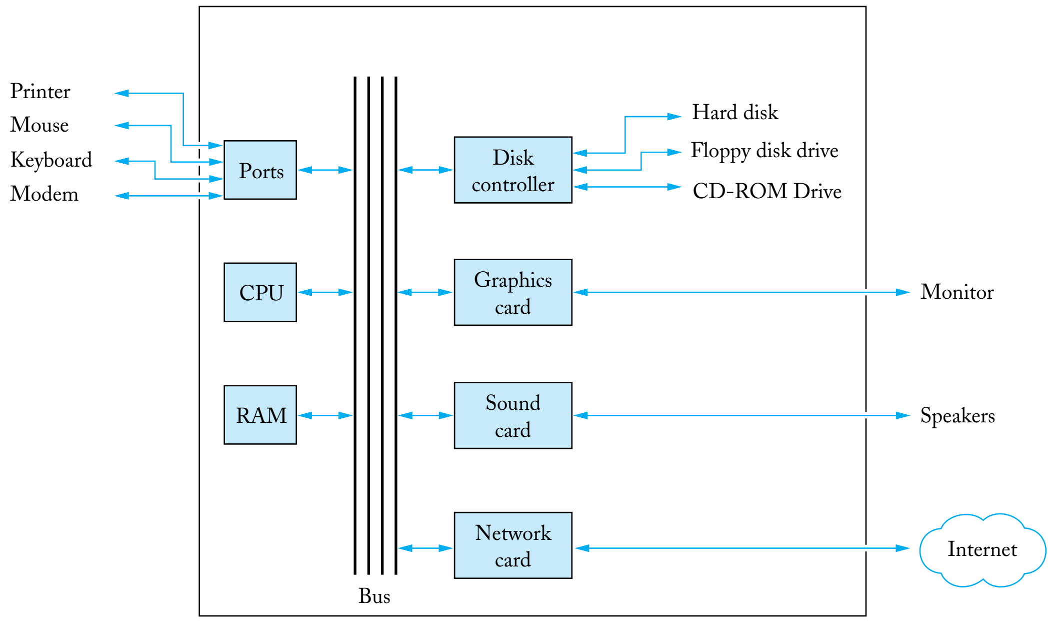


Figure 6

Schematic Diagram of a Computer

The CPU reads machine instructions from memory. The instructions direct it to communicate with memory, secondary storage, and peripheral devices.

RAM or to secondary storage. Some program instructions will cause the CPU to place dots on the display screen or to vibrate the speaker. As these actions happen many times over and at great speed, the human user will perceive images and sound. Similarly, the CPU can send instructions to a printer to mark the paper with patterns of closely spaced dots, which a human recognizes as text characters and pictures. Some program instructions read user input from the keyboard or mouse. The program analyzes the nature of these inputs and then executes the next appropriate instructions.

1.4 Translating Human-Readable Programs to Machine Code

Generally, machine code depends on the CPU type. However, the instruction set of the Java virtual machine (JVM) can be executed on many CPUs.

On the most basic level, computer instructions are extremely primitive. The processor executes *machine instructions*. CPUs from different vendors, such as the Intel Pentium or the Sun SPARC, have different sets of machine instructions. To enable Java applications to run on multiple CPUs without modification, most Java programs contain machine instructions for a so-called “Java virtual machine” (JVM), an idealized CPU that is then simulated by a program run on the actual CPU. The difference between actual and virtual machine instructions is not important to us—all you need to know is that machine instructions are very simple and can be executed very quickly.

A typical sequence of machine instructions is

1. Load the contents of memory location 40.
2. Load the value 100.
3. If the first value is greater than the second value, continue with the instruction that is stored in memory location 240.

Actually, machine instructions are encoded as numbers so that they can be stored in memory. On the Java virtual machine, this sequence of instruction is encoded as the sequence of numbers

```
21 40 16 100 163 240
```

Because machine instructions are encoded as numbers, it is difficult to write programs in machine code.

On a processor such as a Pentium or SPARC, the encoding would be quite different.

When the virtual machine fetches this sequence of numbers, it decodes them and executes the associated sequence of commands.

How can you communicate the command sequence to the computer? The simplest method is to place the actual numbers into the computer memory. This is, in fact, how the very earliest computers worked. However, a long program is composed of thousands of individual commands, and it is tedious and error-prone to look up the numeric codes for all commands and place the codes manually into memory. As we said before, computers are really good at automating tedious and error-prone activities, and it did not take long for computer programmers to realize that the computers themselves could be harnessed to help in the programming process.

The first step was to assign short names to the commands. For example, `iload` denotes “integer load”, `bipush` means “push integer constant”, and `if_icmpgt` means “if integers compare greater”. Using these commands, the instruction sequence becomes

```
iload 40
bipush 100
if_icmpgt 240
```

Assembly language makes it easier to generate machine instructions by translating mnemonics and symbolic names.

That is a lot easier to read for humans. To get the instruction sequences accepted by the computer, though, the names must be translated into the machine codes. Early computers used a computer program called an *assembler* to carry out these translations. An assembler takes the sequence of characters such as `iload`, translates it into the command code 21, and carries out similar operations on the other commands. Assemblers have another feature: They can give names to *memory locations* as well as to instructions. Our program sequence might have checked that some interest rate was greater than 100 percent, and the interest rate was stored in memory location 40. It is usually not important where a value is stored; any available memory location will do. When symbolic names are used instead of memory addresses, the program gets even easier to read:

```
iload      intRate
bipush     100
if_icmpgt  intError
```

It is the job of the assembler program to find suitable numeric addresses for the symbolic names and to put those addresses into the generated code sequence.

Assembler instructions were a major advance over programming with raw machine instructions, but they suffer from two problems: It still takes a great many instructions to achieve even the simplest goals, and the exact instruction sequence differs from one processor to another.

High-level languages let you describe tasks at a higher conceptual level than machine code.

In the mid-1950s, *high-level* programming languages began to appear. In these languages, the programmer expresses the idea behind the task that needs to be performed, and a special computer program, called a *compiler*, translates the high-level description into machine instructions for a particular processor.

For example, in Java, the high-level programming language that you will use in this book, you might give the following instruction:

```
if (intRate > 100)
   System.out.print("Interest rate error");
```

This means, "If the interest rate is over 100, display an error message." It is then the job of the compiler program to look at the sequence of characters `if (intRate > 100)` and translate that into

```
21 40 16 100 163 240
```

A compiler translates programs written in a high-level language into machine code.

Compilers are quite sophisticated programs. They have to translate logical statements, such as the `if`, into sequences of computations, tests, and jumps, and they must find memory locations for *variables*—items of information identified by symbolic names—like `intRate`. In this course, we will generally take the existence of a compiler for granted. If you decide to become a professional computer scientist, you may well learn more about compiler-writing techniques later in your studies.

1.5 Programming Languages

Each programming language has its own set of rules for forming instructions. Compilers enforce these rules strictly.

High-level programming languages are independent of specific computer architecture, but they are human creations. As such, they follow certain conventions. To ease the translation process, those conventions are much stricter than they are for human languages. When you talk to another person, and you scramble or omit a word or two, your conversation partner will usually still understand what you have to say. Compilers are less forgiving. For example, if you omit the quotation mark close to the end of the instruction,

```
if (intRate > 100)
  System.out.print("Interest rate error);
```

the Java compiler will get quite confused and complain that it cannot translate an instruction containing this error. That is actually a good thing. If the compiler were to try to guess what you did wrong and tried to fix it, it might not guess your intentions correctly. In that case, the resulting program would do the wrong thing—quite possibly with disastrous effects, if that program controlled a device on whose functions someone's well-being depends. When a compiler reads programming instructions in a programming language, it will translate them into machine code only if the input follows the language conventions exactly.

Just as there are many human languages, there are many programming languages. Consider the instruction

```
if (intRate > 100)
  System.out.print("Interest rate error");
```

This is how you must express a decision in Java. Java is a very popular programming language, and it is the one we use in this book. But in Pascal (another programming language that was in common use in the 1970s and 1980s) the same instruction would be written as

```
if intRate > 100 then write('Interest rate error');
```

In this case, the differences between the Java and Pascal versions are slight. For other constructions, there will be far more substantial differences. Compilers are language-specific. The Java compiler will translate only Java code, whereas a Pascal compiler will reject anything but legal Pascal code. For example, if a Java compiler reads the instruction `if intRate > 100 then ...`, it will complain, because the condition of the `if` statement isn't surrounded by parentheses `()` and the compiler doesn't expect the word `then`. The choice of the layout for a language construct like the `if` statement is somewhat arbitrary, and the designers of different languages choose different tradeoffs among readability, easy translation, and consistency with other languages.

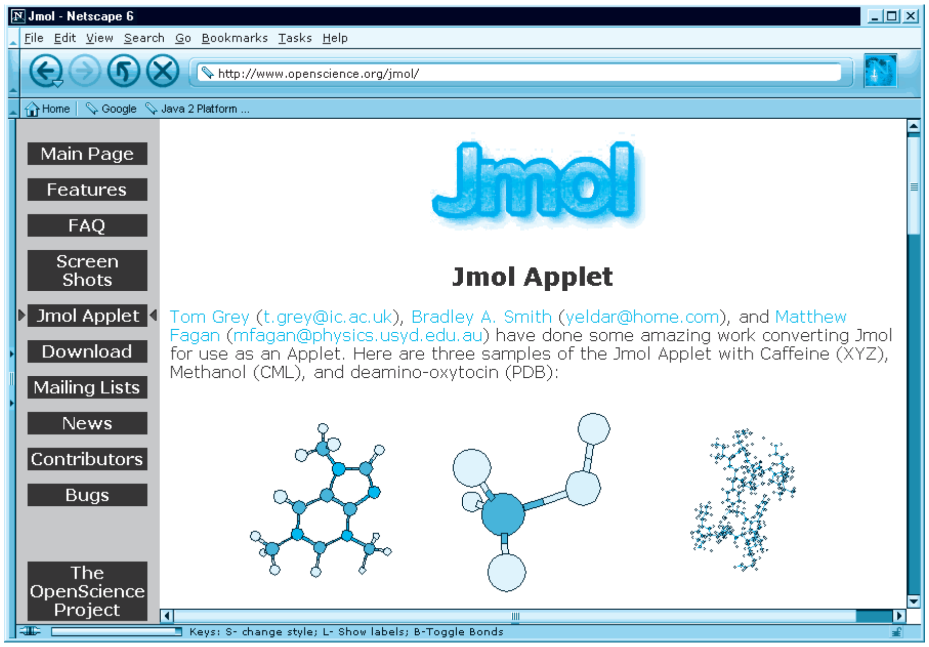
1.6 The Java Programming Language

In 1991, a group led by James Gosling and Patrick Naughton at Sun Microsystems designed a language that they code-named "Green" for use in consumer devices such as intelligent television "set-top" boxes. The language was designed to be simple and architecture-neutral, so that it could be executed on a variety of hardware. No customer was ever found for this technology.

Java was originally designed for programming consumer devices, but it was first successfully used to write Internet applets.

Gosling recounts that in 1994 the team realized, "We could write a really cool browser. It was one of the few things in the client/server mainstream that needed some of the weird things we'd done: architecture neutral, real-time, reliable, secure." The HotJava browser, which was shown to an enthusiastic crowd at the SunWorld exhibition in 1995, had one unique property: It could download programs, called *applets*, from the web and run them. Applets, written in the language now called Java, let web developers provide a variety of animation and interaction that can greatly extend the capabilities of a web page (see Figure 7). Since 1996, both Netscape and Microsoft have supported Java in their browsers.

Java has grown at a phenomenal rate. Programmers have embraced the language because it is simpler than its closest rival, C++. In addition to the programming language itself, Java has a rich *library* that makes it possible to write portable programs that can



Jmol Applet

Tom Grey (t.grey@ic.ac.uk), Bradley A. Smith (yeldar@home.com), and Matthew Fagan (mfagan@physics.usyd.edu.au) have done some amazing work converting Jmol for use as an Applet. Here are three samples of the Jmol Applet with Caffeine (XYZ), Methanol (CML), and deamino-oxytocin (PDB):

Keys: S- change style; L- Show labels; B-Toggle Bonds

Figure 7

An Applet for Visualizing Molecules (`http://www.openscience.org/jmol`)

bypass proprietary operating systems—a feature that was eagerly sought by those who wanted to be independent of those proprietary systems and was bitterly fought by their vendors.

Some of the early expectations that were placed on the Java language were overly optimistic, and the slogan "write once, run anywhere" turned into "write once, debug everywhere" for the early adopters of Java, who had to deal with less-than-perfect implementations. Since then, Java has come a long way. The Java 2 language and library, released in 1998, has brought a much greater level of stability to Java development. A "micro edition" and an "enterprise edition" of the Java library make Java programmers at home on hardware ranging from the smallest embedded devices to the largest Internet servers.

Java was designed to be safe and portable, benefitting both Internet users and students.

Because Java was designed for the Internet, it has two attributes that make it very suitable for beginners: safety and portability. If you visit a web page that contains applets, those applets automatically start running. It is important that you can trust that applets are inherently safe. If an applet could do something evil, such as damaging data or reading personal information on your computer, then you would be in real danger every time you browsed the Web—an unscrupulous designer might put up a web page containing dangerous code that would execute on your machine as soon as you visited the page. The Java language has an assortment of security features that guarantee that no evil applets can run on your computer. As an added benefit, these features also help you to learn the language faster. The Java virtual machine can catch many kinds of beginners' mistakes and report them accurately. (In contrast, many beginners' mistakes in the C language merely produce programs that act in random and confusing ways.) The other benefit of Java is portability. The same Java program will run, without change, on Windows, UNIX, Linux, or the Macintosh. This too is a requirement for applets. When you visit a web page, the web server that serves up the page contents has no idea what computer you are using to browse the Web. It simply returns you the portable code that was generated by the Java compiler. The virtual machine on your computer executes that portable code. Again, there is a benefit for the student. You do not have to learn how to write programs for different computers' operating systems.

At this time, Java has already established itself as one of the most important languages for general-purpose programming as well as for computer science instruction. However, although Java is a good language for beginners, it is not perfect, for two reasons.

Because Java was not specifically designed for students, no thought was given to make it really simple to write basic programs. A certain amount of technical machinery is necessary in Java to write even the simplest programs. To understand what this technical machinery does, you need to know something about programming. This is not a problem for a professional programmer with prior experience in another programming language, but not having a linear learning path is a drawback for the student. As you learn how to program in Java, there will be times when you will be asked to be satisfied with a preliminary explanation and wait for complete details in a later chapter.

Java has a very large library. Focus on learning those parts of the library that you need for your programming projects.

Furthermore, you cannot hope to learn all of Java in one semester. The Java language itself is relatively simple, but Java contains a vast set of *library packages* that are necessary to write useful programs. There are packages for graphics, user interface design, cryptography, networking, sound, database storage, and many other purposes. Even expert Java programmers do not know the contents of all of the packages—they just use those that they need for particular projects. Using this book, you should expect to learn a good deal about the Java language and about the most important packages. Keep in mind that the central goal of this book is not to make you memorize Java minutiae, but to teach you how to think about programming.

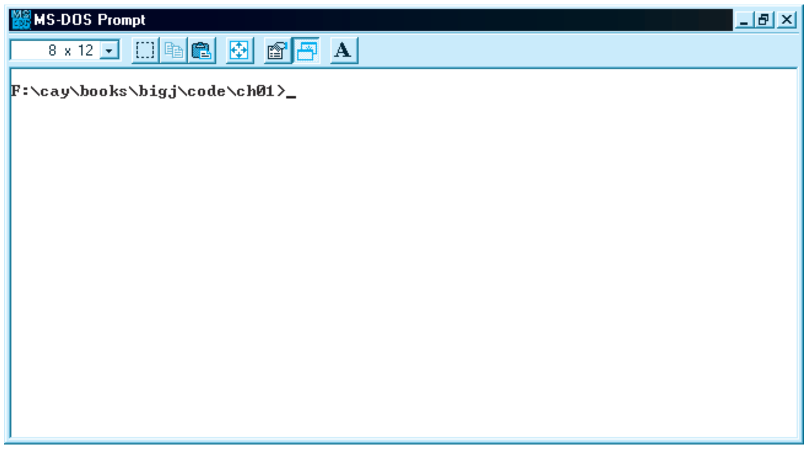
1.7 Becoming Familiar with Your Computer

Set aside some time to become familiar with the computer system and the Java compiler that you will use for your class work.

You may be taking your first programming course as you read this book, and you may well be doing your work on an unfamiliar computer system. You should spend some time making yourself familiar with the computer. Because computer systems vary widely, this book can only give an outline of the steps you need to follow. Using a new and unfamiliar computer system can be frustrating, especially if you are on your own. Look for training courses that your campus offers, or just ask a friend to give you a brief tour.

Step 1. Log In

If you use your own home computer, you probably don't need to worry about this step. Computers in a lab, however, are usually not open to everyone. Access is usually restricted



```
F:\cay\books\bigj\code\ch01>_
```

Figure 8

A Shell Window

to those who have paid the necessary fees, and often each student account has permissions and restrictions that enable the student to do class work but not mess up the system for others. You will need an account name or number and a password to gain access to such a system.

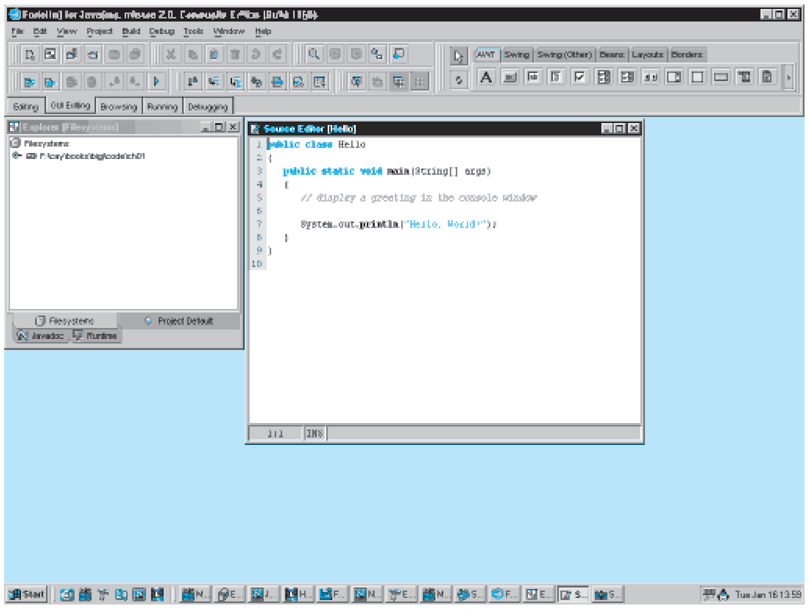
Step 2. Locate the Java Compiler

Computer systems differ greatly in this regard. On some systems you must open a *shell window* (see Figure 8) and type commands to launch the compiler. Other systems have an *integrated development environment* in which you can write and test your programs (see Figure 9). Many university labs have information sheets and tutorials that walk you through the tools that are installed in the lab. The companion web site for this book (reference [1] at the end of this chapter) contains instructions for several popular compilers.

Step 3. Understand Files and Folders

As a programmer, you will write Java programs, try them out, and improve them. You will be provided a place in secondary storage to store them, and you need to find out where that place is. Information in secondary storage is kept in *files*. A file is a collection of items of information that are kept together, such as the text of a word processing document or the instructions of a Java program. Files have names, and the rules for legal names differ from one system to another. Some systems allow spaces in file names; others don't. Some distinguish between upper- and lowercase letters; others don't. Most Java compilers require that Java files end in an *extension* `.java`; for example, `Test.java`. Java file names cannot contain spaces, and the distinction between upper- and lowercase letters is important.

Files are stored in *folders* or *directories*. These file containers can be *nested*. That is, a folder can contain not only files but also other folders, which themselves can contain



```
public class Hello
{
   public static void main(String[] args)
   {
      // display a greeting in the console window

      System.out.println("Hello, World!");
   }
}
```

Figure 9

An Integrated Development Environment

more files and folders (see Figure 10). This hierarchy can be quite large, especially on networked computers, where some of the files may be on your local disk, others elsewhere on the network. While you need not be concerned with every branch of the hierarchy, you should familiarize yourself with your local environment. Different systems have different ways of showing files and directories. Some use a graphical display and let you move around by clicking the mouse on folder icons. In other systems, you must enter commands to visit or inspect different locations.

Step 4. Write a Simple Program

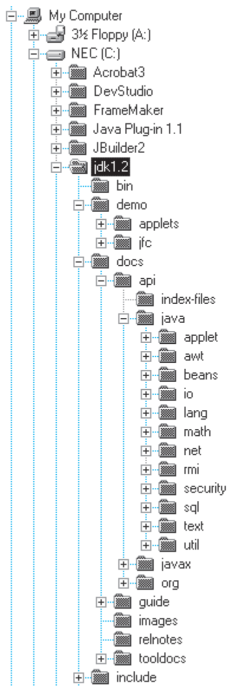
In the next section, we will introduce a very simple program. You will need to learn how to type it in, how to run it, and how to fix mistakes.

Step 5. Save Your Work

You will spend many hours typing Java program code and improving it. The resulting program files have some value, and you should treat them as you would other important property. A conscientious safety strategy is particularly important for computer files. They are more fragile than paper documents or other more tangible objects. It is easy to delete a file by accident, and occasionally files are lost because of a computer malfunction. Unless you kept another copy, you must then retype the contents. Because you probably won't remember the entire file, you will likely find yourself spending almost as much time again as you did to enter and improve it in the first place. This costs time, and it may cause you to miss deadlines. It is therefore crucially important that you learn how

Figure 10

Nested Folders



to safeguard files and that you get in the habit of doing so *before* disaster strikes. You can make safety or *backup* copies of files by saving copies on a floppy, into another folder, or to a different computer on your local area network or the Internet.

Productivity Hint 1.1

Backup Copies

Backing up on floppy disks is the easiest and most convenient method for most people. Another increasingly popular form of backup is Internet file storage. Here are a few pointers to keep in mind.

Develop a strategy for keeping backup copies of your work before disaster strikes.

- *Back up often.* Backing up a file takes only a few seconds, and you will hate yourself if you have to spend many hours recreating work that you could have saved easily. Back up your work once every thirty minutes, and every time before you test one of your programs.
- *Rotate backups.* Use more than one floppy disk for backups, and rotate them. That is, first back up onto the first floppy disk and put it aside. Then back up onto the second floppy disk. Then use the third, and then go back to the first. That way you always have three recent backups. Even if one of the floppy disks has a defect, you can use one of the others.

- *Back up source files only.* The compiler translates the files that you write into files consisting of machine code. There is no need to back up the machine code files, since you can recreate them easily by running the compiler again. Focus your backup activity on those files that represent your effort. That way your backup disks won't fill up with files that you don't need.
- *Pay attention to the backup direction.* Backing up involves copying files from one place to another. It is important that you do this right—that is, copy from your work location to the backup location. If you do it the wrong way, you will overwrite a newer file with an older version.
- *Check your backups once in a while.* Double-check that your backups are where you think they are. There is nothing more frustrating than to find out that the backups are not there when you need them. This is particularly true if you use a backup program that stores files on an unfamiliar device (such as data tape) or in a compressed format.
- *Relax, then restore.* When you lose a file and need to restore it from backup, you are likely to be in an unhappy, nervous state. Take a deep breath and think through the recovery process before you start. It is not uncommon for an agitated computer user to wipe out the last backup when trying to restore a damaged file.

1.8 Compiling a Simple Program

You are now ready to write and run your first Java program. The traditional choice for the very first program in a new programming language is a program that displays a simple greeting: "Hello, World!". Let us follow that tradition. Here is the "Hello, World!" program in Java.

File Hello.java

```
public class Hello
{
   public static void main(String[] args)
   {
      // display a greeting in the console window

      System.out.println("Hello, World!");
   }
}
```

We will examine this program in a minute. For now, you should make a new program file and call it `Hello.java`. Enter the program instructions and compile and run the program, following the procedure that is appropriate for your compiler.

Java is case-sensitive. You must be careful about distinguishing between upper- and lowercase letters.

Java is *case-sensitive*. You must enter upper- and lowercase letters exactly as they appear in the program listing. You cannot type `MAIN` or `PrintLn`. If you are not careful, you will run into problems—see Common Error 1.1.

On the other hand, Java has *free-form layout*. You can use any number of spaces and line breaks to separate words. You can cram as many words as possible into each line,

```
public class Hello{public static void main(String[]
args){// display a greeting in the console window
System.out.println("Hello, World!");}}
```

You can even write every word and symbol on a separate line,

```
public
class
Hello
{
public
static
void
main
(
. . .
```

Lay out your programs so that they are easy to read.

However, good taste dictates that you lay out your programs in a readable fashion. Chapters 2 and 3 contain recommendations for good layout.

When you run the program, the message

```
Hello, World!
```

Classes are the fundamental building blocks of Java programs.

will appear somewhere on the screen (see Figures 11 and 12). The exact location depends on your programming environment.

Now that you have seen the program working, it is time to understand its makeup. The first line,

```
public class Hello
```

Each class contains definitions of methods. Each method contains a sequence of instructions.

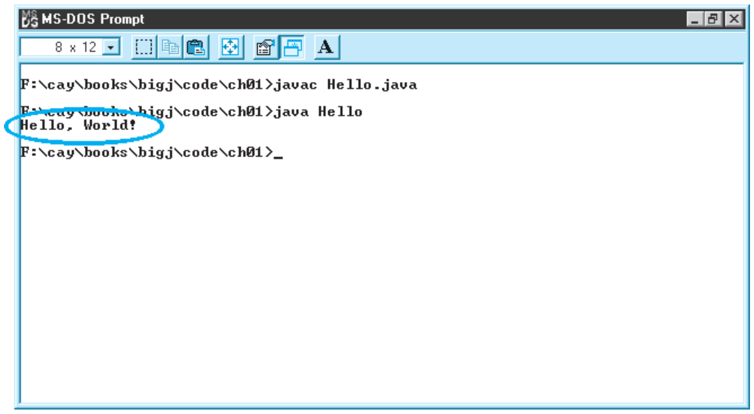
starts a new *class*. Classes are a fundamental concept in Java, and you will begin to study them in Chapter 2. In Java, every program consists of one or more classes.

The keyword `public` denotes that the class is usable by the "public". You will later encounter `private` features, which are not.

At this point, you should simply regard the

```
public class ClassName
{
   . . .
}
```

as a necessary part of the "plumbing" that is required to write any Java program. In Java, every source file can contain at most one public class, and the name of the public class must match the name of the file containing the class. For example, the class `Hello` *must be* contained in a file Hello.java. It is very important that the names *and the capitalization* match exactly. You can get strange error messages if you call the class `HELLO` or the file `hello.java`.



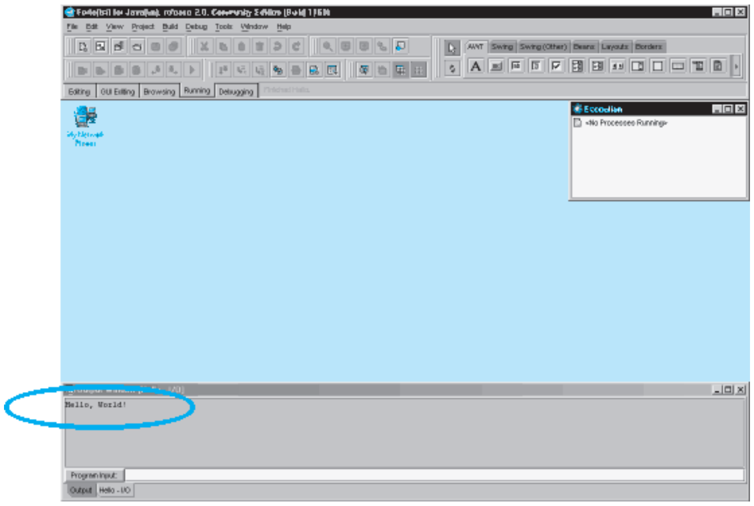
```
F:\cay\books\bigj\code\ch01>javac Hello.java

F:\cay\books\bigj\code\ch01>java Hello
Hello, World!

F:\cay\books\bigj\code\ch01>_
```

Figure 11

Running the `Hello` Program in a Console Window



```
Hello, World!
```

Figure 12

Running the `Hello` Program in an Integrated Development Environment

The construction

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

defines a *method* called `main`. A method contains a collection of programming instructions that describe how to carry out a particular task. Every Java application must have a `main` method. Most Java programs contain other methods besides `main`, and you will see in Chapter 2 how to write other methods.

Every Java application contains a class with a `main` method. When the application starts, the instructions in the `main` method are executed.

The *parameter* `String[] args` is a required part of the `main` method. (It contains *command line arguments*, which we will not discuss until Chapter 16.) The keyword `static` indicates that the `main` method does not operate on an *object*. (As you will see in Chapter 2, most methods in Java do operate on objects, and `static` methods are not common in large Java programs. Nevertheless, `main` must always be `static`, because it starts running before the program can create objects.)

At this time, simply consider

```
public class ClassName
{
   public static void main(String[] args)
   {
      . . .
   }
}
```

as yet another part of the "plumbing". Our first program has all instructions inside the `main` method of a class.

Use comments to help human readers understand your program.

The first line inside the `main` method is a *comment*

```
// display a greeting in the console window
```

This comment is purely for the benefit of the human reader, to explain in more detail what the next statement does. Any text enclosed between `//` and the end of the line is completely ignored by the compiler. Comments are used to explain the program to other programmers or to yourself.

The instructions or *statements* in the *body* of the `main` method—that is, the statements inside the curly braces `{}`—are executed one by one. Each statement ends in a semicolon `;`. Our method has a single statement:

```
System.out.println("Hello, World!");
```

This statement prints a line of text, namely "Hello, World!". However, there are many places where a program can send that string: to a window, to a file, or to a networked computer on the other side of the world. You need to specify that the destination for the string is the *standard output*—that is, a console window. The console window is represented in Java by an object called `out`. Just as you needed to place the `main` method in a `Hello` class, the designers of the Java library needed to place the `out` object into a class. They placed it in the `System` class, which contains useful objects and methods to access system resources. To use the `out` object in the `System` class, you must refer to it as `System.out`.

You call a method by specifying an object, the method name, and the method parameters.

To use an object such as `System.out`, you specify what you want to do to it. In this case, you want to print a line of text. The `println` method carries out this task. You do not have to implement this method—the programmers who wrote the Java library already did that for us—but you do need to *call* the method.

Whenever you call a method in Java, you need to specify three items:

1. The object that you want to use (in this case, `System.out`)
2. The name of the method you want to use (in this case, `println`)
3. A pair of parentheses, containing any other information the method needs (in this case, `("Hello, World!")`)

Note that the two periods in `System.out.println` have different meanings. The first period means "locate the `out` object in the `System` class". The second period means "apply the `println` method to that object".

Syntax 1.1 : Method Call

object.methodName (parameters)

Example:

```
System.out.println("Hello, Dave!");
```

Purpose:

To invoke a method on an object and supply any additional parameters

A sequence of characters enclosed in quotation marks

```
"Hello, World!"
```

is called a *string*. You must enclose the contents of the string inside quotation marks so that the compiler knows you literally mean `"Hello, World!"`. There is a reason for this requirement. Suppose you need to print the word *main*. By enclosing it in quotation marks, `"main"`, the compiler knows you mean the sequence of characters `m a i n`, not the method named `main`. The rule is simply that you must enclose all text strings in quotation marks, so that the compiler considers them plain text and does not try to interpret them as program instructions.

A string is a sequence of characters enclosed in quotation marks.

You can also print numerical values. For example, the statement

```
System.out.println(3 + 4);
```

displays the number 7.

The `println` method prints a string or a number and then starts a new line. For example, the sequence of statements

```
System.out.println("Hello");
System.out.println("World!");
```

prints two lines of text:

```
Hello
World!
```

There is a second method, called `print`, that you can use to print an item without starting a new line afterward. For example, the output of the two statements

```
System.out.print("00");
System.out.println(3 + 4);
```

is the single line

```
007
```

Common Error 1.1

Omitting Semicolons

In Java every statement must end in a semicolon. Forgetting to type a semicolon is a common error. It confuses the compiler, because the compiler uses the semicolon to find where one statement ends and the next one starts. The compiler does not use line breaks or closing braces to recognize the end of statements. For example, the compiler considers

```
System.out.println("Hello")
System.out.println("World!");
```

a single statement, as if you had written

```
System.out.println("Hello") System.out.println("World!");
```

Then it doesn't understand that statement, because it does not expect the word `System` following the closing parenthesis after `"Hello"`. The remedy is simple. Scan every statement for a terminating semicolon, just as you would check that every English sentence ends in a period.

Advanced Topic 1.1

Alternative Comment Syntax

In Java there are two methods for writing comments. You already learned that the compiler ignores anything that you type between `//` and the end of the current line. The compiler also ignores any text between a `/*` and `*/`.

```
/* A simple Java program */
```

The `//` comment is easier to type if the comment is only a single line long. If you have a comment that is longer than a line, then the `/* . . . */` comment is simpler:

```
/*
This is a simple Java program that you can use to try out
your compiler and interpreter.
*/
```

It would be somewhat tedious to add the `//` at the beginning of each line and to move them around whenever the text of the comment changes.

In this book, we use `//` for comments that will never grow beyond a line, and `/* . . . */` for longer comments. If you prefer, you can always use the `//` style. The readers of your code will be grateful for *any* comments, no matter which style you use.

Advanced Topic 1.2

Escape Sequences

Suppose you want to display a string containing quotation marks, such as

```
Hello, "World"!
```

You can't use

```
System.out.println("Hello, "World"!");
```

As soon as the compiler reads `"Hello, "`, it thinks the string is finished, and then it gets all confused about `World` followed by two quotation marks. A human would probably realize that the second and third quotation marks were supposed to be part of the string, but compilers have a one-track mind. If a simple analysis of the input doesn't make sense to them, they just refuse to go on, and they report an error. Well, how do you then display quotation marks on the screen? You precede the quotation marks inside the string with a *backslash* character. Inside a string, the sequence `\"` denotes a literal quote, not the end of a string. The correct display statement is therefore

```
System.out.println("Hello, \"World\"!");
```

The backslash character is used as an *escape* character, and the character sequence `\"` is called an escape sequence. The backslash does not denote itself; instead, it is used to encode other characters that would otherwise be difficult to include in a string.

Now, what do you do if you actually want to print a backslash (for example, to specify a Windows file name)? You must enter two `\\` in a row, like this:

```
System.out.println(
   "The secret message is in C:\\Temp\\Secret.txt");
```

This statement prints

```
The secret message is in C:\Temp\Secret.txt
```

Another escape sequence occasionally used is `\n`, which denotes a *newline* or line feed character. Printing a newline character causes the start of a new line on the display. For example, the statement

```
System.out.print("*\n**\n***\n");
```

prints the characters

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

on three separate lines. Of course, you could have achieved the same effect with three separate calls to `println`.

Finally, escape sequences are useful for including international characters in a string. For example, suppose you want to print "All the way to San José!", with an accented letter é. If you use a U.S. keyboard, you may not have a key to generate that letter. Java uses the *Unicode* encoding scheme to denote international characters. For example, the é character has Unicode encoding 00E9. You can include that character inside a string by writing `\u`, followed by its Unicode encoding:

```
System.out.println("All the way to San Jos\u00E9!");
```

You can look up the codes for the U.S. English and Western European characters in Appendix A6, and codes for thousands of characters in reference [2].

1.9 Errors

Experiment a little with the Hello program. What happens if you make a typing error such as

```
System.ouch.println("Hello, World!");
System.out.println("Hello, World!);
System.out.println("Hell, World!");
```

A syntax error is a violation of the rules of the programming language. The compiler detects syntax errors.

In the first case, the compiler will complain. It will say that it has no clue what you mean by `ouch`. The exact wording of the error message is dependent on the compiler, but it might be something like “Undefined symbol ouch”. This is a *compile-time error* or *syntax error.* Something is wrong according to the language rules, and the compiler finds it. When the compiler finds one or more errors, it refuses to translate the program to Java virtual machine instructions, and as a consequence you have no program that you can run. You must fix the error and compile again. In fact, the compiler is quite picky, and it is common to go through several rounds of fixing compile-time errors before compilation succeeds for the first time.

If the compiler finds an error, it will not simply stop and give up. It will try to report as many errors as it can find, so you can fix them all at once. Sometimes, however, one error throws it off track. This is likely to happen with the error in the second line. Because the closing quotation mark is missing, the compiler will think that the `);` characters are still part of the string. In such cases, it is common for the compiler to emit bogus error reports for neighboring lines. You should fix only those error messages that make sense to you and then recompile.

The error in the third line is of a different kind. The program will compile and run, but its output will be wrong. It will print

```
Hell, World!
```

A logic error causes a program to take an action that the programmer did not intend. You must test your programs to find logic errors.

This is a *run-time error* or *logic error.* The program is syntactically correct and does something, but it doesn’t do what it is supposed to do. The compiler cannot find the error. You, the programmer, must flush out this type of error. Run the program, and carefully look at its output.

During program development, errors are unavoidable. Once a program is longer than a few lines, it requires superhuman concentration to enter it correctly without slipping up once. You will find yourself omitting semicolons or quotes more often than you would like, but the compiler will track down these problems for you.

Logic errors are more troublesome. The compiler will not find them—in fact, the compiler will cheerfully translate any program as long as its syntax is correct—but the resulting program will do something wrong. It is the responsibility of the program author to test the program and find any logic errors. Testing programs is an important topic that you will encounter many times in this book. Another important aspect of good craftsmanship is *defensive programming:* structuring programs and development processes in such a way that an error in one place of a program does not trigger a disastrous response.

The error examples that you saw so far were not difficult to diagnose or fix, but as you learn more sophisticated programming techniques, there will also be much more room for error. It is an uncomfortable fact that locating all errors in a program is very difficult. Even if you can observe that a program exhibits faulty behavior, it may not at all be obvious what part of the program caused it and how you can fix it. Special software tools (so-called *debuggers*) let you trace through a program to find *bugs*—that is, logic errors. In this course you will learn how to use a debugger effectively.

Note that all these errors are different from the kind of errors that you are likely to make in calculations. If you total up a column of numbers, you may miss a minus sign or accidentally drop a carry, perhaps because you are bored or tired. Computers do not make these kinds of errors. When a computer adds up numbers, it will get the correct answer. Admittedly, computers can make overflow or roundoff errors, just as pocket calculators do when you ask them to perform computations whose result falls outside their numeric range. An overflow error occurs if the result of a computation is very large or very small. For example, most computers and pocket calculators overflow when you try to compute 10^{1000}. A roundoff error occurs when a value cannot be represented precisely. For example, 1/3 may be stored in the computer as 0.3333333, a value that is close to, but not exactly equal to, 1/3. If you compute $1 - (3 \times 1/3)$, you may obtain 0.0000001, not 0, as a result of the roundoff error. We will consider such errors logic errors, because the programmer should have chosen a more appropriate calculation scheme that handles overflow or roundoff correctly.

You will learn a three-part error management strategy in this book. First, you will learn about common errors and how to avoid them. Then you will learn defensive programming strategies to minimize the likelihood and impact of errors. Finally, you will learn debugging strategies to flush out those errors that remain.

Common Error 1.2

Misspelling Words

If you accidentally misspell a word, then strange things may happen, and it may not always be completely obvious from the error messages what went wrong. Here is a good example of how simple spelling errors can cause trouble:

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

This class defines a method called `Main`. The compiler will not consider this to be the same as the `main` method, because `Main` starts with an uppercase letter and the Java language is *case-sensitive*. Upper- and lowercase letters are considered to be completely different from each other, and to the compiler `Main` is no better match for `main` than `rain`. The compiler will cheerfully compile your `Main` method, but when the Java interpreter reads the compiled file, it will complain about the missing `main` method and refuse to run the program. Of course, the message “missing main method” should give you a clue where to look for the error.

If you get an error message that seems to indicate that the compiler is on the wrong track, it is a good idea to check for spelling and capitalization. All Java keywords use only lowercase letters. Names of classes usually start with an uppercase letter, names of methods and variables with a lowercase letter. If you misspell the name of a symbol (for example, `ouch` instead of `out`), the compiler will complain about an “undefined symbol”. That error message is usually a good clue that you made a spelling error.

1.10 The Compilation Process

Some Java development environments are very convenient to use. You just enter the code in one window, click on a button to compile, and click on another button to execute your program. Error messages show up in a second window, and the program runs in a third window. With such an environment you are completely shielded from the details of the compilation process. On other systems you must carry out every step manually, by typing commands such as

```
edit Hello.java
javac Hello.java
java Hello
```

into a shell window.

An editor is a program for entering and modifying text, such as a Java program.

No matter which compilation environment you use, you begin your activity by typing in the program statements. The program that you use for entering and modifying the program text is called an *editor*. Remember to *save* your work to disk frequently, because otherwise the text editor stores the text only in the computer's memory. If something goes wrong with the computer and you need to restart it, the contents of the primary memory (including your program text) are lost, but anything stored on a hard disk or floppy disk is permanent even if you need to restart the computer.

The Java compiler translates source code into instructions for the Java virtual machine, called bytecode.

When you compile your program, the compiler translates the Java *source code* (that is, the statements that you wrote) into *bytecode*, which consists of virtual machine instructions and some other pieces of information on how to load the program into memory prior to execution. The bytecode for a program is stored in a separate file, with extension `.class`. For example, the bytecode for the Hello program will be stored in a file `Hello.class`. However, the compiler produces a class file only after you have corrected all syntax errors.

The class file contains the translation of only the instructions that you wrote. That is not enough actually to run the program. To display a string on a window, quite a bit of low-level activity is necessary. The authors of the `System` and `PrintStream` classes (which define the `out` object and the `println` method) have implemented all necessary actions and placed the required bytecode into a *library*. A library is a collection of code that has been programmed and translated by someone else, ready for you to use in your program. (More complicated programs are built from more than one class file and more than one library.)

The *Java interpreter* loads the bytecode of the program that you wrote, starts your program, and loads the necessary library files as they are required.

The Java interpreter runs a program, loading the necessary bytecode from class files and library files.

The steps of compiling and running your program are outlined in Figure 13.

Your programming activity centers around these steps. You start in the editor, writing the source file. You compile the program and look at the error messages. You go back to the editor and fix the syntax errors. When the compiler succeeds, you run your program. If you find an error, you can run the debugger to execute it a line at a time. Once you find the cause of the error, you go back to the editor and fix it. You compile and run again to see whether the error has gone away. If not, you go back to the editor. This is called the *edit–compile–test loop* (see Figure 14). You will spend a substantial amount of time in this loop whenever you work on programming assignments.

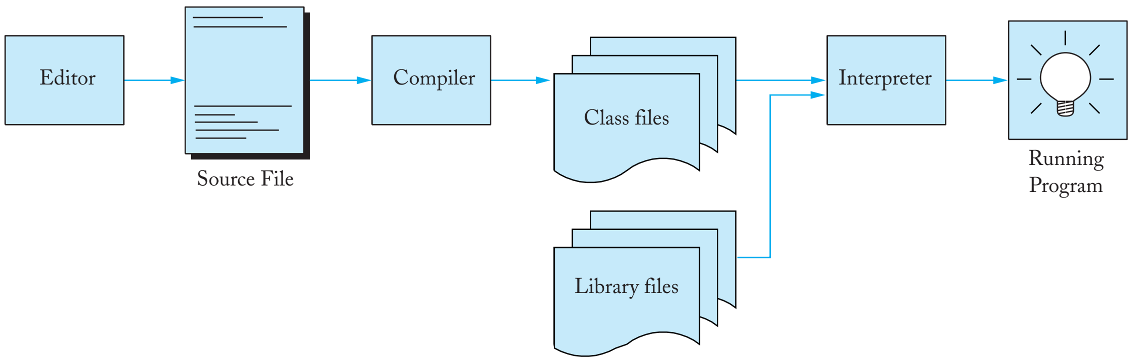


Figure 13

From Source Code to Running Program

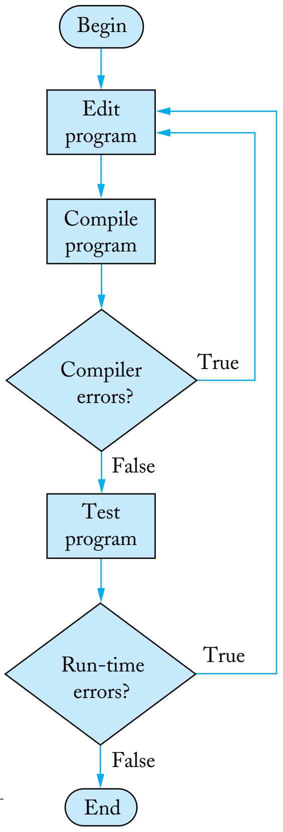


Figure 14

The Edit-Compile-Test Loop

Chapter Summary

1. A computer must be programmed to perform tasks. Different tasks require different programs.
2. A computer program executes a sequence of very basic operations in rapid succession.
3. Programmers develop computer programs to make computers perform new tasks.
4. At the heart of the computer lies the central processing unit (CPU).
5. Data and programs are stored in primary storage (memory) and secondary storage (such as a hard disk).
6. The CPU reads machine instructions from memory. The instructions direct it to communicate with memory, secondary storage, and peripheral devices.
7. Generally, machine code depends on the CPU type. However, the instruction set of the Java virtual machine (JVM) can be executed on many CPUs.
8. Because machine instructions are encoded as numbers, it is difficult to write programs in machine code.

9. Assembly language makes it easier to generate machine instructions by translating mnemonics and symbolic names.
10. High-level languages let you describe tasks at a higher conceptual level than machine code.
11. A compiler translates programs written in a high-level language into machine code.
12. Each programming language has its own set of rules for forming instructions. Compilers enforce these rules strictly.
13. Java was originally designed for programming consumer devices, but it was first successfully used to write Internet applets.
14. Java was designed to be safe and portable, benefitting both Internet users and students.
15. Java has a very large library. Focus on learning those parts of the library that you need for your programming projects.
16. Set aside some time to become familiar with the computer system and the Java compiler that you will use for your class work.
17. Develop a strategy for keeping backup copies of your work before disaster strikes.
18. Java is case-sensitive. You must be careful about distinguishing between upper- and lowercase letters.
19. Lay out your programs so that they are easy to read.
20. Classes are the fundamental building blocks of Java programs.
21. Each class contains definitions of methods. Each method contains a sequence of instructions.
22. Every Java application contains a class with a `main` method. When the application starts, the instructions in the `main` method are executed.
23. Use comments to help human readers understand your program.
24. You call a method by specifying an object, the method name, and the method parameters.
25. A string is a sequence of characters enclosed in quotation marks.
26. A syntax error is a violation of the rules of the programming language. The compiler detects syntax errors.
27. A logic error causes a program to take an action that the programmer did not intend. You must test your programs to find logic errors.
28. An editor is a program for entering and modifying text, such as a Java program.
29. The Java compiler translates source code into instructions for the Java virtual machine, called bytecode.
30. The Java interpreter runs a program, loading the necessary bytecode from class files and library files.

Further Reading

[1] http://www.horstmann.com/bigjava/help/compilers.html Instructions for using several popular Java compilers.
[2] http://www.unicode.org/ The web site of the Unicode Consortium. It contains character tables that show the Unicode values of characters from many scripts.

Classes, Objects, and Methods Introduced in This Chapter

Here is a list of all classes, methods, static variables, and constants introduced in this chapter. Turn to the documentation in Appendix A6 for more information.

```
java.io.PrintStream
   print
   println
java.lang.String
   length
java.lang.System
   out
```

Review Exercises

Exercise R1.1. Explain the difference between using a computer program and programming a computer.

Exercise R1.2. What distinguishes a computer from a typical household appliance?

Exercise R1.3. Rank the storage devices that can be part of a computer system by (*a*) speed, (*b*) cost, and (*c*) storage capacity.

Exercise R1.4. What is the Java virtual machine?

Exercise R1.5. What is an applet?

Exercise R1.6. Explain two benefits of higher-level programming languages over assembler code.

Exercise R1.7. List the programming languages mentioned in this chapter.

Exercise R1.8. What is an integrated programming environment?

Exercise R1.9. What is a console window?

Exercise R1.10. Describe *exactly* what steps you would take to back up your work after you have typed in the `Hello.java` program.

Exercise R1.11. On your own computer or on your lab computer, find the exact location (folder or directory name) of

- The sample file `Hello.java`, which you wrote with the editor
- The Java interpreter `java.exe`
- The library file `rt.jar` that contains the runtime library

Exercise R1.12. Explain the special role of the `\` escape character in Java character strings.

Exercise R1.13. Write three versions of the `Hello.java` program that have different syntax errors. Write a version that has a logic error.

Exercise R1.14. How do you discover syntax errors? How do you discover logic errors?

Programming Exercises

Exercise P1.1. Write a program that displays your name inside a box on the console screen, like this:

```
+----+
|Dave|
+----+
```

Do your best to approximate lines with characters like `| - +`.

Exercise P1.2. Write a program that prints a face, using text characters, hopefully better looking than this one:

```
  /////
 | o o |
(|  ^  |)
 | \_/ |
  -----
```

Use *comments* to indicate when you print the hair, ears, mouth, and so on.

Exercise P1.3. Write a program that prints a Christmas tree:

```
   /\
  /  \
 /    \
/      \
--------
  "  "
  "  "
  "  "
```

Remember to use escape sequences to print the `\` and `"` characters.

Exercise P1.4. Write a program that prints a staircase:

```
            +---+
            |   |
        +---+---+
        |   |   |
    +---+---+---+
    |   |   |   |
+---+---+---+---+
|   |   |   |   |
+---+---+---+---+
```

Exercise P1.5. Write a program that computes the sum of the first ten positive integers, 1 + 2 + · · · + 10. *Hint:* Write a program of the form

```
public class Sum10
{
   public static void main(String[] args)
   {
      System.out.println(           );
   }
}
```

Exercise P1.6. Write a program that computes the sum of the reciprocals 1/1 + 1/2 + · · · + 1/10. This is harder than it sounds. Try writing the program, and check the result. The program's result isn't likely to be correct. Then write the numbers as *floating-point* numbers, `1.0`, `2.0`, . . . , `10.0`, and run the program again. Can you explain the difference in the results? We will explore this phenomenon in Chapter 3.