



Introduction to Computers, the Internet and the Web

Objectives

- To understand basic computer science concepts.
- To become familiar with different types of programming languages.
- To introduce the Java development environment.
- To understand Java's role in developing distributed client/server applications for the Internet and Web.
- To introduce object-oriented design with the UML and design patterns.
- To preview the remaining chapters of the book.

Our life is frittered away by detail ... Simplify, simplify.

Henry Thoreau

High thoughts must have high language.

Aristophanes

The chief merit of language is clearness.

Galen

My object all sublime

I shall achieve in time.

W. S. Gilbert

*He had a wonderful talent for packing thought close, and
rendering it portable.*

Thomas Babington Macaulay

*Egad, I think the interpreter is the hardest to be understood
of the two!*

Richard Brinsley Sheridan



Outline

- 1.1 Introduction**
- 1.2 What Is a Computer?**
- 1.3 Computer Organization**
- 1.4 Evolution of Operating Systems**
- 1.5 Personal, Distributed and Client/Server Computing**
- 1.6 Machine Languages, Assembly Languages and High-Level Languages**
- 1.7 History of C++**
- 1.8 History of Java**
- 1.9 Java Class Libraries**
- 1.10 Other High-Level Languages**
- 1.11 Structured Programming**
- 1.12 The Internet and the World Wide Web**
- 1.13 Basics of a Typical Java Environment**
- 1.14 General Notes about Java and This Book**
- 1.15 Thinking About Objects: Introduction to Object Technology and the Unified Modeling Language**
- 1.16 Discovering Design Patterns: Introduction**
- 1.17 Tour of the Book**
- 1.18 (Optional) A Tour of the Case Study on Object-Oriented Design with the UML**
- 1.19 (Optional) A Tour of the “Discovering Design Patterns” Sections**

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

1.1 Introduction

Welcome to Java! We have worked hard to create what we hope will be an informative, entertaining and challenging learning experience for you. Java is a powerful computer programming language that is fun to use for novices and appropriate for experienced programmers building substantial information systems. *Java How to Program: Fourth Edition* is designed to be an effective learning tool for each of these audiences.

How can one book appeal to both groups? The answer is that the common core of the book emphasizes achieving program *clarity* through the proven techniques of *structured programming* and *object-oriented programming*. Nonprogrammers will learn programming the right way from the beginning. We have attempted to write in a clear and straightforward manner. The book is abundantly illustrated. Perhaps most importantly, the book presents hundreds of working Java programs and shows the outputs produced when those programs are run on a computer. We teach all Java features in the context of complete working Java programs. We call this the *live-code™ approach*. These examples are available from three locations—they are on the CD that accompanies this book, they may be

downloaded from our Web site www.deitel.com and they are available on our interactive CD product, the *Java 2 Multimedia Cyber Classroom: Fourth Edition*. The Cyber Classroom's features and ordering information appear at the back of this book. The Cyber Classroom also contains answers to approximately half of the solved exercises in this book, including short answers, small programs and many full projects. If you purchased *The Complete Java 2 Training Course: Fourth Edition*, you already have the Cyber Classroom.

The early chapters introduce the fundamentals of computers, computer programming and the Java computer programming language. Novices who have taken our courses tell us that the material in those chapters presents a solid foundation for the deeper treatment of Java in the later chapters. Experienced programmers tend to read the early chapters quickly and find that the treatment of Java in the later chapters is rigorous and challenging.

Many experienced programmers have told us that they appreciate our structured programming treatment. Often, they have been programming in structured languages like C or Pascal, but they were never formally introduced to structured programming, so they are not writing the best possible code in these languages. As they review structured programming in the chapters "Control Structures: Part 1" and "Control Structures: Part 2," they are able to improve their C and Pascal programming styles as well. So whether you are a novice or an experienced programmer, there is much here to inform, entertain and challenge you.

Most people are familiar with the exciting tasks computers perform. Using this textbook, you will learn how to command computers to perform those tasks. It is *software* (i.e., the instructions you write to command computers to perform *actions* and make *decisions*) that controls computers (often referred to as *hardware*), and Java is one of today's most popular software-development languages. Java was developed by Sun Microsystems and an implementation of it is available free over the Internet from the Sun Web site

java.sun.com/j2se

This book is based on the *Java 2 Platform, Standard Edition*, which describes the Java language, libraries and tools. Other vendors can implement *Java development kits* based on the *Java 2 Platform*. Sun provides an implementation of the *Java 2 Platform, Standard Edition* called the *Java 2 Software Development Kit, Standard Edition (J2SDK)* that includes the minimum set of tools you need to write software in Java. At the time of this publication, the most recent version was J2SDK 1.3.1. You can download future updates to the J2SDK from the Sun Web site java.sun.com/j2se.

Computer use is increasing in almost every field of endeavor. In an era of steadily rising costs, computing costs have been decreasing dramatically due to rapid developments in both hardware and software technology. Computers that might have filled large rooms and cost millions of dollars two decades ago can now be inscribed on the surfaces of silicon chips smaller than a fingernail, costing perhaps a few dollars each. Ironically, silicon is one of the most abundant materials on earth—it is an ingredient in common sand. Silicon-chip technology has made computing so economical that hundreds of millions of general-purpose computers are in use worldwide helping people in business, industry, government, and in their personal lives. The number of computers worldwide easily could double in the next few years.

This book will challenge you for several reasons. For many years, students learned C or Pascal as their first programming language. They probably learned the programming methodology called *structured programming*. You will learn both structured programming and the exciting newer methodology, *object-oriented programming*. Why do we teach both? We

believe that object orientation is the key programming methodology of the future. You will build and work with many *objects* in this course. However, you will discover that the internal structure of those objects is built with structured programming techniques. Also, the logic of manipulating objects is occasionally best expressed with structured programming.

Another reason we present both methodologies is the continuing migration from C-based systems (built primarily with structured programming techniques) to C++ and Java-based systems (built primarily with object-oriented programming techniques). There is a huge amount of so-called “legacy C code” in place, because C has been in use for over three decades. Once people learn C++ or Java, they find these languages to be more powerful than C. These people often choose to move their programming projects to C++ or Java. They begin converting their legacy systems and begin employing the object-oriented programming capabilities of C++ or Java to realize the full benefits of these languages. Often, the choice between C++ and Java is made based on the simplicity of Java compared to C++.

Java has become the language of choice for implementing Internet-based and Intranet-based applications and software for devices that communicate over a network. Do not be surprised when your new stereo and other devices in your home will be networked together by Java technology! Also, do not be surprised when your wireless devices, like cell phones, pagers and personal digital assistants (PDAs) communicate over the so-called Wireless Internet via the kind of Java-based networking protocols that you will learn in this book and its companion *Advanced Java 2 Platform How to Program*.

Java is a particularly attractive first programming language. At the JavaOne™ trade show in June 2001, it was announced that Java is now a required part of the programming languages curriculum in 56% of US colleges and universities. Also, 87% of US colleges and universities offer Java courses. Java is attractive to high schools as well. In 2003, the College Board will standardize on Java for Advanced Placement computer science courses.

Java has evolved rapidly into the large-scale applications arena. Java is no longer a language used simply to make World Wide Web pages “come alive.” Java has become the preferred language for meeting many organizations’ programming needs.

For many years, languages like C and C++ appealed to universities because of their portability. Introductory courses could be offered in these languages on any hardware/operating system combination, as long as a C/C++ compiler was available. However, the programming world has become more complex and more demanding. Today, users want applications with graphical user interfaces (GUIs). They want applications that use multimedia capabilities such as graphics, images, animation, audio and video. They want applications that can run on the Internet and the World Wide Web and communicate with other applications. They want applications that can take advantage of the flexibility and performance improvements of multithreading (which enables programmers to specify that several activities should occur in parallel). They want applications with richer file processing than is provided by C or C++. They want applications that are not limited to the desktop or even to some local computer network, but can integrate Internet components and remote databases as well. They want applications that can be written quickly and correctly in a manner that takes advantage of prebuilt software components. They want easy access to a growing universe of reusable software components. Programmers want all these benefits in a truly portable manner, so that applications will run without modification on a variety of *platforms* (i.e., different types of computers running different operating systems). Java offers all these benefits to the programming community.

Another reason Java is attractive for university courses is that it is fully object oriented. One reason that C++ use has grown so quickly is that it extends C programming into the arena of object orientation. For the huge community of C programmers, this has been a powerful advantage. C++ includes ANSI/ISO C and offers the ability to do object-oriented programming as well. (ANSI is the American National Standards Institute, and ISO is the International Standards Organization.) An enormous amount of C code has been written in industry over the last several decades. C++ is a superset of C, so many organizations find it to be an ideal next step. Programmers can take their C code, compile it (often with nominal changes) in a C++ compiler and continue writing C-like code while mastering the object paradigm. Then, the programmers can gradually migrate portions of the legacy C code into C++ as time permits. New systems can be entirely written in object-oriented C++. Such strategies have been appealing to many organizations. The downside is that, even after adopting this strategy, companies tend to continue producing C-like code for many years. This, of course, means that they do not realize the benefits of object-oriented programming quickly and could produce programs that are confusing and hard to maintain as a result of their hybrid design. Many organizations would prefer to plunge 100% into object-oriented development, but the realities of mountains of legacy code and the temptation to take a C-programming approach often prevent this.

Java is a fully object-oriented language with strong support for proper software engineering techniques. It is difficult to write C-like, so-called procedural programs in Java. You must create and manipulate objects. Error processing is built into the language. Many of the complex details of C and C++ programming that prevent programmers from “looking at the big picture” are not included in Java. For universities, these features are powerfully appealing. Students will learn object-oriented programming from the start. They will simply think in an object-oriented manner.

Here, too, there is a trade-off. Organizations turning to Java for new applications development do not want to convert all their legacy code to Java. So Java allows for so-called *native code*. This means that existing C and C++ code can be integrated with Java code. Although this may seem a bit awkward (and it certainly can be), it presents a pragmatic solution to a problem most organizations face.

The fact that Java is free for download at the Sun Web site, java.sun.com/j2se, is appealing to universities facing tight budgets and lengthy budget planning cycles. Also, as bug fixes and new versions of Java are developed, these become available immediately over the Internet, so universities can keep their Java software current.

Can Java be taught in a first programming course—the intended audience for this book? We think so. Prior to writing this book, Deitel & Associates, Inc. instructors taught hundreds of Java courses to several thousand people at all levels of expertise, including many nonprogrammers. We found that nonprogrammers become productive faster with Java than with C or C++. They are anxious to experiment with Java’s powerful features for graphics, graphical user interfaces, multimedia, animation, multithreading, networking and the like—and they are successful at building substantial Java programs even in their first courses.

For many years, the Pascal programming language was the preferred vehicle for use in introductory and intermediate programming courses. Many people said that C was too difficult a language for these courses. In 1992, we published the first edition of *C How to Program*, to encourage universities to try C instead of Pascal in these courses. We used the same pedagogic approach we had used in our university courses for a dozen years, but

wrapped the concepts in C rather than Pascal. We found that students were able to handle C at about the same level as Pascal. However, there was one noticeable difference—students appreciated that they were learning a language (C) likely to be valuable to them in industry. Our industry clients appreciated the availability of C-literate graduates who could work immediately on substantial projects rather than first having to go through costly and time-consuming training programs.

The first edition of *C How to Program* included a 60-page introduction to C++ and object-oriented programming. We saw C++ coming on strong, but we felt it would be at least a few more years before the universities would be ready to teach C++ and object-oriented programming (OOP) in introductory courses.

During 1993, we saw a surge in interest in C++ and OOP among our industry clients, but we still did not sense that the universities were ready to switch to C++ and OOP en masse. So, in January 1994, we published the Second Edition of *C How to Program* with a 300-page section on C++ and OOP. In May 1994, we published the first edition of *C++ How to Program*, a 950-page book devoted to the premise that C++ and OOP were now ready for prime time in introductory university courses for many schools that wanted to be at the leading edge of programming-languages education.

In 1995, we were following the introduction of Java carefully. In November 1995, we attended an Internet conference in Boston. A representative from Sun Microsystems gave a presentation on Java that filled one of the large ballrooms at the Hynes Convention Center. As the presentation proceeded, it became clear to us that Java would play a significant part in the development of interactive, multimedia Web pages. We immediately saw a much greater potential for the language. We saw Java as the proper language for universities to teach first-year programming language students in this modern world of graphics, images, animation, audio, video, database, networking, multithreading and collaborative computing. At the time, we were busy writing the second edition of *C++ How to Program*. We discussed with our publisher, Prentice Hall, our vision of Java making a strong impact in the university curriculum. We all agreed to delay the second edition of *C++ How to Program* a bit so that we could get the first edition of *Java How to Program* (based on Java 1.0.2) to the market in time for fall 1996 courses.

As Java rapidly evolved to Java 1.1, we wrote *Java How to Program: Second Edition* in 1997, less than a year after the first edition reached bookstores. Hundreds of universities and corporate training programs worldwide used the second edition. To keep pace with the enhancements in Java, we published *Java How to Program: Third Edition* in 1999. The third edition was a major overhaul to upgrade the book to the *Java 2 Platform*.

Java continues to evolve rapidly, so we wrote this fourth edition of *Java How to Program*—our first book to reach a fourth edition—just five years after the first edition was published. This edition is based on the *Java 2 Platform, Standard Edition (J2SE)*. Java has grown so rapidly over the last several years that it now has two other editions. The *Java 2 Platform, Enterprise Edition (J2EE)* is geared toward developing large-scale, distributed networking applications and Web-based applications. The *Java 2 Platform, Micro Edition (J2ME)* is geared toward development of applications for small devices (such as cell phones, pagers and personal digital assistants) and other memory-constrained applications. The number of topics to cover in Java has become far too large for one book. So, in parallel with *Java How to Program, Fourth Edition*, we are publishing *Advanced Java 2 Platform How to Program*, which emphasizes developing applications with J2EE and provides cov-

erage of several high-end topics from the J2SE. In addition, this book also includes substantial materials on J2ME and wireless-application development.

So, there you have it! You are about to start on a challenging and rewarding path. As you proceed, please share your thoughts on Java and *Java How to Program: Fourth Edition* with us via e-mail at deitel@deitel.com. We will respond promptly.

Prentice Hall maintains www.prenhall.com/deitel—a Web site dedicated to our Prentice Hall publications, including textbooks, professional books, interactive multimedia CD-based *Cyber Classrooms*, *Complete Training Courses* (boxed products containing both a *Cyber Classroom* and the corresponding book), Web-based training, e-whitepapers, e-books and ancillary materials for all these products. For each of our books, the site contains companion Web sites that include frequently asked questions (FAQs), code downloads, errata, updates, additional text and examples, additional self-test questions and new developments in programming languages and object-oriented programming technologies. If you would like to learn more about the authors or Deitel & Associates, Inc. please visit www.deitel.com. Good luck!

1.2 What Is a Computer?

A *computer* is a device capable of performing computations and making logical decisions at speeds millions, even billions, of times faster than human beings can. For example, many of today's personal computers can perform hundreds of millions, even billions, of additions per second. A person operating a desk calculator might require decades to complete the same number of calculations a powerful personal computer can perform in one second. (*Points to ponder:* How would you know whether the person added the numbers correctly? How would you know whether the computer added the numbers correctly?) Today's fastest *supercomputers* can perform hundreds of billions of additions per second—about as many calculations as hundreds of thousands of people could perform in one year! And trillion-instruction-per-second computers are already functioning in research laboratories!

Computers process *data* under the control of sets of instructions called *computer programs*. These programs guide the computer through orderly sets of actions specified by people called *computer programmers*.

The various devices that comprise a computer system (such as the keyboard, screen, disks, memory and processing units) are referred to as *hardware*. The computer programs that run on a computer are referred to as *software*. Hardware costs have been declining dramatically in recent years, to the point that personal computers have become a commodity. Unfortunately, software-development costs have been rising steadily, as programmers develop ever more powerful and complex applications without being able to improve significantly the technology of software development. In this book, you will learn proven software-development methods that can reduce software-development costs—top-down stepwise refinement, functionalization and object-oriented programming. Object-oriented programming is widely believed to be the significant breakthrough that can greatly enhance programmer productivity.

1.3 Computer Organization

Regardless of differences in physical appearance, virtually every computer may be envisioned as being divided into six *logical units* or sections. These are as follows:

1. *Input unit.* This is the “receiving” section of the computer. It obtains information (data and computer programs) from *input devices* and places this information at the disposal of the other units so that the information may be processed. Most information is entered into computers today through typewriter-like keyboards, “mouse” devices and disks. In the future, most information will be entered by speaking to computers, by electronically scanning images and by video recording.
2. *Output unit.* This is the “shipping” section of the computer. It takes information processed by the computer and places it on various *output devices* to make the information available for use outside the computer. Information output from computers is displayed on screens, printed on paper, played through audio speakers, magnetically recorded on disks and tapes or used to control other devices.
3. *Memory unit.* This is the rapid-access, relatively low-capacity “warehouse” section of the computer. It retains information that has been entered through the input unit so that the information may be made immediately available for processing when it is needed. The memory unit also retains information that has already been processed until that information can be placed on output devices by the output unit. The memory unit often is called either *memory*, *primary memory* or *random-access memory (RAM)*.
4. *Arithmetic and logic unit (ALU).* This is the “manufacturing” section of the computer. It is responsible for performing calculations such as addition, subtraction, multiplication and division. It contains the decision mechanisms that allow the computer, for example, to compare two items from the memory unit to determine whether they are equal.
5. *Central processing unit (CPU).* This is the “administrative” section of the computer. It is the computer’s coordinator and is responsible for supervising the operation of the other sections. The CPU tells the input unit when information should be read into the memory unit, tells the ALU when information from the memory unit should be utilized in calculations and tells the output unit when to send information from the memory unit to certain output devices.
6. *Secondary storage unit.* This is the long-term, high-capacity “warehousing” section of the computer. Programs or data not being used by the other units are normally placed on secondary storage devices (such as disks) until they are needed, possibly hours, days, months or even years later. Information in secondary storage takes longer to access than information in primary memory. The cost per unit of secondary storage is much less than the cost per unit of primary memory.

1.4 Evolution of Operating Systems

Early computers were capable of performing only one *job* or *task* at a time. This form of computer operation is often called single-user *batch processing*. The computer runs a single program at a time while processing data in groups or *batches*. In these early systems, users generally submitted their jobs to the computer center on decks of punched cards. Users often had to wait hours or even days before printouts were returned to their desks.

Software systems called *operating systems* were developed to help make it more convenient to use computers. Early operating systems managed the smooth transition between

jobs. This minimized the time it took for computer operators to switch between jobs and hence increased the amount of work, or *throughput*, computers could process.

As computers became more powerful, it became evident that single-user batch processing rarely utilized the computer's resources efficiently. Instead, it was thought that many jobs or tasks could be made to *share* the resources of the computer to achieve better utilization. This is called *multiprogramming*. Multiprogramming involves the "simultaneous" operation of many jobs on the computer—the computer shares its resources among the jobs competing for its attention. With early multiprogramming operating systems, users still submitted jobs on decks of punched cards and waited hours or days for results.

In the 1960s, several groups in industry and the universities pioneered *timesharing* operating systems. Timesharing is a special case of multiprogramming in which users access the computer through *terminals*, typically devices with keyboards and screens. In a typical timesharing computer system, there may be dozens or even hundreds of users sharing the computer at once. The computer does not actually run all the users' jobs simultaneously. Rather, it runs a small portion of one user's job and moves on to service the next user. The computer does this so quickly that it might provide service to each user several times per second. Thus the users' programs *appear* to be running simultaneously. An advantage of timesharing is that the user receives almost immediate responses to requests rather than having to wait long periods for results, as with previous modes of computing. Also, if a particular user is currently idle, the computer can continue to service other users rather than wait for one user.

1.5 Personal, Distributed and Client/Server Computing

In 1977, Apple Computer popularized the phenomenon of *personal computing*. Initially, it was a hobbyist's dream. Computers became economical enough for people to buy them for their own personal use. In 1981, IBM, the world's largest computer vendor, introduced the IBM Personal Computer. Almost overnight, personal computing became legitimate in business, industry and government organizations.

But these computers were "stand-alone" units—people did their work on their own machines and transported disks back and forth to share information. Although early personal computers were not powerful enough to timeshare several users, these machines could be linked together in computer networks, sometimes over telephone lines and sometimes in *local area networks (LANs)* within an organization. This led to the phenomenon of *distributed computing*, in which an organization's computing, instead of being performed strictly at some central computer installation, is distributed over networks to the sites at which the real work of the organization is performed. Personal computers were powerful enough both to handle the computing requirements of individual users and to handle the basic communications tasks of passing information back and forth electronically.

Today's most powerful personal computers are as powerful as the million-dollar machines of just a decade ago. The most powerful desktop machines—called *workstations*—provide individual users with enormous capabilities. Information is shared easily across computer networks where some computers called *file servers* offer a common store of programs and data that may be used by *client* computers distributed throughout the network (hence the term *client/server computing*). C and C++ have become and remain the languages of choice for writing operating systems. They also remain popular for writing computer networking, distributed client/server and Internet and Web applications, although

Java is now the dominant language in each of these areas. Many programmers have discovered that programming in Java helps them be more productive than programming in C or C++. Today's popular operating systems, such as UNIX, Linux, MacOS, Windows and Windows 2000, provide the kinds of capabilities discussed in this section.

1.6 Machine Languages, Assembly Languages and High-Level Languages

Programmers write instructions in various programming languages, some directly understandable by computers and others that require intermediate *translation* steps. Hundreds of computer languages are in use today. These may be divided into three general types:

1. Machine languages
2. Assembly languages
3. High-level languages

Any computer can directly understand only its own *machine language*. Machine language is the “natural language” of a particular computer. It is defined by the hardware design of that computer. Machine languages generally consist of strings of numbers (ultimately reduced to 1s and 0s) that instruct computers to perform their most elementary operations one at a time. Machine languages are *machine dependent* (i.e., a particular machine language can be used on only one type of computer). Machine languages are cumbersome for humans, as can be seen by the following section of a machine-language program that adds overtime pay to base pay and stores the result in gross pay.

```
+1300042774
+1400593419
+1200274027
```

As computers became more popular, it became apparent that machine-language programming was simply too slow and tedious for most programmers. Instead of using the strings of numbers that computers could directly understand, programmers began using English-like abbreviations to represent the elementary operations of computers. These English-like abbreviations formed the basis of *assembly languages*. *Translator programs* called *assemblers* were developed to convert assembly-language programs to machine language at computer speeds. The following section of an assembly-language program also adds overtime pay to base pay and stores the result in gross pay, but somewhat more clearly than its machine-language equivalent.

```
LOAD    BASEPAY
ADD     OVERPAY
STORE   GROSSPAY
```

Although such code is clearer to humans, it is incomprehensible to computers until translated to machine language.

Computer usage increased rapidly with the advent of assembly languages, but programming in these still required many instructions to accomplish even the simplest tasks. To speed the programming process, *high-level languages* were developed in which single statements could be written to accomplish substantial tasks. The translator programs that convert high-level language programs into machine language are called *compilers*. High-

level languages allow programmers to write instructions that look almost like everyday English and contain commonly used mathematical notations. A payroll program written in a high-level language might contain a statement such as

```
grossPay = basePay + overTimePay
```

Obviously, high-level languages are much more desirable from the programmer's standpoint than either machine languages or assembly languages. C, C++ and Java are among the most powerful and most widely used high-level programming languages.

The process of compiling a high-level language program into machine language can take a considerable amount of computer time. *Interpreter* programs were developed to execute high-level language programs directly without the need for compiling those programs into machine language. Although compiled programs execute much faster than interpreted programs, interpreters are popular in program-development environments in which programs are recompiled frequently as new features are added and errors are corrected. Once a program is developed, a compiled version can be produced to run most efficiently. As we study Java, you will see that interpreters have played an especially important part in helping Java achieve its goal of portability across a great variety of platforms.

1.7 History of C++

C++ evolved from C, which evolved from two previous languages, BCPL and B. BCPL was developed in 1967 by Martin Richards as a language for writing operating-systems software and compilers. Ken Thompson modeled many features in his language B after their counterparts in BCPL and used B to create early versions of the UNIX operating system at Bell Laboratories in 1970 on a Digital Equipment Corporation PDP-7 computer. Both BCPL and B were “typeless” languages—every data item occupied one “word” in memory. For example, it was the programmer's responsibility to treat a data item as a whole number or a real number.

The C language was evolved from B by Dennis Ritchie at Bell Laboratories and was originally implemented on a DEC PDP-11 computer in 1972. C uses many important concepts of BCPL and B while adding data typing and other features. C initially became widely known as the development language of the UNIX operating system. Today, virtually all new major operating systems are written in C or C++. Over the past two decades, C has become available for most computers. C is hardware independent. With careful design, it is possible to write C programs that are *portable* to most computers.

By the late 1970s, C had evolved into what is now referred to as “traditional C,” or “Kernighan and Ritchie C.” The publication by Prentice Hall in 1978 of Kernighan and Ritchie's book, *The C Programming Language*, brought wide attention to the language. This publication became one of the most successful computer science books ever.

The widespread use of C with various types of computers (sometimes called *hardware platforms*) led to many variations. These were similar, but often incompatible. This was a serious problem for programmers who needed to write portable programs that would run on several platforms. It became clear that a standard version of C was needed. In 1983, the X3J11 technical committee was created under the American National Standards Committee on Computers and Information Processing (X3) to “provide an unambiguous and machine-independent definition of the language.” In 1989, the standard was approved. ANSI cooperated with the International Standards Organization (ISO) to standardize C

worldwide; the joint standard document was published in 1990 and is referred to as ANSI/ISO 9899: 1990. The second edition of Kernighan and Ritchie,¹ published in 1988, reflects this version called ANSI C, a version of the language still used worldwide (Ke88).

C++, an extension of C, was developed by Bjarne Stroustrup in the early 1980s at Bell Laboratories. C++ provides a number of features that “spruce up” the C language, but more importantly, it provides capabilities for *object-oriented programming*. C++ was also standardized by the ANSI and ISO committees.

There is a revolution brewing in the software community. Building software quickly, correctly and economically remains an elusive goal, and this at a time when demands for new and more powerful software are soaring. *Objects* are essentially reusable software *components* that model items in the real world. Software developers are discovering that using a modular, object-oriented design and implementation approach can make software-development groups much more productive than is possible with previous popular programming techniques such as structured programming. Object-oriented programs are often easier to understand, correct and modify.

Many other object-oriented languages have been developed, including Smalltalk, developed at Xerox’s Palo Alto Research Center (PARC). Smalltalk is a pure object-oriented language—literally everything is an object. C++ is a hybrid language—it is possible to program in either a C-like style, an object-oriented style or both.

1.8 History of Java

Perhaps the microprocessor revolution’s most important contribution to date is that it made possible the development of personal computers, which now number in the hundreds of millions worldwide. Personal computers have had a profound impact on people and the way organizations conduct and manage their business.

Many people believe that the next major area in which microprocessors will have a profound impact is in intelligent consumer-electronic devices. Recognizing this, Sun Microsystems funded an internal corporate research project code-named Green in 1991. The project resulted in the development of a C- and C++-based language that its creator, James Gosling, called Oak after an oak tree outside his window at Sun. It was later discovered that there already was a computer language called Oak. When a group of Sun people visited a local coffee place, the name Java was suggested, and it stuck.

The Green project ran into some difficulties. The marketplace for intelligent consumer-electronic devices was not developing as quickly as Sun had anticipated. Worse yet, a major contract for which Sun competed was awarded to another company. So the project was in danger of being canceled. By sheer good fortune, the World Wide Web exploded in popularity in 1993, and Sun people saw the immediate potential of using Java to create Web pages with so-called *dynamic content*. This breathed new life into the project.

Sun formally announced Java at a major conference in May 1995. Ordinarily, an event like this would not have generated much attention. However, Java generated immediate interest in the business community because of the phenomenal interest in the World Wide Web. Java is now used to create Web pages with dynamic and interactive content, to develop large-scale enterprise applications, to enhance the functionality of World Wide

1. Kernighan, B. W., and D. M. Ritchie, *The C Programming Language* (Second Edition), Englewood Cliffs, NJ: Prentice Hall, 1988.

Web servers (the computers that provide the content we see in our Web browsers), to provide applications for consumer devices (such as cell phones, pagers and personal digital assistants) and for many other purposes.

1.9 Java Class Libraries

Java programs consist of pieces called *classes*. Classes consist of pieces called *methods* that perform tasks and return information when they complete their tasks. You can program each piece you may need to form a Java program. However, most Java programmers take advantage of rich collections of existing classes in *Java class libraries*. The class libraries are also known as the *Java APIs (Application Programming Interfaces)*. Thus, there are really two pieces to learning the Java “world.” The first is learning the Java language itself so that you can program your own classes; the second is learning how to use the classes in the extensive Java class libraries. Throughout the book, we discuss many library classes. Class libraries are provided primarily by compiler vendors, but many class libraries are supplied by independent software vendors (ISVs). Also, many class libraries are available from the Internet and World Wide Web as *freeware* or *shareware*. You can download freeware products and use them for free—subject to any restrictions specified by the copyright owner. You also can download shareware products for free, so you can try the software. Shareware products often are free of charge for personal use. However, for shareware products that you use regularly or use for commercial purposes, you are expected to pay a fee designated by the copyright owner.

Many freeware and shareware products are also *open source*. The source code for open-source products is freely available on the Internet, which enables you to learn from the source code, validate that the code serves its stated purpose and even modify the code. Often, open-source products require that you publish any enhancements you make so the open-source community can continue to evolve those products. One example of a popular open-source product is the Linux operating system.



Software Engineering Observation 1.1

Use a building-block approach to creating programs. Avoid reinventing the wheel. Use existing pieces—this is called software reuse and it is central to object-oriented programming.

[*Note:* We will include many of these *Software Engineering Observations* throughout the text to explain concepts that affect and improve the overall architecture and quality of software systems, and particularly, of large software systems. We will also highlight *Good Programming Practices* (practices that can help you write programs that are clearer, more understandable, more maintainable and easier to test and debug), *Common Programming Errors* (problems to watch out for so you do not make these same errors in your programs), *Performance Tips* (techniques that will help you write programs that run faster and use less memory), *Portability Tips* (techniques that will help you write programs that can run, with little or no modifications, on a variety of computers; these tips also include general observations about how Java achieves its high degree of portability), *Testing and Debugging Tips* (techniques that will help you remove bugs from your programs and, more important, techniques that will help you write bug-free programs to begin with) and *Look and Feel Observations* (techniques that will help you design the “look” and “feel” of your graphical user interfaces for appearance and ease of use). Many of these techniques and practices are only guidelines; you will, no doubt, develop your own preferred programming style.]

**Software Engineering Observation 1.2**

When programming in Java, you will typically use the following building blocks: Classes from class libraries, classes and methods you create yourself and classes and methods other people create and make available to you.

The advantage of creating your own classes and methods is that you know exactly how they work and you can examine the Java code. The disadvantage is the time-consuming and complex effort that goes into designing and developing new classes and methods.

**Performance Tip 1.1**

Using Java API classes and methods instead of writing your own versions can improve program performance, because these classes and methods are carefully written to perform efficiently. This technique also improves the prototyping speed of program development (i.e., the time it takes to develop a new program and get its first version running).

**Portability Tip 1.1**

Using classes and methods from the Java API instead of writing your own versions improves program portability, because these classes and methods are included in every Java implementation (assuming the same version number).

**Software Engineering Observation 1.3**

Extensive class libraries of reusable software components are available over the Internet and the Web. Many of these libraries provide source code and are available at no charge.

1.10 Other High-Level Languages

Hundreds of high-level languages have been developed, but only a few have achieved broad acceptance. *Fortran* (FORmula TRANslator) was developed by IBM Corporation between 1954 and 1957 to be used for scientific and engineering applications that require complex mathematical computations. Fortran is still widely used.

COBOL (COmmon Business Oriented Language) was developed in 1959 by a group of computer manufacturers and government and industrial computer users. COBOL is used primarily for commercial applications that require precise and efficient manipulation of large amounts of data. Today, about half of all business software is still programmed in COBOL. Approximately one million people are actively writing COBOL programs.

Pascal was designed at about the same time as C. It was created by Professor Nicklaus Wirth and was intended for academic use. We discuss Pascal further in the next section.

Basic was developed in 1965 at Dartmouth College as a simple language to help novices become comfortable with programming. *Bill Gates* implemented Basic on several early personal computers. Today, *Microsoft*—the company Bill Gates created—is the world's leading software-development organization.

1.11 Structured Programming

During the 1960s, many large software-development efforts encountered severe difficulties. Software schedules were typically late, costs greatly exceeded budgets and the finished products were unreliable. People began to realize that software development was a far more complex activity than they had imagined. Research activity in the 1960s resulted

in the evolution of *structured programming*—a disciplined approach to writing programs that are clearer than unstructured programs, easier to test and debug and easier to modify. Chapters 4 and 5 discuss the principles of structured programming.

One of the more tangible results of this research was the development of the Pascal programming language by Nicklaus Wirth in 1971. Pascal, named after the seventeenth-century mathematician and philosopher Blaise Pascal, was designed for teaching structured programming in academic environments and rapidly became the preferred programming language in most universities. Unfortunately, the language lacks many features needed to make it useful in commercial, industrial and government applications, so it has not been widely accepted in these environments.

The Ada programming language was developed under the sponsorship of the United States Department of Defense (DOD) during the 1970s and early 1980s. Hundreds of separate languages were being used to produce DOD's massive command-and-control software systems. DOD wanted a single language that would fill most of its needs. Pascal was chosen as a base, but the final Ada language is quite different from Pascal. The language was named after Lady Ada Lovelace, daughter of the poet Lord Byron. Lady Lovelace is credited with writing the world's first computer program in the early 1800s (for the Analytical Engine mechanical computing device designed by Charles Babbage). One important capability of Ada is called *multitasking*, which allows programmers to specify that many activities are to occur in parallel. The native capabilities of other widely used high-level languages we have discussed—including C and C++—generally allow the programmer to write programs that perform only one activity at a time. Java, through a technique we will explain called *multithreading*, also enables programmers to write programs with parallel activities. [Note: Most operating systems provide libraries specific to individual platforms (sometimes called *platform-dependent libraries*) that enable high-level languages like C and C++ to specify that many activities are to occur in parallel in a program.]

1.12 The Internet and the World Wide Web

The *Internet* was developed more than three decades ago with funding supplied by the Department of Defense. Originally designed to connect the main computer systems of about a dozen universities and research organizations, the Internet today is accessible by hundreds of millions of computers worldwide.

With the introduction of the *World Wide Web*—which allows computer users to locate and view multimedia-based documents on almost any subject—the Internet has exploded into one of the world's premier communication mechanisms.

The Internet and the World Wide Web will surely be listed among the most important and profound creations of humankind. In the past, most computer applications ran on computers that were not connected to one another. Today's applications can be written to communicate among the world's hundreds of millions of computers. The Internet mixes computing and communications technologies. It makes our work easier. It makes information instantly and conveniently accessible worldwide. It makes it possible for individuals and local small businesses to get worldwide exposure. It is changing the nature of the way business is done. People can search for the best prices on virtually any product or service. Special-interest communities can stay in touch with one another. Researchers can be made instantly aware of the latest breakthroughs worldwide.

Java How to Program: Fourth Edition presents programming techniques that allow Java applications to use the Internet and World Wide Web to interact with other applications. These capabilities, and the capabilities discussed in our companion book *Advanced Java 2 Platform How to Program*, allow Java programmers to develop the kind of enterprise-level distributed applications that are used in industry today. Java applications can be written to execute on any computer platform, yielding major savings in systems development time and cost for corporations. If you have been hearing a great deal about the Internet and World Wide Web lately, and if you are interested in developing applications to run over the Internet and the Web, learning Java may be the key to challenging and rewarding career opportunities for you.

1.13 Basics of a Typical Java Environment

Java systems generally consist of several parts: An environment, the language, the Java Applications Programming Interface (API) and various class libraries. The following discussion explains a typical Java program development environment, as shown in Fig. 1.1.

Java programs normally go through five phases to be executed (Fig. 1.1). These are: *edit*, *compile*, *load*, *verify* and *execute*. We discuss these concepts in the context of the *Java 2 Software Development Kit (J2SDK)* that is included on the CD that accompanies this book. *Carefully follow the installation instructions for the J2SDK provided on the CD to ensure that you set up your computer properly to compile and execute Java programs.* [Note: If you are not using UNIX/Linux, Windows 95/98/ME or Windows NT/2000, refer to the manuals for your system's Java environment or ask your instructor how to accomplish these tasks in your environment (which will probably be similar to the environment in Fig. 1.1).]

Phase 1 consists of editing a file. This is accomplished with an *editor program* (normally known as an *editor*). The programmer types a Java program, using the editor, and makes corrections, if necessary. When the programmer specifies that the file in the editor should be saved, the program is stored on a secondary storage device, such as a disk. Java program file names end with the **.java** extension. Two editors widely used on UNIX/Linux systems are **vi** and **emacs**. On Windows 95/98/ME and Windows NT/2000, simple edit programs like the DOS Edit command and the Windows Notepad will suffice. Java integrated development environments (IDEs), such as Forté for Java Community Edition, NetBeans, Borland's JBuilder, Symantec's Visual Cafe and IBM's VisualAge have built-in editors that are integrated into the programming environment. We assume the reader knows how to edit a file.

[Note that Forté for Java Community Edition is written in Java and is free for non-commercial use. It is included on the CD accompanying this book. Sun updates this software approximately twice a year. Newer versions can be downloaded from

www.sun.com/forte/ffj

Forté for Java Community Edition executes on most major platforms. This book is written for any generic Java 2 development environment. It is not dependent on Forté for Java Community Edition. Our example programs should operate properly with most Java integrated development environments.]

In Phase 2 (discussed again in Chapters 2 and 3), the programmer gives the command **javac** to *compile* the program. The Java compiler translates the Java program into *byte-*

codes—the language understood by the Java interpreter. To compile a program called **Welcome.java**, type

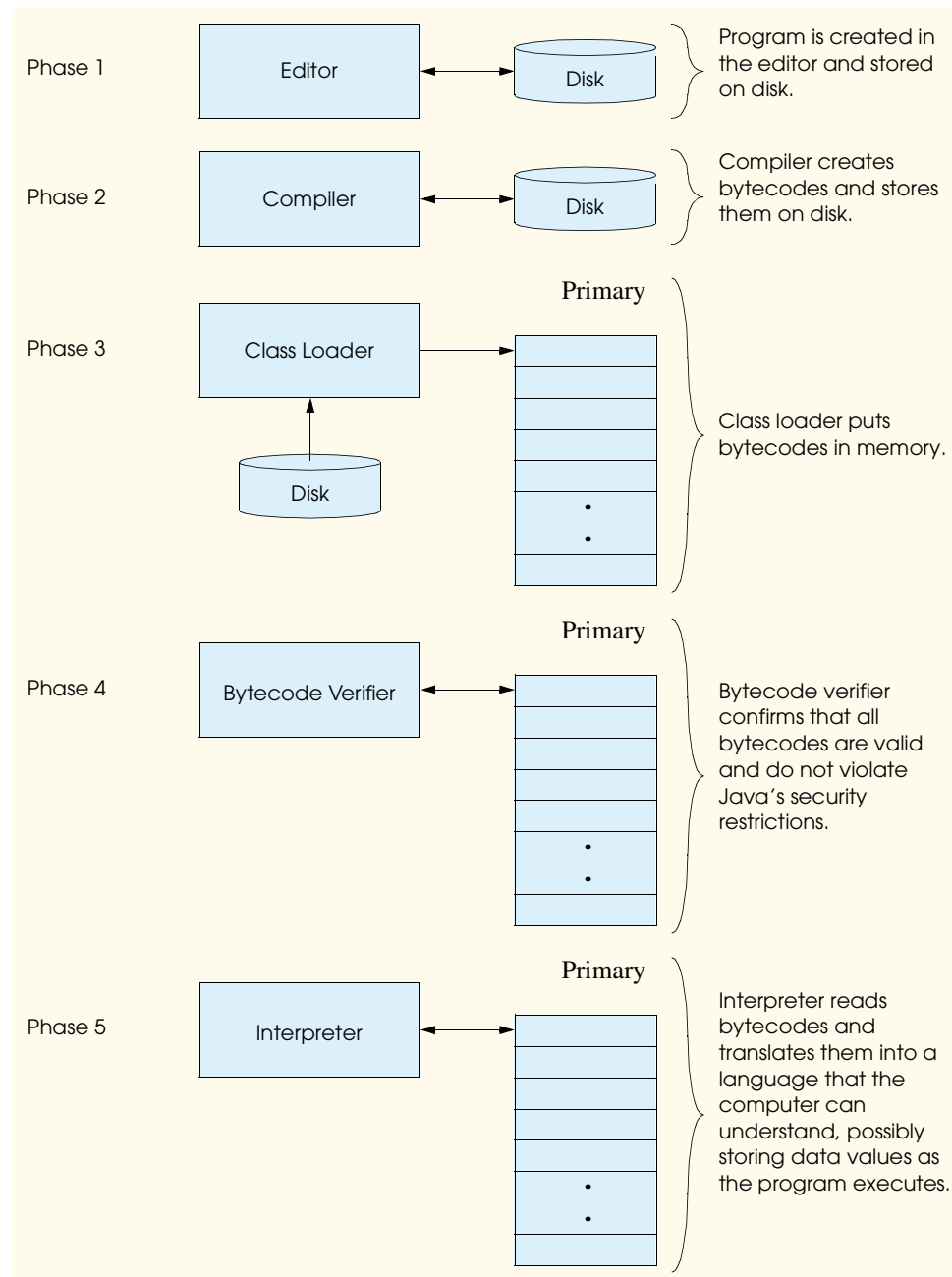


Fig. 1.1 Typical Java environment.

```
javac Welcome.java
```

at the command window of your system (i.e., the *MS-DOS prompt* in Windows, the *Command Prompt* in Windows NT/2000 or the *shell prompt* in UNIX/Linux). If the program compiles correctly, the compiler produces a file called **Welcome.class**. This is the file containing the bytecodes that will be interpreted during the execution phase.

Phase 3 is called *loading*. The program must first be placed in memory before it can be executed. This is done by the *class loader*, which takes the **.class** file (or files) containing the bytecodes and transfers it to memory. The **.class** file can be loaded from a disk on your system or over a network (such as your local university or company network or even the Internet). There are two types of programs for which the class loader loads **.class** files—*applications* and *applets*. An application is a program (such as a word-processor program, a spreadsheet program, a drawing program or an e-mail program) that normally is stored and executed from the user's local computer. An applet is a small program that normally is stored on a remote computer that users connect to via a World Wide Web browser. Applets are loaded from a remote computer into the browser, executed in the browser and discarded when execution completes. To execute an applet again, the user must point a browser at the appropriate location on the World Wide Web and reload the program into the browser.

Applications are loaded into memory and executed by using the *Java interpreter* via the command **java**. When executing a Java application called **Welcome**, the command

```
java Welcome
```

invokes the interpreter for the **Welcome** application and causes the class loader to load information used in the **Welcome** program. [Note: Many Java programmers refer to the interpreter as the *Java Virtual Machine* or the *JVM*.]

The class loader also executes when a World Wide Web browser such as *Netscape Navigator* or *Microsoft Internet Explorer* loads a Java applet. Browsers are used to view documents on the World Wide Web called *Hypertext Markup Language (HTML)* documents. HTML describes the format of a document in a manner that is understood by the browser application (we introduce HTML in Section 3.4; for a detailed treatment of HTML and other Internet programming technologies, please see our text *Internet and World Wide Web How to Program, Second Edition*). An HTML document may refer to a Java applet. When the browser sees an applet referenced in an HTML document, the browser launches the Java class loader to load the applet (normally from the location where the HTML document is stored). Each browser that supports Java has a built-in Java interpreter. After the applet loads, the browser's Java interpreter executes the applet. Applets can also execute from the command line, using the **appletviewer** command provided with the J2SDK—the set of tools including the compiler (**javac**), interpreter (**java**), **appletviewer** and other tools used by Java programmers. Like Netscape Navigator and Microsoft Internet Explorer, the **appletviewer** requires an HTML document to invoke an applet. For example, if the **Welcome.html** file refers to the **Welcome** applet, the **appletviewer** command is used as follows:

```
appletviewer Welcome.html
```

This causes the class loader to load the information used in the **Welcome** applet. The **appletviewer** is a minimal browser—it knows only how to interpret references to applets and ignores all other HTML in a document.

Before the Java interpreter built into a browser or the **appletviewer** executes the bytecodes in an applet, the bytecodes are verified by the *bytecode verifier* in Phase 4. This ensures that the bytecodes for classes that are loaded from the Internet (referred to as *downloaded classes*) are valid and that they do not violate Java's security restrictions. Java enforces strong security, because Java programs arriving over the network should not be able to cause damage to your files and your system (as computer viruses might). Note that bytecode verification also occurs in applications that download classes from a network.

Finally, in Phase 5, the computer, under the control of its CPU, interprets the program one bytecode at a time, thus performing the actions specified by the program.

Programs might not work on the first try. Each of the preceding phases can fail because of various errors that we will discuss in this text. For example, an executing program might attempt to divide by zero (an illegal operation in Java just as it is in arithmetic). This would cause the Java program to print an error message. The programmer would return to the edit phase, make the necessary corrections and proceed through the remaining phases again to determine that the corrections work properly.



Common Programming Error 1.1

Errors like division-by-zero errors occur as a program runs, so these errors are called runtime errors or execution-time errors. Fatal runtime errors cause programs to terminate immediately without having successfully performed their jobs. Nonfatal runtime errors allow programs to run to completion, often producing incorrect results.

Most programs in Java input or output data. When we say that a program prints a result, we normally mean that the program displays results on the computer screen. Data may be output to other devices, such as disks and hardcopy printers.

1.14 General Notes about Java and This Book

Java is a powerful language. Experienced programmers sometimes take pride in being able to create some weird, contorted, convoluted usage of a language. This is a poor programming practice. It makes programs more difficult to read, more likely to behave strangely, more difficult to test and debug and more difficult to adapt to changing requirements. This book is also geared for novice programmers, so we stress *clarity*. The following is our first "good programming practice."



Good Programming Practice 1.1

Write your Java programs in a simple and straightforward manner. This is sometimes referred to as KIS ("keep it simple"). Do not "stretch" the language by trying bizarre usages.

You have heard that Java is a portable language and that programs written in Java can run on many different computers. For programming in general, *portability is an elusive goal*. For example, the ANSI C standard document² contains a lengthy list of portability issues, and complete books have been written that discuss portability.^{3,4}

2. ANSI, *American National Standard for Information Systems—Programming Language C (ANSI Document ANSI/ISO 9899: 1990)*, New York, NY: American National Standards Institute, 1990.

3. Jaeschke, R., *Portability and the C Language*, Indianapolis, IN: Hayden Books, 1989.

4. Rabinowitz, H., and C. Schaap, *Portable C*, Englewood Cliffs, NJ: Prentice Hall, 1990.



Portability Tip 1.2

Although it is easier to write portable programs in Java than in other programming languages, there are differences among compilers, interpreters and computers that can make portability difficult to achieve. Simply writing programs in Java does not guarantee portability. The programmer will occasionally need to deal with compiler and computer variations.



Testing and Debugging Tip 1.1

Always test your Java programs on all systems on which you intend to run those programs, to ensure that your Java programs will work correctly for their intended audience.

We have done a careful walkthrough of Sun's Java documentation and audited our presentation against it for completeness and accuracy. However, Java is a rich language, and there are some subtleties in the language and some topics we have not covered. If you need additional technical details on Java, we suggest that you read the most current Java documentation available over the Internet at java.sun.com. Our book contains an extensive bibliography of books and papers on the Java language in particular and on object-oriented programming in general. A Web-based version of the Java API documentation can be found at java.sun.com/j2se/1.3/docs/api/index.html. Also, you can download this documentation to your own computer from java.sun.com/j2se/1.3/docs.html.



Good Programming Practice 1.2

Read the documentation for the version of Java you are using. Refer to this documentation frequently to be sure you are aware of the rich collection of Java features and that you are using these features correctly.



Good Programming Practice 1.3

Your computer and compiler are good teachers. If, after carefully reading your Java documentation manual, you are not sure how a feature of Java works, experiment and see what happens. Study each error or warning message you get when you compile your programs, and correct the programs to eliminate these messages.



Good Programming Practice 1.4

The Java 2 Software Development Kit comes with the Java source code. Many programmers read the actual source code of the Java API classes to determine how those classes work and to learn additional programming techniques. If the Java API documentation is not clear on a particular topic, try studying the source code of the class.

In this book, we explain how Java works in its current implementations. Perhaps the most striking problem with the early versions of Java is that Java programs execute interpretively on the client's machine. Interpreters execute slowly compared to fully compiled machine code.



Performance Tip 1.2

Interpreters have an advantage over compilers for the Java world, namely that an interpreted program can begin execution immediately as soon as it is downloaded to the client's machine, whereas a source program to be compiled must first suffer a potentially long delay as the program is compiled before it can be executed.



Portability Tip 1.3

Although only Java interpreters were available to execute bytecodes at the client's site on early Java systems, compilers that translate Java bytecodes (or in some cases the Java source code) into the native machine code of the client's machine have been written for most popular platforms. These compiled programs perform comparably to compiled C or C++ code. However, there are not bytecode compilers for every Java platform, so Java programs will not perform at the same level on all platforms.

Applets present some more interesting issues. Remember, an applet could be coming from virtually any Web server in the world. So the applet will have to be able to run on any possible Java platform.



Portability Tip 1.4

Short, fast-executing Java applets can certainly still be interpreted. But what about more substantial, compute-intensive applets? Here, the user might be willing to suffer the compilation delay to get better execution performance. For some especially performance-intensive applets, the user might have no choice; interpreted code would run too slowly for the applet to perform properly, so the applet would have to be compiled.



Portability Tip 1.5

An intermediate step between interpreters and compilers is a just-in-time (JIT) compiler that, as the interpreter runs, produces compiled code for the programs and executes the programs in machine language rather than reinterpreting them. JIT compilers do not produce machine language that is as efficient as that from a full compiler.



Portability Tip 1.6

For the latest information on high-speed Java program translation, you might want to read about Sun's HotSpot™ compiler, so visit java.sun.com/products/hotspot. The HotSpot compiler is a standard component of the Java 2 Software Development Kit.

The Java compiler, **javac**, is not a traditional compiler in that it does not convert a Java program from source code into native machine code for a particular computer platform. Instead, the Java compiler translates source code into bytecodes. Bytecodes are the language of the Java Virtual Machine—a program that simulates the operation of a computer and executes its own machine language (i.e., Java bytecodes). The Java Virtual Machine is implemented in the J2SDK as the **java** interpreter, which translates the bytecodes into native machine language for the local computer platform.



Software Engineering Observation 1.4

For organizations wanting to do heavy-duty information systems development, Integrated Development Environments (IDEs) are available from many major software suppliers, including Sun Microsystems. The IDEs provide many tools for supporting the software-development process, such as editors for writing and editing programs, debuggers for locating logic errors in programs and many other features.



Software Engineering Observation 1.5

Sun Microsystems, Inc.'s powerful Java IDE—Forté for Java, Community Edition—is available on the CD that accompanies this book and can be downloaded from www.sun.com/forte/ffj.

1.15 Thinking About Objects: Introduction to Object Technology and the Unified Modeling Language

Now we begin our early introduction to object orientation. We will see that object orientation is a natural way of thinking about the world and a natural way of writing computer programs.

In the bodies of each of the first seven chapters, we concentrate on the “conventional” methodology of structured programming, because the objects we will build will be composed in part of structured-program pieces. However, we end each chapter with a “Thinking About Objects” section in which we present a carefully paced introduction to object orientation. Our goal in these “Thinking About Objects” sections is to help you develop an object-oriented way of thinking, so that you immediately can use the object-oriented programming techniques that we present starting in Chapter 8. The “Thinking About Objects” sections also introduce you to the *Unified Modeling Language (UML)*. The UML is a graphical language that allows people who build systems (e.g., software architects, systems engineers, programmers and so on) to represent their object-oriented designs, using a common notation.

In this section, we introduce basic concepts (i.e., “object think”) and terminology (i.e., “object speak”). Chapters 2–13, 15 and 22 and Appendices G–I include optional “Thinking About Objects” sections that present a substantial case study that applies the techniques of *object-oriented design (OOD)*. The optional sections at the ends of Chapters 2 through 7 analyze a typical problem statement that requires a system to be built, determine the objects required to implement that system, determine the attributes the objects will have, determine the behaviors these objects will exhibit and specify how the objects will interact with one another to meet the system requirements. All this occurs *before* you learn to write object-oriented Java programs! The optional sections at the ends of Chapters 8–13 and 15 modify and enhance the design presented in Chapters 2–7. Chapter 22 presents how to display our multimedia-rich design on the screen. The optional “Thinking About Objects” sections in each chapter apply the concepts discussed in that chapter to the case study. In Appendices G, H and I, we present a complete Java implementation of the object-oriented system we design in the earlier chapters.

This case study will help prepare you for the kinds of substantial projects you are likely to encounter in industry. If you are a student and your instructor does not plan to include this case study in your course, you may want to cover it on your own time. We believe it will be well worth your time to walk through this large and challenging project, because the material presented in the case-study sections reinforces the material covered in the corresponding chapters. You will experience a solid introduction to object-oriented design with the UML. Also, you will sharpen your code-reading skills by touring a carefully written and well-documented 3,594-line Java program that completely solves the problem presented in the case study.

We begin our introduction to object orientation with some key terminology. Everywhere you look in the real world you see them—*objects*: People, animals, plants, cars, planes, buildings, computers and so on. Humans think in terms of objects. We possess the marvelous ability of *abstraction*, which enables us to view screen images such as people, planes, trees and mountains as objects, rather than as individual dots of color (called *pixels*—for “picture elements”). We can, if we wish, think in terms of beaches rather than grains of sand, forests rather than trees and houses rather than bricks.

We might be inclined to divide objects into two categories—animate objects and inanimate objects. Animate objects are “alive” in some sense; they move around and do things. Inanimate objects, on the other hand, seem not to do much at all. They do not move on their own. All these objects, however, do have some things in common. They all have *attributes* like size, shape, color, and weight, and they all exhibit *behaviors* (e.g., a ball rolls, bounces, inflates and deflates; a baby cries, sleeps, crawls, walks and blinks; a car accelerates, brakes and turns; a towel absorbs water).

Humans learn about objects by studying their attributes and observing their behaviors. Different objects can have similar attributes and can exhibit similar behaviors. Comparisons can be made, for example, between babies and adults and between humans and chimpanzees. Cars, trucks, little red wagons and roller skates have much in common.

Object-oriented design models real-world objects. It takes advantage of *class* relationships, where objects of a certain class—such as a class of vehicles—have the same characteristics. It takes advantage of *inheritance* relationships, and even *multiple-inheritance*⁵ relationships, where newly created classes of objects are derived by absorbing characteristics of existing classes and adding unique characteristics of their own. An object of class “convertible” certainly has the characteristics of the more general class “automobile,” plus a convertible’s roof goes up and down.

Object-oriented design provides a more natural and intuitive way to view the design process—namely, by *modeling* real-world objects, their attributes, their behavior. OOD also models communication between objects. Just as people send *messages* to one another (e.g., a sergeant commanding a soldier to stand at attention), objects also communicate via messages.

OOD *encapsulates* data (attributes) and functions (behavior) into *objects*; the data and functions of an object are intimately tied together. Objects have the property of *information hiding*. This means that, although objects may know how to communicate with one another across well-defined *interfaces*, objects normally are not allowed to know how other objects are implemented—implementation details are hidden within the objects themselves. Surely, it is possible to drive a car effectively without knowing the details of how engines, transmissions and exhaust systems work internally. We will see why information hiding is so crucial to good software engineering.

Languages such as Java are *object-oriented*—programming in such a language is called *object-oriented programming (OOP)* and allows designers to implement the object-oriented design as a working system. Languages such as C, on the other hand, are *procedural programming languages*, so programming tends to be *action-oriented*. In C, the unit of programming is the *function*. In Java, the unit of programming is the *class* from which objects are eventually *instantiated* (a fancy term for “created”). Java classes contain *methods* (that implement class behaviors) and attributes (that implement class data).

C programmers concentrate on writing functions. Groups of actions that perform some common task are formed into functions, and functions are grouped to form programs. Data are certainly important in C, but the view is that data exist primarily in support of the actions that functions perform. The *verbs* in a system specification help the C programmer determine the set of functions needed to implement that system.

Java programmers concentrate on creating their own *user-defined types* called *classes* and *components*. Each class contains data and the set of functions that manipulate that data.

5. We will learn later that although Java—unlike C++—does not support multiple inheritance, it does offer most of the key benefits of this technology by supporting multiple interfaces per class.

The data components of a Java class are called *attributes*. The function components of a Java class are called *methods*. Just as an instance of a built-in type such as **int** is called a *variable*, an instance of a user-defined type (i.e., a class) is called an *object*. The programmer uses built-in types as the “building blocks” for constructing user-defined types. The focus in Java is on classes (out of which we make objects) rather than on functions. The *nouns* in a system specification help the Java programmer determine the set of classes from which objects will be created that will work together to implement the system.

Classes are to objects as blueprints are to houses. We can build many houses from one blueprint, and we can instantiate many objects from one class. Classes can also have relationships with other classes. For example, in an object-oriented design of a bank, the “bank-teller” class needs to relate to the “customer” class. These relationships are called *associations*.

We will see that, when software is packaged as classes, these classes can be *reused* in future software systems. Groups of related classes are often packaged as reusable *components*. Just as real-estate brokers tell their clients that the three most important factors affecting the price of real estate are “location, location and location,” many people in the software community believe that the three most important factors affecting the future of software development are “reuse, reuse and reuse.”

Indeed, with object technology, we can build much of the software we will need by combining “standardized, interchangeable parts” called classes. This book teaches you how to “craft valuable classes” for reuse. Each new class you create will have the potential to become a valuable software asset that you and other programmers can use to speed and enhance the quality of future software-development efforts—an exciting possibility.

Introduction to Object-Oriented Analysis and Design (OOAD)

You soon will be writing programs in Java. How will you create the code for your programs? If you are like many beginning programmers, you will simply turn on your computer and start typing. This approach may work for small projects, but what would you do if you were asked to create a software system to control the automated teller machines for a major bank? Such a project is too large and complex for you to sit down and simply start typing.

To create the best solutions, you should follow a detailed process for obtaining an *analysis* of your project’s *requirements* and developing a *design* for satisfying those requirements. Ideally, you would go through this process and have its results reviewed and approved by your superiors before writing any code for your project. If this process involves analyzing and designing your system from an object-oriented point of view, we call it an *object-oriented analysis and design (OOAD) process*. Experienced programmers know that, no matter how simple a problem appears, time spent on analysis and design can save innumerable hours that might be lost from abandoning an ill-planned system-development approach part of the way through its implementation.

OOAD is the generic term for the ideas behind the process we employ to analyze a problem and develop an approach for solving it. Small problems like the ones discussed in these first few chapters do not require an exhaustive process. It may be sufficient to write *pseudocode* before we *begin* writing code. (Pseudocode is an informal means of expressing program code. It is not actually a programming language, but we can use it as a kind of “outline” to guide us as we write our code. We introduce pseudocode in Chapter 4.)

Pseudocode can suffice for small problems, but as problems and the groups of people solving these problems increase in size, the methods of OOAD become more involved. Ide-

ally, a group should agree on a strictly defined process for solving the problem and on a uniform way of communicating the results of that process to one another. Although many different OOAD processes exist, a single graphical language for communicating the results of any OOAD process has become widely used. This language is known as the *Unified Modeling Language (UML)*. The UML was developed in the mid-1990s under the initial direction of three software methodologists: Grady Booch, James Rumbaugh and Ivar Jacobson.

History of the UML

In the 1980s, increasing numbers of organizations began using OOP to program their applications, and a need developed for an established process with which to approach OOAD. Many methodologists—including Booch, Rumbaugh and Jacobson—individually produced and promoted separate processes to satisfy this need. Each of these processes had its own notation, or “language” (in the form of graphical diagrams), to convey the results of analysis and design.

By the early 1990s, different companies, and even different divisions within the same company, were using different processes and notations. Additionally, these companies wanted to use software tools that would support their particular processes. With so many processes, software vendors found it difficult to provide such tools. Clearly, a standard notation and standard processes were needed.

In 1994, James Rumbaugh joined Grady Booch at Rational Software Corporation, and the two began working to unify their popular processes. They were soon joined by Ivar Jacobson. In 1996, the group released early versions of the UML to the software engineering community and requested feedback. Around the same time, an organization known as the *Object Management Group™ (OMG™)* invited submissions for a common modeling language. The OMG is a not-for-profit organization that promotes the use of object-oriented technology by issuing guidelines and specifications for object-oriented technologies. Several corporations—among them HP, IBM, Microsoft, Oracle and Rational Software—had already recognized the need for a common modeling language. These companies formed the *UML Partners* in response to the OMG’s request for proposals. This consortium developed the UML version 1.1 and submitted it to the OMG. The OMG accepted the proposal and, in 1997, assumed responsibility for the continuing maintenance and revision of the UML. In 2001, the OMG released the UML version 1.4 (the current version at the time this book was published) and is working on version 2.0 (scheduled tentatively for release in 2002).

What is the UML?

The Unified Modeling Language is now the most widely used graphical representation scheme for modeling object-oriented systems. It has indeed unified the various popular notational schemes. Those who design systems use the language (in the form of graphical diagrams) to model their systems.

An attractive feature of the UML is its flexibility. The UML is extendable and is independent of the many OOAD processes. UML modelers are free to develop systems by using various processes, but all developers can now express those systems with one standard set of notations.

The UML is a complex, feature-rich graphical language. In our “Thinking About Objects” sections, we present a concise, simplified subset of these features. We then use this subset to guide the reader through a first design experience with the UML intended for the

novice object-oriented designer/programmer. For a more complete discussion of the UML, refer to the Object Management Group's Web site (www.omg.org) and to the official UML 1.4 specifications document (www.omg.org/uml). In addition, many UML books have been published: *UML Distilled: Second Edition*, by Martin Fowler (with Kendall Scott) (ISBN #020165783X) provides a detailed introduction to the UML, with many examples. *The Unified Modeling Language User Guide* (ISBN #0201571684), written by Booch, Rumbaugh and Jacobson, is the definitive tutorial to the UML. The reader looking for an interactive learning product might consider Grady Booch's *The Complete UML Training Course* (ISBN #0130870145).

Object-oriented technology is ubiquitous in the software industry, and the UML is rapidly becoming so. Our goal in these "Thinking About Objects" sections is to encourage you to think in an object-oriented manner as early, and as often, as possible. In the "Thinking About Objects" section at the end of Chapter 2, you will begin to apply object technology to implement a solution to a substantial problem. We hope that you will find this optional project to be an enjoyable and challenging introduction to object-oriented design with the UML and to object-oriented programming.

1.16 Discovering Design Patterns: Introduction

This section begins our treatment of design patterns, entitled "Discovering Design Patterns." Most of the examples provided in this book contain fewer than 150 lines of code. These examples do not require an extensive design process, because they use only a few classes and illustrate introductory programming concepts. However, some programs, such as our optional elevator-simulation case study, are more complex—they can require thousands of lines of code or even more, contain many interactions among objects and involve many user interactions. Larger systems, such as automated teller machines or air-traffic control systems, could contain millions of lines of code. Effective design is crucial to the proper construction of such complex systems.

Over the past decade, the software engineering industry has made significant progress in the field of *design patterns*—proven architectures for constructing flexible and maintainable object-oriented software.⁶ Using design patterns can substantially reduce the complexity of the design process. Designing an ATM system will be a somewhat less formidable task if developers use design patterns. In addition, well-designed object-oriented software allows designers to reuse and integrate preexisting components in future systems. Design patterns benefit system developers by

- helping to construct reliable software using proven architectures and accumulated industry expertise
- promoting design reuse in future systems
- helping to identify common mistakes and pitfalls that occur when building systems
- helping to design systems independently of the language in which they will ultimately be implemented

6. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns; Elements of Reusable Object-Oriented Software*. (Massachusetts: Addison-Wesley, 1995).

- establishing a common design vocabulary among developers
- shortening the design phase in a software-development process

The notion of using design patterns to construct software systems originated in the field of architecture. Architects use a set of established architectural design elements, such as arches and columns, when designing buildings. Designing with arches and columns is a proven strategy for constructing sound buildings—these elements may be viewed as architectural design patterns.

In software, design patterns are neither classes nor objects. Rather, designers use design patterns to construct sets of classes and objects. To use design patterns effectively, designers must familiarize themselves with the most popular and effective patterns used in the software-engineering industry. In this chapter, we discuss fundamental object-oriented design patterns and architectures, as well as their importance in constructing well-engineered software.

We present several design patterns in Java, but these design patterns can be implemented in any object-oriented language, such as C++ or Visual Basic. We describe several design patterns used by Sun Microsystems in the Java API. We use design patterns in many programs in this book, which we will identify throughout our discussion. These programs provide examples of using design patterns to construct reliable, robust object-oriented software.

History of Object-Oriented Design Patterns

During 1991–1994, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides—collectively known as the “gang of four”—used their combined expertise to write the book *Design Patterns, Elements of Reusable Object-Oriented Software* (Addison-Wesley: 1995). This book described 23 design patterns, each providing a solution to a common software design problem in industry. The book groups design patterns into three categories—*creational design patterns*, *structural design patterns* and *behavioral design patterns*. Creational design patterns describe techniques to instantiate objects (or groups of objects). Structural design patterns allow designers to organize classes and objects into larger structures. Behavioral design patterns assign responsibilities to objects.

The gang-of-four book showed that design patterns evolved naturally through years of industry experience. In his article *Seven Habits of Successful Pattern Writers*,⁷ John Vlissides states that “the single most important activity in pattern writing is reflection.” This statement implies that, to create patterns, developers must reflect on, and document, their successes (and mistakes). Developers use design patterns to capture and employ this collective industry experience, which ultimately helps them avoid making the same mistakes twice.

New design patterns are being created all the time and being introduced rapidly to designers worldwide via the Internet. The topic of design patterns has generally been viewed as advanced, but authors such as ourselves are working this material into introductory and intermediate-level textbooks to help make this important knowledge available to a much wider audience.

Our treatment of design patterns begins with this required section in Chapter 1 and continues with five optional “Discovering Design Patterns” sections at the ends of Chapters 9, 13, 15, 17 and 21. Each of these sections is placed at the end of the chapter that introduces

7. Vlissides, John. *Pattern Hatching; Design Patterns Applied*. (Massachusetts: Addison-Wesley, 1998).

the necessary Java technologies. If you are a student and your instructor does not plan to include this material in your course, we encourage you to read this material on your own.

1.17 Tour of the Book

You are about to study one of today's most exciting and rapidly developing computer programming languages. Mastering Java will help you develop powerful business and personal computer-applications software. In this section, we take a tour of the many capabilities of Java you will study in *Java How to Program: Fourth Edition*.

Chapter 1—Introduction to Computers, the Internet and the Web—discusses what computers are, how they work and how they are programmed. The chapter gives a brief history of the development of programming languages from machine languages, to assembly languages, to high-level languages. The origin of the Java programming language is discussed. The chapter includes an introduction to a typical Java programming environment. The chapter also introduces object technology, the Unified Modeling Language and design patterns.

Chapter 2—Introduction to Java Applications—provides a lightweight introduction to programming *applications* in the Java programming language. The chapter introduces nonprogrammers to basic programming concepts and constructs. The programs in this chapter illustrate how to display (also called *outputting*) data on the screen to the user and how to obtain (also called *inputting*) data from the user at the keyboard. Some of the input and output is by performed using a *graphical user interface (GUI)* component called **JOptionPane** that provides predefined windows (called dialog boxes) for input and output. This allows a nonprogrammer to concentrate on fundamental programming concepts and constructs rather than on the more complex GUI event handling. Using **JOptionPane** here enables us to delay our introduction of GUI event handling to Chapter 6, "Methods." Chapter 2 also provides detailed treatments of *decision making* and *arithmetic operations*. After studying this chapter, the student will understand how to write simple, but complete, Java applications.

Chapter 3—Introduction to Java Applets—introduces another type of Java program, called an *applet*. Applets are Java programs designed to be transported over the Internet and executed in World Wide Web browsers (like Netscape Navigator and Microsoft Internet Explorer). The chapter introduces applets, using several of the demonstration applets supplied with the Java 2 Software Development Kit (J2SDK). We use **appletviewer** (a utility supplied with the J2SDK) or a Web browser to execute several sample applets. We then write Java applets that perform tasks similar to the programs of Chapter 2, and we explain the similarities and differences between applets and applications. After studying this chapter, the student will understand how to write simple, but complete, Java applets. The next several chapters use both applets and applications to demonstrate additional key programming concepts.

Chapter 4—Control Structures: Part 1—focuses on the program-development process. The chapter discusses how to take a *problem statement* (i.e., a *requirements document*) and from it develop a working Java program, including performing intermediate steps in pseudocode. The chapter introduces some fundamental data types and simple control structures used for decision making (**if** and **if/else**) and repetition (**while**). We examine counter-controlled repetition and sentinel-controlled repetition, and introduce Java's increment, decrement and assignment operators. The chapter uses simple flowcharts

to show the flow of control through each of the control structures. The techniques discussed in Chapters 2 through 7 constitute a large part of what has been traditionally taught in the universities under the topic of structured programming. With Java, we do object-oriented programming. In doing so, we discover that the insides of the objects we build make abundant use of control structures. We have had a particularly positive experience assigning problems 4.11 through 4.14 in our introductory courses. Since these four problems have similar structure, doing all four is a nice way for students to “get the hang of” the program-development process. This chapter helps the student develop good programming habits in preparation for dealing with the more substantial programming tasks in the remainder of the text.

Chapter 5—Control Structures: Part 2—continues the discussions of Java control structures (**for**, the **switch** selection structure and the **do/while** repetition structure). The chapter explains the labeled **break** and **continue** statements with live-code examples. The chapter also contains a discussion of logical operators—**&&** (logical AND), **&** (boolean logical AND), **||** (logical OR), **|** (boolean logical inclusive OR), **^** (boolean logical exclusive OR) and **!** (NOT). There is a substantial exercise set including mathematical, graphical and business applications. Students will enjoy Exercise 5.25, which asks them to write a program with repetition and decision structures that prints the iterative song, “The Twelve Days of Christmas.” The more mathematically inclined students will enjoy problems on binary, octal, decimal and hexadecimal number systems, calculating the mathematical constant π with an infinite series, Pythagorean triples and De Morgan’s Laws. Our students particularly enjoy the challenges of triangle-printing and diamond-printing in Exercises 5.10, 5.18 and 5.20; these problems help students learn to deal with nested repetition structures—a complex topic to master in introductory courses.

Chapter 6—Methods—takes a deeper look inside objects. Objects contain data called instance variables and executable units called methods (these are often called *functions* in non-object-oriented procedural programming languages like C and *member functions* in C++). We explore methods in depth and include a discussion of methods that “call themselves,” so-called recursive methods. We discuss class-library methods, programmer-defined methods and recursion. The techniques presented in Chapter 6 are essential to the production of properly structured programs, especially the kinds of larger programs and software that system programmers and application programmers are likely to develop in real-world applications. The “divide and conquer” strategy is presented as an effective means for solving complex problems by dividing them into simpler interacting components. Students enjoy the treatment of random numbers and simulation, and they appreciate the discussion of the dice game of craps that makes elegant use of control structures (this is one of our most successful lectures in our introductory courses). The chapter offers a solid introduction to recursion and includes a table summarizing the dozens of recursion examples and exercises distributed throughout the remainder of the book. Some texts leave recursion for a chapter late in the book; we feel this topic is best covered gradually throughout the text. The topic of method overloading (i.e., allowing multiple methods to have the same name as long as they have different “signatures”) is motivated and explained clearly. We introduce *events* and *event handling*—elements required for programming graphical user interfaces. Events are notifications of state change such as button clicks, mouse clicks and pressing a keyboard key. Java allows programmers to specify the responses to events by coding methods called event handlers.

The extensive collection of exercises at the end of the chapter includes several classical recursion problems such as the Towers of Hanoi; we revisit this problem later in the text where we employ graphics, animation and sound to make the problem “come alive.” There are many mathematical and graphical examples. Our students particularly enjoy the development of a “Computer-Assisted Instruction” system in Exercises 6.31 through 6.33; we ask students to develop a multimedia version of this system later in the book. Students will enjoy the challenges of the “mystery programs.” The more mathematically inclined students will enjoy problems on perfect numbers, greatest common divisors, prime numbers and factorials.

Chapter 7—Arrays—explores the processing of data in lists and tables of values. Arrays in Java are processed as objects, further evidence of Java’s commitment to almost 100% object orientation. We discuss the structuring of data into arrays, or groups, of related data items of the same type. The chapter presents numerous examples of both single-subscripted arrays and double-subscripted arrays. It is widely recognized that structuring data properly is just as important as using control structures effectively in the development of properly structured programs. Examples in the chapter investigate various common array manipulations, printing histograms, sorting data, passing arrays to methods and an introduction to the field of survey data analysis (with simple statistics). A feature of this chapter is the discussion of elementary sorting and searching techniques and the presentation of binary searching as a dramatic improvement over linear searching. The end-of-chapter exercises include a variety of interesting and challenging problems, such as improved sorting techniques, the design of an airline reservations system, an introduction to the concept of turtle graphics (made famous in the LOGO programming language) and the Knight’s Tour and Eight Queens problems that introduce the notions of heuristic programming so widely employed in the field of artificial intelligence. The exercises conclude with a series of recursion problems including the selection sort, palindromes, linear search, binary search, the eight queens, printing an array, printing a string backwards and finding the minimum value in an array. The chapter exercises include a delightful simulation of the classic race between the tortoise and the hare, card shuffling and dealing algorithms, recursive quicksort and recursive maze traversals. A special section entitled “Building Your Own Computer” explains machine-language programming and proceeds with the design and implementation of a computer simulator that allows the reader to write and run machine language programs. This unique feature of the text will be especially useful to the reader who wants to understand how computers really work. Our students enjoy this project and often implement substantial enhancements; many enhancements are suggested in the exercises. In Chapter 19, another special section guides the reader through building a compiler; the machine language produced by the compiler is then executed on the machine language simulator produced in Chapter 7. Information is communicated from the compiler to the simulator in sequential files (presented in Chapter 16).

Chapter 8—Object-Based Programming—begins our deeper discussion of classes. The chapter represents a wonderful opportunity for teaching data abstraction the “right way”—through a language (Java) expressly devoted to implementing abstract data types (ADTs). The chapter focuses on the essence and terminology of classes and objects. What is an object? What is a class of objects? What does the inside of an object look like? How are objects created? How are they destroyed? How do objects communicate with one another? Why are classes such a natural mechanism for packaging software as reusable

componentry? The chapter discusses implementing ADTs as Java-style classes, accessing class members, enforcing information hiding with **private** instance variables, separating interface from implementation, using access methods and utility methods and initializing objects with constructors (and using overloaded constructors). The chapter discusses declaring and using constant references, *composition*—the process of building classes that have as members references to objects, the **this** reference that enables an object to “know itself,” dynamic memory allocation, **static** class members for containing and manipulating class-wide data and examples of popular abstract data types such as stacks and queues. The chapter introduces the **package** statement and discusses how to create reusable packages. The chapter also introduces creating *Java archive (JAR)* files and demonstrates how to use JAR files to deploy applets that consist of multiple classes. The chapter exercises challenge the student to develop classes for complex numbers, rational numbers, times, dates, rectangles, huge integers, a class for playing Tic-Tac-Toe, a savings-account class and a class for holding sets of integers.

Chapter 9—Object-Oriented Programming—discusses the relationships among classes of objects and programming with related classes. How can we exploit commonality between classes of objects to minimize the amount of work it takes to build large software systems? What is polymorphism? What does it mean to “program in the general” rather than “program in the specific?” How does programming in the general make it easy to modify systems and add new features with minimal effort? How can we program for a whole category of objects rather than programming individually for each type of object? The chapter deals with one of the most fundamental capabilities of object-oriented programming languages, inheritance, which is a form of software reusability in which new classes are developed quickly and easily by absorbing the capabilities of existing classes and adding appropriate new capabilities. The chapter discusses the notions of superclasses and subclasses, **protected** members, direct superclasses, indirect superclasses, use of constructors in superclasses and subclasses, and software engineering with inheritance. This chapter introduces *inner classes* that help hide implementation details. Inner classes are most frequently used to create GUI event handlers. Named inner classes can be declared inside other classes and are useful in defining common event handlers for several GUI components. Anonymous inner classes are declared inside methods and are used to create one object—typically an event handler for a specific GUI component. The chapter compares inheritance (“is a” relationships) with composition (“has a” relationships). A feature of the chapter is its several substantial case studies. In particular, a lengthy case study implements a point, circle and cylinder class hierarchy. The exercises ask the student to compare the creation of new classes by inheritance vs. composition; to extend the inheritance hierarchies discussed in the chapter; to write an inheritance hierarchy for quadrilaterals, trapezoids, parallelograms, rectangles and squares and to create a more general shape hierarchy with two-dimensional shapes and three-dimensional shapes. The chapter explains polymorphic behavior. When many classes are related through inheritance to a common superclass, each subclass object may be treated as a superclass object. This enables programs to be written in a general manner independent of the specific types of the subclass objects. New kinds of objects can be handled by the same program, thus making systems more extensible. Polymorphism enables programs to eliminate complex **switch** logic in favor of simpler “straight-line” logic. A video game screen manager, for example, can send a “draw” message to every object in a linked list of objects to be drawn. Each object knows

how to draw itself. A new type of object can be added to the program without modifying that program as long as that new object also knows how to draw itself. This style of programming is typically used to implement today's popular graphical user interfaces. The chapter distinguishes between **abstract** classes (from which objects *cannot* be instantiated) and concrete classes (from which objects *can* be instantiated). The chapter also introduces interfaces—sets of methods that must be defined by any class that **implements** the interface. Interfaces are Java's replacement for the dangerous (albeit powerful) feature of C++ called multiple inheritance.

Abstract classes are useful for providing a basic set of methods and default implementation to classes throughout the hierarchy. Interfaces are useful in many situations similar to **abstract** classes; however, interfaces do not include any implementation—interfaces have no method bodies and no instance variables. A feature of the chapter is its three major polymorphism case studies—a payroll system, a shape hierarchy headed up by an **abstract** class and a shape hierarchy headed up by an interface. The chapter exercises ask the student to discuss a number of conceptual issues and approaches, work with **abstract** classes, develop a basic graphics package, modify the chapter's employee class—and pursue all these projects with polymorphic programming.

Chapter 10—Strings and Characters—deals with processing words, sentences, characters and groups of characters. The key difference between Java and C here is that Java strings are objects. This makes string manipulation more convenient and much safer than in C where string and array manipulations are based on dangerous pointers. We present classes **String**, **StringBuffer**, **Character** and **StringTokenizer**. For each, we provide extensive live-code examples demonstrating most of their methods “in action.” In all cases, we show output windows so that the reader can see the precise effects of each of the string and character manipulations. Students will enjoy the card shuffling and dealing example (which they will enhance in the exercises to the later chapters on graphics and multimedia). A key feature of the chapter is an extensive collection of challenging string-manipulation exercises related to limericks, pig Latin, text analysis, word processing, printing dates in various formats, check protection, writing the word equivalent of a check amount, Morse Code and metric-to-English conversions. Students will enjoy the challenges of developing their own spell checker and crossword-puzzle generator.

Advanced Topics

Chapters 11, 12 and 13 were coauthored with our colleague, Mr. Tem Nieto of Deitel & Associates, Inc. Tem's infinite patience, attention to detail, illustration skills and creativity are apparent throughout these chapters. [Take a fast peek at Figure 12.19 to see what happens when we turn Tem loose!]

Chapter 11—Graphics and Java2D—is the first of several chapters that present the multimedia “sizzle” of Java. We consider Chapters 11 through 22 to be the book's advanced material. This is “fun stuff.” Traditional C and C++ programming are pretty much confined to character-mode input/output. Some versions of C++ are supported by platform-dependent class libraries that can do graphics, but using these libraries makes your applications nonportable. Java's graphics capabilities are platform independent and hence, portable—and we mean portable in a worldwide sense. You can develop graphics-intensive Java applets and distribute them over the World Wide Web to colleagues everywhere, and they will run nicely on the local Java platforms. We discuss graphics contexts

and graphics objects; drawing strings, characters and bytes; color and font control; screen manipulation and paint modes and drawing lines, rectangles, rounded rectangles, three-dimensional rectangles, ovals, arcs and polygons. We introduce the Java2D API, which provides powerful graphical manipulation tools. Figure 11.22 is an example of how easy it is to use the Java2D API to create complex graphics effects such as textures and gradients. The chapter has 23 figures that painstakingly illustrate each of these graphics capabilities with live-code™ examples, appealing screen outputs, detailed features tables and detailed line art. Some of the 27 exercises challenge students to develop graphical versions of their solutions to previous exercises on Turtle Graphics, the Knight's Tour, the Tortoise and the Hare simulation, Maze Traversal and the Bucket Sort. Our companion book, *Advanced Java 2 Platform How to Program*, presents the Java 3D API.

Chapter 12—Graphical User Interface Components: Part 1—introduces the creation of applets and applications with user-friendly graphical user interfaces (GUIs). This chapter focuses on Java's *Swing GUI components*. These *platform-independent* GUI components are written entirely in Java. This provides Swing GUI components with great flexibility—the GUI components can be customized to look like the computer platform on which the program executes, or they can use the standard Java look-and-feel that provides an identical user interface across all computer platforms. GUI development is a huge topic, so we divided it into two chapters. These chapters cover the material in sufficient depth to enable you to build “industrial-strength” GUI interfaces. We discuss the **javax.swing** package, which provides much more powerful GUI components than the **java.awt** components that originated in Java 1.0. Through its 16 programs and many tables and line drawings, the chapter illustrates GUI principles, the **javax.swing** hierarchy, labels, push buttons, lists, text fields, combo boxes, checkboxes, radio buttons, panels, handling mouse events, handling keyboard events and using three of Java's simpler GUI layout managers, namely, **FlowLayout**, **BorderLayout** and **GridLayout**. The chapter concentrates on the delegation event model for GUI processing. The 33 exercises challenge the student to create specific GUIs, exercise various GUI features, develop drawing programs that let the user draw with the mouse and control fonts.

Chapter 13—Graphical User Interface Components: Part 2—continues the detailed Swing discussion started in Chapter 12. Through its 13 programs, as well as tables and line drawings, the chapter illustrates GUI design principles, the **javax.swing** hierarchy, text areas, subclassing Swing components, sliders, windows, menus, pop-up menus, changing the look-and-feel, and using three of Java's advanced GUI layout managers, namely, **BoxLayout**, **CardLayout** and **GridBagLayout**. Two of the most important examples introduced in this chapter are a program that can run as either an applet or application and a program that demonstrates how to create a *multiple document interface (MDI)* graphical user interface. MDI is a complex graphical user interface in which one window—called the *parent*—acts as the controlling window for the application. This parent window contains one or more child windows—which are always graphically displayed within the parent window. Most word processors use MDI graphical user interfaces. The chapter concludes with a series of exercises that encourage the reader to develop substantial GUIs with the techniques and components presented in the chapter. One of the most challenging exercises in this chapter is a complete drawing application that asks the reader to create an object oriented-program that keeps track of the shapes the user has drawn. Other exercises use inheritance to subclass Swing compo-

nents and reinforce layout manager concepts. The first six chapters of our companion book, *Advanced Java 2 Platform How to Program*, are designed for courses in advanced GUI programming.

Chapter 14—Exception Handling—is one of the most important chapters in the book from the standpoint of building so-called “mission-critical” or “business-critical” applications that require high degrees of robustness and fault tolerance. Things do go wrong, and at today’s computer speeds—commonly hundreds of millions operations per second (with recent personal computers running at a billion or more instructions per second)—if they can go wrong they will, and rather quickly at that. Programmers are often a bit naive about using components. They ask, “How do I request that a component do something for me?” They also ask “What value(s) does that component return to me to indicate it has performed the job I asked it to do?” But programmers also need to be concerned with, “What happens when the component I call on to do a job experiences difficulty? How will that component signal that it had a problem?” In Java, when a component (e.g., a class object) encounters difficulty, it can “throw an exception.” The environment of that component is programmed to “catch” that exception and deal with it. Java’s exception-handling capabilities are geared to an object-oriented world in which programmers construct systems largely from reusable, prefabricated components built by other programmers. To use a Java component, you need to know not only how that component behaves when “things go well,” but also what exceptions that component throws when “things go poorly.” The chapter distinguishes between rather serious system **Errors** (normally beyond the control of most programs) and **Exceptions** (which programs generally deal with to ensure robust operation). The chapter discusses the vocabulary of exception handling. The **try** block executes program code that either executes properly or **throws** an exception if something goes wrong. Associated with each **try** block are one or more **catch** blocks that handle thrown exceptions in an attempt to restore order and keep systems “up and running” rather than letting them “crash.” Even if order cannot be fully restored, the **catch** blocks can perform operations that enable a system to continue executing, albeit at reduced levels of performance—such activity is often referred to as “graceful degradation.” Regardless of whether exceptions are thrown, a **finally** block accompanying a **try** block will always execute; the **finally** block normally performs cleanup operations like closing files and releasing resources acquired in the **try** block. The material in this chapter is crucial to many of the live-code examples in the remainder of the book. The chapter enumerates many of the **Errors** and **Exceptions** of the Java packages. The chapter has some of the most appropriate quotes in the book, thanks to Barbara Deitel’s painstaking research. The vast majority of the book’s Testing and Debugging Tips emerged naturally from the material in Chapter 14.

Chapter 15—Multithreading—deals with programming applets and applications that can perform multiple activities in parallel. Although our bodies are quite good at this (breathing, eating, blood circulation, vision, hearing, etc. can all occur in parallel), our conscious minds have trouble with this. Computers used to be built with a single, rather expensive processor. Today, processors are becoming so inexpensive that it is possible to build computers with many processors that work in parallel—such computers are called *multi-processors*. The trend is clearly towards computers that can perform many tasks in parallel. Most of today’s programming languages, including C and C++, do not include built-in features for expressing parallel operations. These languages are often referred to as “sequen-

tial” programming languages or “single-thread-of-control” languages. Java includes capabilities to enable multithreaded applications (i.e., applications that can specify that multiple activities are to occur in parallel). This makes Java better prepared to deal with the more sophisticated multimedia, network-based multiprocessor-based applications programmers will develop. As we will see, multithreading is effective even on single-processor systems. For years, the “old guy” taught operating systems courses and wrote operating systems textbooks, but he never had a multithreaded language like Java available to demonstrate the concepts. In this chapter, we thoroughly enjoyed presenting multithreaded programs that demonstrate clearly the kinds of problems that can occur in parallel programming. There are all kinds of subtleties that develop in parallel programs that you simply never think about when writing sequential programs. A feature of the chapter is the extensive set of examples that show these problems and how to solve them. Another feature is the implementation of the “circular buffer,” a popular means of coordinating control between asynchronous, concurrent “producer” and “consumer” processes that, if left to run without synchronization, would cause data to be lost or duplicated incorrectly, often with devastating results. We discuss the monitor construct developed by C. A. R. Hoare and implemented in Java; this is a standard topic in operating systems courses. The chapter discusses threads and thread methods. It walks through the various thread states and state transitions with a detailed line drawing showing the life-cycle of a thread. We discuss thread priorities and thread scheduling and use a line drawing to show Java’s fixed-priority scheduling mechanism. We examine a producer/consumer relationship without synchronization, observe the problems that occur and solve the problem with thread synchronization. We implement a producer/consumer relationship with a circular buffer and proper synchronization with a monitor. We discuss daemon threads that “hang around” and perform tasks (e.g., “garbage collection”) when processor time is available. We discuss interface **Runnable** which enables objects to run as threads without having to subclass class **Thread**. We close with a discussion of thread groups which, for example, enable separation to be enforced between system threads like the garbage collector and user threads. The chapter has a nice complement of exercises. The featured exercise is the classic readers and writers problem, a favorite in upper level operating systems courses; citations appear in the exercises for students who wish to research this topic. This is an important problem in database-oriented transaction-processing systems. It raises subtle issues of solving problems in concurrency control while ensuring that every separate activity that needs to receive service does so without the possibility of “indefinite postponement,” that could cause some activities never to receive service—a condition also referred to as “starvation.” Operating systems professors will enjoy the projects implemented by Java-literate students. We can expect substantial progress in the field of parallel programming as Java’s multithreading capabilities enable large numbers of computing students to pursue parallel-programming class projects. As these students enter industry over the next several years, we expect a surge in parallel systems programming and parallel applications programming. We have been predicting this for decades—Java is making it a reality.

If this is your first Java book and you are an experienced computing professional, you may well be thinking, “Hey, this just keeps getting better and better. I can’t wait to get started programming in this language. It will let me do all kinds of stuff I would like to do, but that was never easy for me to do with the other languages I have used.” You’ve got it right. Java is an enabler. So, if you liked the multithreading discussion, hold onto your hat,

because Java will let you program multimedia applications and make them available instantaneously over the World Wide Web.

Chapter 16—Files and Streams—deals with input/output that is accomplished through streams of data directed to and from files. This is one of the most important chapters for programmers who will be developing commercial applications. Modern business is centered around data. In this chapter, we translate data (objects) into a persistent format usable by other applications. Being able to store data in files or move it across networks (Chapter 17) makes it possible for programs to save data and to communicate with each other. This is the real strength of software today. The chapter begins with an introduction to the data hierarchy from bits, to bytes, to fields, to records, to files. Next, Java's simple view of files and streams is presented. We then present a walkthrough of the dozens of classes in Java's extensive input/output files and streams class hierarchy. We put many of these classes to work in live-code examples in this chapter and in Chapter 17. We show how programs pass data to secondary storage devices, like disks, and how programs retrieve data already stored on those devices. Sequential-access files are discussed using a series of three programs that show how to open and close files, how to store data sequentially in a file and how to read data sequentially from a file. Random-access files are discussed using a series of four programs that show how to create a file sequentially for random access, how to read and write data to a file with random access and how to read data sequentially from a randomly accessed file. The fourth random-access program combines many of the techniques of accessing files both sequentially and randomly into a complete transaction-processing program. We discuss buffering and how it helps programs that do significant amounts of input/output perform better. We discuss class **File** which programs use to obtain a variety of information about files and directories. We explain how objects can be output to, and input from, secondary storage devices. Students in our industry seminars have told us that, after studying the material on file processing, they were able to produce substantial file-processing programs that were immediately useful to their organizations. The exercises ask the student to implement a variety of programs that build and process sequential-access files and random-access files.

Chapter 17—Networking—deals with applets and applications that can communicate over computer networks. This chapter presents Java's lowest level networking capabilities. We write programs that "walk the Web." The chapter examples illustrate an applet interacting with the browser in which it executes, creating a mini Web browser, communicating between two Java programs using streams-based sockets and communicating between two Java programs using packets of data. A key feature of the chapter is the live-code implementation of a collaborative client/server Tic-Tac-Toe game in which two clients play Tic-Tac-Toe with one another arbitrated by a multithreaded server—great stuff! The multithreaded server architecture is exactly what is used today in popular UNIX and Windows NT network servers. The capstone example in the chapter is the Deitel Messenger case study, which simulates many of today's popular instant-messaging applications that enable computers users to communicate with friends, relatives and coworkers over the Internet. This 1130-line, multithreaded, client/server case study uses most of the programming techniques presented up to this point in the book. The messenger application also introduces multicasting, which enables a program to send packets of data to groups of clients. The chapter has a nice collection of exercises including several suggested modifications to the multithreaded server example. Our companion book, *Advanced Java 2 Platform*

How to Program, offers a much deeper treatment of networking and distributed computing, with topics including remote method invocation (RMI), servlets, JavaServer Pages (JSP), Java 2 Enterprise Edition, wireless Java (and the Java 2 Micro Edition) and Common Object Request Broker Architecture (CORBA).

Chapter 18—Multimedia: Images, Animation and Audio—is the first of two chapters that present Java’s capabilities for making computer applications come alive (Chapter 22 offers an extensive treatment of the Java Media Framework). It is remarkable that students in first programming courses will be writing applications with all these capabilities. The possibilities are intriguing. Imagine having access (over the Internet and through CD-ROM technology) to vast libraries of graphics images, audios and videos and being able to weave your own together with those in the libraries to form creative applications. Already, most new computers sold come “multimedia equipped.” Students can create extraordinary term papers and classroom presentations with components drawn from vast public-domain libraries of images, line drawings, voices, pictures, videos, animations and the like. A “paper” when many of us were in the earlier grades was a collection of characters, possibly handwritten, possibly typewritten. A “paper” today can be a multimedia “extravaganza” that makes the subject matter come alive. It can hold your interest, pique your curiosity and make you feel what the subjects of the paper felt when they were making history. Multimedia is making science labs much more exciting. Textbooks are coming alive. Instead of looking at a static picture of some phenomenon, you can watch that phenomenon occur in a colorful, animated, presentation with sounds, videos and various other effects, leveraging the learning process. People are able to learn more, learn it in more depth and experience more viewpoints.

The chapter discusses images and image manipulation, audios and animation. A feature of the chapter is the image maps that enable a program to sense the presence of the mouse pointer over a region of an image, without clicking the mouse. We present a live-code image-map application with the icons from the programming tips you have seen in this chapter and will see throughout the book. As the user moves the mouse pointer across the seven icon images, the type of tip is displayed, either “Good Programming Practice” for the thumbs-up icon, “Portability Tip” for the bug with the suitcase icon and so on. Once you have read the chapter, you will be eager to try out all these techniques, so we have included 35 problems to challenge and entertain you (more are provided in Chapter 22). Here are some of the exercises that you may want to turn into term projects:

<i>15 Puzzle</i>	<i>Game of Pool</i>	<i>One-Armed Bandit</i>
<i>Analog Clock</i>	<i>Horse Race</i>	<i>Random Inter-Image Transition</i>
<i>Animation</i>	<i>Image Flasher</i>	<i>Randomly Erasing an Image</i>
<i>Artist</i>	<i>Image Zooming</i>	<i>Reaction Time Tester</i>
<i>Calendar/Tickler File</i>	<i>Jigsaw Puzzle Generator</i>	<i>Rotating Images</i>
<i>Calling Attention to an Image</i>	<i>Kaleidoscope</i>	<i>Scrolling Image Marquee</i>
<i>Coloring Black and White Images</i>	<i>Limericks</i>	<i>Scrolling Text Marquee</i>
<i>Crossword</i>	<i>Maze Generator and Walker</i>	<i>Shuffleboard</i>
<i>Fireworks Designer</i>	<i>Multimedia Simpletron Simulator</i>	<i>Text Flasher</i>

You are going to have a great time attacking these problems! Some will take a few hours and some are great term projects. We see all kinds of opportunities for multimedia electives starting to appear in the university computing curriculum. We hope you will have contests with your classmates to develop the best solutions to several of these problems.

Chapter 19—Data Structures—is particularly valuable in second- and third-level university courses. The chapter discusses the techniques used to create and manipulate dynamic data structures, such as linked lists, stacks, queues (i.e., waiting lines) and trees. The chapter begins with discussions of self-referential classes and dynamic memory allocation. We proceed with a discussion of how to create and maintain various dynamic data structures. For each type of data structure, we present live-code programs and show sample outputs. Although it is valuable to know how these classes are implemented, Java programmers will quickly discover that many of the data structures they need are already available in class libraries, such as Java's own **java.util** that we discuss in Chapter 20 and Java **Collections** that we discuss in Chapter 21. The chapter helps the student master Java-style references (i.e., Java's replacement for the more dangerous pointers of C and C++). One problem when working with references is that students could have trouble visualizing the data structures and how their nodes are linked together. So we present illustrations that show the links and the sequence in which they are created. The binary tree example is a nice capstone for the study of references and dynamic data structures. This example creates a binary tree; enforces duplicate elimination and introduces recursive preorder, inorder and postorder tree traversals. Students have a genuine sense of accomplishment when they study and implement this example. They particularly appreciate seeing that the inorder traversal prints the node values in sorted order. The chapter includes a substantial collection of exercises. A highlight of the exercises is the special section "Building Your Own Compiler." This exercise is based on earlier exercises that walk the student through the development of an infix-to-postfix conversion program and a postfix-expression evaluation program. We then modify the postfix evaluation algorithm to generate machine-language code. The compiler places this code in a file (using techniques the student mastered in Chapter 16). Students then run the machine language produced by their compilers on the software simulators they built in the exercises of Chapter 7! The many exercises include a supermarket simulation using queueing, recursively searching a list, recursively printing a list backwards, binary tree node deletion, level-order traversal of a binary tree, printing trees, writing a portion of an optimizing compiler, writing an interpreter, inserting/deleting anywhere in a linked list, analyzing the performance of binary tree searching and sorting and implementing an indexed list class.

Chapter 20—Java Utilities Package and Bit Manipulation—walks through the classes of the **java.util** package and discusses each of Java's bitwise operators. This is a nice chapter for reinforcing the notion of reuse. When classes already exist, it is much faster to develop software by simply reusing these classes than by "reinventing the wheel." Classes are included in class libraries because the classes are generally useful, correct, performance tuned, portability certified and/or for a variety of other reasons. Someone has invested considerable work in preparing these classes so why should you write your own? The world's class libraries are growing at a phenomenal rate. Given this, your skill and value as a programmer will depend on your familiarity with what classes exist and how you can reuse them cleverly to develop high-quality software rapidly. University data structures courses will be changing drastically over the next several years because most important data structures are already implemented in widely available class libraries. This chapter discusses many classes. Two of the most useful are **Vector** (a dynamic array that can grow and shrink as necessary) and **Stack** (a dynamic data structure that allows insertions and deletions from only one end—called the top—thus ensuring last-in-first-out behavior). The

beauty of studying these two classes is that they are related through inheritance, as is discussed in Chapter 9, so the **java.util** package itself implements some classes in terms of others, thus avoiding reinventing the wheel and taking advantage of reuse. We also discuss classes **Dictionary**, **Hashtable**, **Properties** (for creating and manipulating persistent **Hashtables**), **Random** and **BitSet**. The discussion of **BitSet** includes live code for one of the classic applications of **BitSets**, namely the *Sieve of Eratosthenes*, used for determining prime numbers. The chapter discusses in detail Java's powerful bit-manipulation capabilities, which enable programmers to exercise lower level hardware capabilities. This helps programs process bit strings, set individual bits on or off and store information more compactly. Such capabilities—inherited from C—are characteristic of low-level assembly languages and are valued by programmers writing system software such as operating systems and networking software.

Chapter 21—Collections—discusses many of the Java 2 classes (of the **java.util** package) that provide predefined implementations of many of the data structures discussed in Chapter 19. This chapter, too, reinforces the notion of reuse. These classes are modeled after a similar class library in C++—the Standard Template Library. Collections provide Java programmers with a standard set of data structures for storing and retrieving data and a standard set of algorithms (i.e., procedures) that allow programmers to manipulate the data (such as searching for particular data items and sorting data into ascending or descending order). The chapter examples demonstrate collections, such as linked lists, trees, maps and sets, and algorithms for searching, sorting, finding the maximum value, finding the minimum value and so on. Each example clearly shows how powerful and easy to use collections are. The exercises suggest modifications to the chapter examples and ask the reader to reimplement data structures presented in Chapter 19 using collections.

Chapter 22—Java Media Framework and Java Sound—is the second of our two chapters dedicated to Java's tremendous multimedia capabilities. This chapter focusses on the Java Media Framework (JMF) and the Java Sound API. The Java Media Framework provides both audio and video capabilities. With the JMF, a Java program can play audio and video media and capture audio and video media from devices such as microphones and video cameras. Many of today's multimedia applications involve sending audio or video feeds across the Internet. For example, you can visit the **cnn.com** Web site to watch or listen to live news conferences, and many people listen to Internet-based radio stations through their Web browsers. The JMF enables Java developers to create so-called *streaming media* applications, in which a Java program sends live or recorded audio or video feeds across the Internet to other computers, then applications on those other computers play the media as it arrives over the network. The JavaSound APIs enable programs to manipulate Musical Instrument Digital Interface (MIDI) sounds and captured media (i.e., media from a device such as a microphone). This chapter concludes with a substantial MIDI-processing application that enables users to select MIDI files to play and record new MIDI files. Users can create their own MIDI music by interacting with the application's simulated synthesizer keyboard. In addition, the application can synchronize playing the notes in a MIDI file with pressing the keys on the simulated synthesizer keyboard—similar to a player piano! As with Chapter 18, once you read this chapter, you will be eager to try all these techniques, so we have included 44 additional multimedia exercises to challenge and entertain you. Some of the interesting projects include the following:

<i>Bouncing Ball Physics Demo</i>	<i>Knight's Tour Walker</i>	<i>Story Teller</i>
<i>Craps</i>	<i>Morse Code</i>	<i>Tic-Tac-Toe</i>
<i>Digital Clock</i>	<i>MP3 Player</i>	<i>Tortoise and the Hare</i>
<i>Flight Simulator</i>	<i>Multimedia Authoring System</i>	<i>Towers of Hanoi</i>
<i>Karaoke</i>	<i>Pinball Machine</i>	<i>Video Conferencing</i>
<i>Kinetics Physics Demo</i>	<i>Roulette</i>	<i>Video Games</i>

Appendix A—Java Demos—presents a huge collection of some of the best Java demos available on the Web. Many of these sites make their source code available to you, so you can download the code and add your own features—a truly great way to learn Java! We encourage our students to do this, and we're amazed at the results! You should start your search by checking out the Sun Microsystems applet Web page, java.sun.com/applets. You can save time finding the best demos by checking out JARS (the Java Applet Rating Service) at www.jars.com. Here's a list of some of the demos mentioned in Appendix A (the URLs and descriptions of each are in Appendix A):

<i>Animated SDSU Logo</i>	<i>Java Game Park</i>	<i>Sevilla RDM 168</i>
<i>Bumpy Lens 3D</i>	<i>Java4fun games</i>	<i>Stereoscopic 3D Hypercube</i>
<i>Centipede</i>	<i>Missile Commando</i>	<i>Teamball demos</i>
<i>Crazy Counter</i>	<i>PhotoAlbum II</i>	<i>Tube</i>
<i>Famous Curves Applet Index</i>	<i>Play A Piano</i>	<i>Urbanoids</i>
<i>Goldmine</i>	<i>Sab's Game Arcade</i>	<i>Warp 1.5</i>
<i>Iceblox game</i>	<i>SabBowl bowling game</i>	

Appendix B—Java Resources—presents some of the best Java resources available on the Web. This is a great way for you to get into the “world of Java.” The appendix lists various Java resources, such as consortia, journals and companies that make various key Java-related products. Here are some of the resources mentioned in Appendix B:

animated applets	Intelligence.com	newsgroups
applets/applications	Java Applet Rating Service	newsletters
arts and entertainment	Java Developer Connection	Object Management Group
audio sites	Java Developer's Journal	products
books	Java Media Framework	projects
Borland JBuilder IDE	Java Report	publications
conferences	Java tools	puzzles
consultants	Java Toys	reference materials
contests	Java Users Group (JUGs)	resources
CORBA homepage	Java Woman	seminars
current information	java.sun.com	sites
databases	JavaWorld on-line magazine	software
demos (many with source code)	learning Java	Sun Microsystems
developer's kit	links to Java sites	SunWorld on-line magazine
development tools	lists of resources	Team Java
discussion groups	lists of what is new and cool	The Java Tutorial
documentation	live chat sessions on Java	trade shows
downloadable applets	multimedia collections	training (please call us!)
FAQs (frequently asked ?s)	NASA multimedia gallery	tutorials for learning java
games	NetBeans IDE	URLs for Java applets
graphics	news	www.javaworld.com
IBM Developers Java Zone	news:comp.lang.java	Yahoo (Web search engine)

Appendix C—Operator Precedence Chart—lists each of the Java operators and indicates their relative precedence and associativity. We list each operator on a separate line and include the full name of the operator.

Appendix D—ASCII Character Set—lists the characters of the ASCII (American Standard Code for Information Interchange) character set and indicates the character code value for each. Java uses the Unicode character set with 16-bit characters for representing all of the characters in the world’s “commercially significant” languages. Unicode includes ASCII as a subset. Currently, most English-speaking countries are using ASCII and just beginning to experiment with Unicode.

Appendix E—Number Systems—discusses the binary (base 2), decimal (base 10), octal (base 8) and hexadecimal (base 16) number systems. This material is valuable for introductory courses in computer science and computer engineering. The appendix is presented with the same pedagogic learning aids as the chapters of the book. A nice feature of the appendix is its 31 exercises, 19 of which are self-review exercises with answers.

Appendix F—Creating `javadoc` Documentation—introduces the `javadoc` documentation-generation tool. Sun Microsystems uses `javadoc` to document the Java APIs. The example in this appendix takes the reader through the `javadoc` documentation process. First, we introduce the comment style and tags that `javadoc` recognizes and uses to create documentation. Next, we discuss the commands and options used to run the utility. Finally, we examine the source files `javadoc` uses and the HTML files `javadoc` creates.

1.18 (Optional) A Tour of the Case Study on Object-Oriented Design with the UML

In this and the next section, we tour the two optional major features of the book—the optional case study of object-oriented design with the UML and our introduction to design patterns. The case study involving object-oriented design with the UML is an important addition to *Java How to Program, Fourth Edition*. This tour previews the contents of the “Thinking About Objects” sections and discusses how they relate to the case study. After completing this case study, you will have completed an object-oriented design and implementation for a significant Java application.

Section 1.15—Thinking About Objects: Introduction to Object Technology and the Unified Modeling Language

This section introduces the object-oriented design case study with the UML. We provide a general background of what objects are and how they interact with other objects. We also discuss briefly the state of the software-engineering industry and how the UML has influenced object-oriented analysis and design processes.

Section 2.9—(Optional Case Study) Thinking About Objects: Examining the Problem Statement

Our case study begins with a *problem statement* that specifies the requirements for a system that we will create. In this case study, we design and implement a simulation of an elevator system in a two-story building. The application user can “create” a person on either floor. This person then walks across the floor to the elevator, presses a button, waits for the elevator to arrive and rides it to the other floor. We provide the design of our elevator system after investigating the structure and behavior of object-oriented systems in general. We dis-

cuss how the UML will facilitate the design process in subsequent “Thinking About Object” sections by providing us with several types of diagrams to model our system. Finally, we provide a list of URL and book references on object-oriented design with the UML. You might find these references helpful as you proceed through our case-study presentation.

Section 3.8—(Optional Case Study) Thinking About Objects: Identifying the Classes in the Problem Statement

In this section, we design the elevator-simulation model, which represents the operations of the elevator system. We identify the classes, or “building blocks,” of our model by extracting the nouns and noun phrases from the problem statement. We arrange these classes into a UML class diagram that describes the class structure of our model. The class diagram also describes relationships, known as *associations*, among classes (for example, a person has an association with the elevator, because the person rides the elevator). Lastly, we extract from the class diagram another type of diagram in the UML—the object diagram. The object diagram models the objects (instances of classes) at a specific time in our simulation.

Section 4.14—(Optional Case Study) Thinking About Objects: Identifying Class Attributes

A class contains both *attributes* (data) and *operations* (behaviors). This section focuses on the attributes of the classes discussed in Section 3.7. As we see in later sections, changes in an object’s attributes often affect the behavior of that object. To determine the attributes for the classes in our case study, we extract the adjectives describing the nouns and noun phrases (which defined our classes) from the problem statement, then place the attributes in the class diagram we create in Section 3.7.

Section 5.11—(Optional Case Study) Thinking About Objects: Identifying Objects’ States and Activities

An object, at any given time, occupies a specific condition called a *state*. A *state transition* occurs when that object receives a message to change state. The UML provides the *state-chart diagram*, which identifies the set of possible states that an object may occupy and models that object’s state transitions. An object also has an *activity*—the work performed by an object in its lifetime. The UML provides the *activity diagram*—a flowchart that models an object’s *activity*. In this section, we use both types of diagrams to begin modeling specific behavioral aspects of our elevator simulation, such as how a person rides the elevator and how the elevator responds when a button is pressed on a given floor.

Section 6.16—(Optional Case Study) Thinking About Objects: Identifying Class Operations

In this section, we identify the operations, or services, of our classes. We extract from the problem statement the verbs and verb phrases that specify the operations for each class. We then modify the class diagram of Fig. 3.16 to include each operation with its associated class. At this point in the case study, we will have gathered all information possible from the problem statement. However, as future chapters introduce such topics as inheritance, event-handling and multithreading, we will modify our classes and diagrams.

Section 7.10—(Optional Case Study) Thinking About Objects: Collaboration Among Objects

At this point, we have created a “rough sketch” of the model for our elevator system. In this section, we see how it works. We investigate the behavior of the model by discussing *col-*

laborations—messages that objects send to each other to communicate. The class operations that we discovered in Section 6.16 turn out to be the collaborations among the objects in our system. We determine the collaborations in our system, then collect them into a *collaboration diagram*—the UML diagram for modeling collaborations. This diagram reveals which objects collaborate and when. We present a collaboration diagram of the people entering and exiting the elevator.

Section 8.17—(Optional Case Study) Thinking About Objects: Starting to Program the Classes for the Elevator Simulation

In this section, we take a break from designing the behavior of our system. We begin the implementation process to emphasize the material discussed in Chapter 8. Using the UML class diagram of Section 3.7 and the attributes and operations discussed in Sections 4.14 and 6.16, we show how to implement a class in Java from a design. We do not implement all classes—because we have not completed the design process. Working from our UML diagrams, we create code for the **Elevator** class.

Section 9.23—(Optional Case Study) Thinking About Objects: Incorporating Inheritance into the Elevator Simulation

Chapter 9 begins our discussion of object-oriented programming. We consider inheritance—classes sharing similar characteristics may inherit attributes and operations from a “base” class. In this section, we investigate how our elevator simulation can benefit from using inheritance. We document our discoveries in a class diagram that models inheritance relationships—the UML refers to these relationships as *generalizations*. We modify the class diagram of Section 3.7 by using inheritance to group classes with similar characteristics. We continue implementing the **Elevator** class of Section 8.17 by incorporating inheritance.

Section 10.22—(Optional Case Study) Thinking About Objects: Event Handling

In this section, we include interfaces necessary for the objects in our elevator simulation to send messages to other objects. In Java, objects often communicate by sending an *event*—a notification that some action has occurred. The object receiving the event then performs an action in response to the type of event received—this is known as *event handling*. In Section 7.10, we outlined the message passing, or the collaborations, in our model, using a collaboration diagram. We now modify this diagram to include event handling, and, as an example, we explain in detail how doors in our simulation open upon the elevator’s arrival.

Section 11.10—(Optional Case Study) Thinking About Objects: Designing Interfaces with the UML

In this section, we design a class diagram that models the relationships between classes and interfaces in our simulation—the UML refers to these relationships as *realizations*. In addition, we list all operations that each interface provides to the classes. Lastly, we show how to create the Java classes that implement these interfaces. As in Section 8.17 and Section 9.23, we use class **Elevator** to demonstrate the implementation.

Section 12.16 - (Optional Case Study) Thinking About Objects: Use Cases

Chapter 12 discusses user interfaces that enable a user to interact with a program. In this section, we discuss the interaction between our elevator simulation and its user. Specifically, we investigate the scenarios that may occur between the application user and the simu-

lation itself—this set of scenarios is called a *use case*. We model these interactions, using *use-case diagrams* of the UML. We then discuss the graphical user interface for our simulation, using our use-case diagrams.

Section 13.17—(Optional Case Study) Thinking About Objects: Model-View-Controller

We designed our system to consist of three components, each having a distinct responsibility. By this point in the case study, we have almost completed the first component, called the *model*, which contains data that represent the simulation. We design the *view*—the second component, dealing with how the model is displayed—in Section 22.8. We design the *controller*—the component that allows the user to control the model—in Section 12.16. A system such as ours that uses the model, view and controller components is said to adhere to *Model-View-Controller (MVC)* architecture. In this section, we explain the advantages of using this architecture to design software. We use the UML *component diagram* to model the three components, then implement this diagram as Java code.

Section 15.12—(Optional Case Study) Thinking About Objects: Multithreading

In the real world, objects operate and interact concurrently. Java is a *multithreaded* language, which enables the objects in our simulation to act seemingly independently from each other. In this section, we declare certain objects as “threads” to enable these objects to operate concurrently. We modify the collaboration diagram originally presented in Section 7.10 (and modified in Section 10.22) to incorporate multithreading. We present the UML *sequence diagram* for modeling interactions in a system. This diagram emphasizes the chronological ordering of messages. We use a sequence diagram to model how a person inside the simulation interacts with the elevator. This section concludes the design of the model portion of our simulation. We design how this model is displayed in Section 22.9, then implement this model as Java code in Appendix H.

Section 22.9—(Optional Case Study) Thinking About Objects: Animation and Sound in the View

This section designs the view, which specifies how the model portion of the simulation is displayed. Chapter 18 presents several techniques for integrating animation in programs, and Chapter 22 presents techniques for integrating sound. Section 22.9 uses some of these techniques to incorporate sound and animation into our elevator simulation. Specifically, this section deals with animating the movements of people and our elevator, generating sound effects and playing “elevator music” when a person rides the elevator. This section concludes the design of our elevator simulation. Appendices G, H and I implement this design as a 3,594-line, fully operational Java program.

Appendix G—Elevator Events and Listener Interfaces

[Note: This appendix is on the CD that accompanies this book.] As we discussed in Section 10.22, several objects in our simulation interact with each other by sending messages, called events, to other objects wishing to receive these events. The objects receiving the events are called *listener objects*—these must implement *listener interfaces*. In this section, we implement all event classes and listener interfaces used by the objects in our simulation.

Appendix H—Elevator Model

[Note: This appendix is on the CD that accompanies this book.] The majority of the case study involved designing the model (i.e., the data and logic) of the elevator simulation. In

this section, we implement that model in Java. Using all the UML diagrams we created, we present the Java classes necessary to implement the model. We apply the concepts of object-oriented design with the UML and object-oriented programming and Java that you learned in the chapters.

Appendix I—Elevator View

[Note: This appendix is on the CD that accompanies this book.] The final section implements how we display the model from Appendix H. We use the same approach to implement the view as we used to implement the model—we create all the classes required to run the view, using the UML diagrams and key concepts discussed in the chapters. By the end of this section, you will have completed an “industrial-strength” design and implementation of a large-scale system. You should feel confident tackling larger systems, such as the 8000-line Enterprise Java case study we present in our companion book *Advanced Java 2 Platform How to Program* and the kinds of applications that professional software engineers build. Hopefully, you will move on to even deeper study of object-oriented design with the UML.

1.19 (Optional) A Tour of the “Discovering Design Patterns” Sections

Our treatment of design patterns is spread over five optional sections of the book. We overview those sections here.

Section 9.24—(Optional) Discovering Design Patterns: Introducing Creational, Structural and Behavioral Design Patterns

This section provides tables that list the sections in which we discuss the various design patterns. We divide the discussion of each section into creational, structural and behavioral design patterns. Creational patterns provide ways to instantiate objects, structural patterns deal with organizing objects and behavioral patterns deal with interactions between objects. The remainder of the section introduces some of these design patterns, such as the Singleton, Proxy, Memento and State design patterns. Finally, we provide several URLs for further study on design patterns.

Section 13.18—(Optional) Discovering Design Patterns: Design Patterns Used in Packages `java.awt` and `javax.swing`

This section contains most of our design-patterns discussion. Using the material on Java Swing GUI components in Chapters 12 and 13, we investigate some examples of pattern use in packages `java.awt` and `javax.swing`. We discuss how these classes use the Factory Method, Adapter, Bridge, Composite, Chain-of-Responsibility, Command, Observer, Strategy and Template Method design patterns. We motivate each pattern and present examples of how to apply them.

Section 15.13—(Optional) Discovering Design Patterns: Concurrent Design Patterns

Developers have introduced several design patterns since those described by the gang of four. In this section, we discuss concurrency design patterns, including Single-Threaded Execution, Guarded Suspension, Balking, Read/Write Lock and Two-Phase Termination—these solve various design problems in multithreaded systems. We investigate how class `java.lang.Thread` uses concurrency patterns.

Section 17.11—(Optional) Discovering Design Patterns: Design Patterns Used in Packages `java.io` and `java.net`

Using the material on files, streams and networking in Chapters 16 and 17, we investigate some examples of pattern use in packages `java.io` and `java.net`. We discuss how these classes use the Abstract Factory, Decorator and Facade design patterns. We also consider architectural patterns, which specify a set of subsystems—aggregates of objects that each collectively comprise a major system responsibility—and how these subsystems interact with each other. We discuss the popular Model-View-Controller and Layers architectural patterns.

Section 21.12—(Optional) Discovering Design Patterns: Design Patterns Used in Package `java.util`

Using the material on data structures and collections in Chapters 19, 20 and 21, we investigate pattern use in package `java.util`. We discuss how these classes use the Prototype and Iterator design patterns. This section concludes the discussion on design patterns. After finishing the *Discovering Design Patterns* material, you should be able to recognize and use key design patterns and have a better understanding of the workings of the Java API. After completing this material, we recommend that you move on to the gang-of-four book.

Well, there you have it! We have worked hard to create this book and its optional Cyber Classroom version. The book is loaded with live-code examples, programming tips, self-review exercises and answers, challenging exercises and projects, and numerous study aids to help you master the material. Java is a powerful programming language that will help you write programs quickly and effectively. And Java is a language that scales nicely into the realm of enterprise-systems development to help organizations build their key information systems. As you read the book, if something is not clear, or if you find an error, please write to us at deitel@deitel.com. We will respond promptly, and we will post corrections and clarifications on our Web site,

www.deitel.com

We hope you enjoy learning with *Java How to Program: Fourth Edition* as much as we enjoyed writing it!

SUMMARY

- Software controls computers (often referred to as hardware).
- Java is one of today's most popular software-development languages.
- Java was developed by Sun Microsystems. Sun provides an implementation of the Java 2 Platform, Standard Edition called the Java 2 Software Development Kit (J2SDK), version 1.3.1 that includes the minimum set of tools you need to write software in Java.
- Java is a fully object-oriented language with strong support for proper software-engineering techniques.
- A computer is a device capable of performing computations and making logical decisions at speeds millions, even billions, of times faster than human beings can.
- Computers process data under the control of sets of instructions called computer programs. These computer programs guide the computer through orderly sets of actions specified by people called computer programmers.

- The various devices that comprise a computer system (such as the keyboard, screen, disks, memory and processing units) are referred to as hardware.
- The computer programs that run on a computer are referred to as software.
- The input unit is the “receiving” section of the computer. It obtains information (data and computer programs) from various input devices and places this information at the disposal of the other units so that the information may be processed.
- The output unit is the “shipping” section of the computer. It takes information processed by the computer and places it on output devices to make it available for use outside the computer.
- The memory unit is the rapid access, relatively low-capacity “warehouse” section of the computer. It retains information that has been entered through the input unit so that the information may be made immediately available for processing when it is needed and retains information that has already been processed until that information can be placed on output devices by the output unit.
- The arithmetic and logic unit (ALU) is the “manufacturing” section of the computer. It is responsible for performing calculations such as addition, subtraction, multiplication and division and for making decisions.
- The central processing unit (CPU) is the “administrative” section of the computer. It is the computer’s coordinator and is responsible for supervising the operation of the other sections.
- The secondary storage unit is the long-term, high-capacity “warehousing” section of the computer. Programs or data not being used by the other units are normally placed on secondary storage devices (such as disks) until they are needed, possibly hours, days, months or even years later.
- Early computers were capable of performing only one job or task at a time. This form of computer operation often is called single-user batch processing.
- Software systems called operating systems were developed to help make it more convenient to use computers. Early operating systems managed the smooth transition between jobs and minimized the time it took for computer operators to switch between jobs.
- Multiprogramming involves the “simultaneous” operation of many jobs on the computer—the computer shares its resources among the jobs competing for its attention.
- Timesharing is a special case of multiprogramming in which dozens or even hundreds of users share a computer through terminals. The computer runs a small portion of one user’s job, then moves on to service the next user. The computer does this so quickly that it might provide service to each user several times per second, so programs appear to run simultaneously.
- An advantage of timesharing is that the user receives almost immediate responses to requests rather than having to wait long periods for results, as with previous modes of computing.
- In 1977, Apple Computer popularized the phenomenon of personal computing.
- In 1981, IBM introduced the IBM Personal Computer. Almost overnight, personal computing became legitimate in business, industry and government organizations.
- Although early personal computers were not powerful enough to timeshare several users, these machines could be linked together in computer networks, sometimes over telephone lines and sometimes in local area networks (LANs) within an organization. This led to the phenomenon of distributed computing, in which an organization’s computing is distributed over networks to the sites at which the real work of the organization is performed.
- Today, information is shared easily across computer networks where some computers called file servers offer a common store of programs and data that may be used by client computers distributed throughout the network—hence the term client/server computing.
- Java has become the language of choice for developing Internet-based applications (and for many other purposes).

- Computer languages may be divided into three general types: machine languages, assembly languages and high-level languages.
- Any computer can directly understand only its own machine language. Machine languages generally consist of strings of numbers (ultimately reduced to 1s and 0s) that instruct computers to perform their most elementary operations one at a time. Machine languages are machine dependent.
- English-like abbreviations formed the basis of assembly languages. Translator programs called assemblers convert assembly-language programs to machine language at computer speeds.
- Compilers translate high-level language programs into machine-language programs. High-level languages (like Java) contain English words and conventional mathematical notations.
- Interpreter programs directly execute high-level language programs without the need for compiling those programs into machine language.
- Although compiled programs execute much faster than interpreted programs, interpreters are popular in program-development environments in which programs are recompiled frequently as new features are added and errors are corrected.
- Objects are essentially reusable software components that model items in the real world. Modular, object-oriented design and implementation approaches make software-development groups more productive than is possible with previous popular programming techniques such as structured programming. Object-oriented programs are often easier to understand, correct and modify.
- Java originated at Sun Microsystems as a project for intelligent consumer-electronic devices.
- When the World Wide Web exploded in popularity in 1993, Sun people saw the immediate potential of using Java to create Web pages with so-called dynamic content.
- Java is now used to create Web pages with dynamic and interactive content, to develop large-scale enterprise applications, to enhance the functionality of Web servers, to provide applications for consumer devices and so on.
- Java programs consist of pieces called classes. Classes consist of pieces called methods that perform tasks and return information when they complete their tasks.
- Most Java programmers use rich collections of existing classes in Java class libraries.
- FORTRAN (FORMula TRANslator) was developed by IBM Corporation between 1954 and 1957 for scientific and engineering applications that require complex mathematical computations.
- COBOL (Common Business Oriented Language) was developed in 1959 by a group of computer manufacturers and government and industrial computer users. COBOL is used primarily for commercial applications that require precise and efficient manipulation of large amounts of data.
- Pascal was designed at about the same time as C. It was created by Professor Nicklaus Wirth and was intended for academic use.
- Basic was developed in 1965 at Dartmouth College as a simple language to help novices become comfortable with programming.
- Structured programming is a disciplined approach to writing programs that are clearer than unstructured programs, easier to test and debug and easier to modify.
- The Ada language was developed under the sponsorship of the United States Department of Defense (DOD) during the 1970s and early 1980s. One important capability of Ada is called multitasking; this allows programmers to specify that many activities are to occur in parallel.
- Most high-level languages—including C and C++—generally allow the programmer to write programs that perform only one activity at a time. Java, through a technique called multithreading, enables programmers to write programs with parallel activities.
- The Internet was developed more than three decades ago with funding supplied by the Department of Defense. Originally designed to connect the main computer systems of about a dozen universi-

ties and research organizations, the Internet today is accessible by hundreds of millions of computers worldwide.

- The Web allows computer users to locate and view multimedia-intensive documents over the Internet.
- Java systems generally consist of several parts: an environment, the language, the Java Applications Programming Interface (API) and various class libraries.
- Java programs normally go through five phases to be executed—edit, compile, load, verify and execute.
- Java program file names end with the **.java** extension.
- The Java compiler (**javac**) translates a Java program into bytecodes—the language understood by the Java interpreter. If a program compiles correctly, the compiler produces a file with the **.class** extension. This is the file containing the bytecodes that are interpreted during the execution phase.
- A Java program must first be placed in memory before it can execute. This is done by the class loader, which takes the **.class** file (or files) containing the bytecodes and transfers it to memory. The **.class** file can be loaded from a disk on your system or over a network.
- An application is a program that is normally stored and executed on the user's local computer.
- An applet is a small program that is normally stored on a remote computer that users connect to via a Web browser. Applets are loaded from a remote computer into the browser, executed in the browser and discarded when execution completes.
- Applications are loaded into memory, then executed by the **java** interpreter.
- Browsers are used to view HTML (Hypertext Markup Language) documents on the World Wide Web.
- When the browser sees an applet in an HTML document, the browser launches the Java class loader to load the applet. The browsers that support Java each have a built-in Java interpreter. Once the applet is loaded, the Java interpreter in the browser begins executing the applet.
- Applets can also be executed from the command line using the **appletviewer** command provided with the Java 2 Software Development Kit (J2SDK). The **appletviewer** is commonly referred to as the minimum browser—it knows only how to interpret applets.
- Before the bytecodes in an applet are executed by the Java interpreter built into a browser or the **appletviewer**, they are verified by the bytecode verifier to ensure that the bytecodes for downloaded classes are valid and that they do not violate Java's security restrictions.
- An intermediate step between interpreters and compilers is a just-in-time (JIT) compiler that, as the interpreter runs, produces compiled code for the programs and executes the programs in machine language rather than reinterpreting them. JIT compilers do not produce machine language that is as efficient as a full compiler.
- For organizations wanting to do heavy-duty information-systems development, Integrated Development Environments (IDEs) are available from the major software suppliers. The IDEs provide many tools for supporting the software-development process.
- Object orientation is a natural way of thinking about the world and of writing computer programs.
- The Unified Modeling Language (UML) is a graphical language that allows people who build systems to represent their object-oriented designs in a common notation.
- Humans think in terms of objects. We possess the marvelous ability of abstraction, which enables us to view screen images as people, planes, trees and mountains rather than as individual dots of color (called pixels—for "picture elements").

- Humans learn about objects by studying their attributes and observing their behaviors. Different objects can have similar attributes and can exhibit similar behaviors.
- Object-oriented design (OOD) models real-world objects. It takes advantage of class relationships, where objects of a certain class—such as a class of vehicles—have the same characteristics. It takes advantage of inheritance relationships, and even multiple-inheritance relationships, where newly created classes of objects are derived by absorbing characteristics of existing classes and adding unique characteristics of their own.
- OOD encapsulates data (attributes) and functions (behavior) into objects; the data and functions of an object are intimately tied together.
- Objects have the property of information hiding. This means that, although objects may know how to communicate with one another across well-defined interfaces, objects normally are not allowed to know how other objects are implemented.
- Languages such as Java are object-oriented—programming in such a language is called object-oriented programming (OOP) and allows designers to implement the object-oriented design as a working system.
- In Java, the unit of programming is the class from which objects are eventually instantiated (a fancy term for “created”). Java classes contain methods (which implement class behaviors) and attributes (which implement class data).
- Java programmers concentrate on creating their own user-defined types, called classes. Each class contains data and the set of functions that manipulate that data. The data components of a Java class are called attributes. The function components of a Java class are called methods.
- An instance of a user-defined type (i.e., a class) is called an object.
- Classes can also have relationships with other classes. These relationships are called associations.
- With object technology, we can build much of the software we will need by combining “standardized, interchangeable parts” called classes.
- The process of analyzing and designing a system from an object-oriented point of view is called object-oriented analysis and design (OOAD).
- The Unified Modeling Language (the UML) is now the most widely used graphical representation scheme for modeling object-oriented systems. Those who design systems use the language (in the form of graphical diagrams) to model their systems.
- Over the past decade, the software-engineering industry has made significant progress in the field of design patterns—proven architectures for constructing flexible and maintainable object-oriented software. Using design patterns can substantially reduce the complexity of the design process.
- Design patterns benefit system developers by helping to construct reliable software using proven architectures and accumulated industry expertise, promoting design reuse in future systems, identifying common mistakes and pitfalls that occur when building systems, helping to design systems independently of the language in which they will be implemented, establishing a common design vocabulary among developers and shortening the design phase in a software-development process.
- Designers use design patterns to construct sets of classes and objects.
- Creational design patterns describe techniques to instantiate objects (or groups of objects).
- Structural design patterns allow designers to organize classes and objects into larger structures.
- Behavioral design patterns assign responsibilities to objects.

TERMINOLOGY

abstraction
Ada

ALU (arithmetic and logic unit)
ANSI C

applet	IDE (Integrated Development Environment)
appletviewer command	information hiding
application	inheritance
arithmetic and logic unit (ALU)	input device
array	input unit
assembly language	input/output (I/O)
attribute	instance variable
Basic	Internet
behavior	interpreter
behavioral design pattern	Java
bytecode	.java extension
bytecode verifier	Java 2 Software Development Kit (J2SDK)
C	java interpreter
C standard library	Java Virtual Machine
C++	javac compiler
central processing unit (CPU)	JIT (just-in-time) compiler
class	KIS (keep it simple)
.class file	legacy systems
class libraries	live-code™ approach
class loader	load phase
client/server computing	logic error
COBOL	machine dependent
collections	machine independent
compile phase	machine language
compiler	memory unit
compile-time error	method
computer	Microsoft
computer program	Microsoft Internet Explorer Web browser
computer programmer	modeling
condition	multiprocessor
CPU (central processing unit)	multitasking
creational design pattern	multithreading
design pattern	Netscape Navigator Web browser
disk	nonfatal run-time error
distributed computing	object
dynamic content	object
edit phase	object-oriented analysis and design (OOAD)
editor	object-oriented design (OOD)
encapsulation	object-oriented programming (OOP)
event-driven programming	open source
execute phase	output device
execution-time error	output unit
fatal runtime error	Pascal
file server	personal computing
Fortran	platforms
freeware	portability
hardware	primary memory
high-level language	problem statement
HotSpot compiler	procedural programming
HTML (Hypertext Markup Language)	programming language

reference	Swing GUI components
requirements document	syntax error
reusable componentry	throughput
runtime error	throw an exception
secondary storage unit	timesharing
shareware	translator programs
software	Unified Modeling Language (UML)
software reuse	verify phase
structural design pattern	video
structured programming	World Wide Web
Sun Microsystems	

SELF-REVIEW EXERCISES

1.1 Fill in the blanks in each of the following statements:

- The company that popularized personal computing was _____.
- The computer that made personal computing legitimate in business and industry was the _____.
- Computers process data under the control of sets of instructions called _____.
- The six key logical units of the computer are the _____, _____, _____, _____, _____ and _____.
- The three classes of languages discussed in the chapter are _____, _____ and _____.
- The programs that translate high-level language programs into machine language are called _____.

1.2 Fill in the blanks in each of the following sentences about the Java environment:

- The _____ command from the Java 2 Software Development Kit executes a Java applet.
- The _____ command from the Java 2 Software Development Kit executes a Java application.
- The _____ command from the Java 2 Software Development Kit compiles a Java program.
- A(n) _____ file is required to invoke a Java applet.
- A Java program file must end with the _____ file extension.
- When a Java program is compiled, the file produced by the compiler ends with the _____ file extension.
- The file produced by the Java compiler contains _____ that are interpreted to execute a Java applet or application.

1.3 Fill in the blanks in each of the following statements:

- The _____ allows computer users to locate and view multimedia-based documents on almost any subject over the Internet.
- Java _____ typically are stored on your computer and are designed to execute independent of a World Wide Web browsers.
- Lists and tables of values are called _____.
- The _____ GUI components are written completely in Java.
- _____ allows an applet or application to perform multiple activities in parallel.
- _____ provide Java programmers with a standard set of data structures for storing and retrieving data and a standard set of algorithms that allow programmers to manipulate the data.

- 1.4** Fill in the blanks in each of the following statements (based on Sections 1.15 and 1.16):
- Over the past decade, the software-engineering industry has made significant progress in the field of _____—proven architectures for constructing flexible and maintainable object-oriented software.
 - Objects have the property of _____.
 - Java programmers concentrate on creating their own user-defined types, called _____.
 - Classes can also have relationships with other classes. These relationships are called _____.
 - The process of analyzing and designing a system from an object-oriented point of view is called _____.

ANSWERS TO SELF-REVIEW EXERCISES

- 1.1** a) Apple. b) IBM Personal Computer. c) programs. d) input unit, output unit, memory unit, arithmetic and logic unit, central processing unit, secondary storage unit. e) machine languages, assembly languages, high-level languages. f) compilers.
- 1.2** a) **appletviewer**. b) **java**. c) **javac**. d) HTML. e) **.java**. f) **.class**. g) byte-codes.
- 1.3** a) World Wide Web. b) applications. c) arrays. d) Swing. e) Multithreading. f) Collections.
- 1.4** a) design patterns. b) information hiding. c) classes. d) associations. e) object-oriented analysis and design (OOAD).

EXERCISES

- 1.5** Categorize each of the following items as either hardware or software:
- CPU
 - Java compiler
 - ALU
 - Java interpreter
 - input unit
 - editor
- 1.6** Why might you want to write a program in a machine-independent language instead of a machine-dependent language? Why might a machine-dependent language be more appropriate for writing certain types of programs?
- 1.7** Fill in the blanks in each of the following statements:
- Which logical unit of the computer receives information from outside the computer for use by the computer? _____.
 - The process of instructing the computer to solve specific problems is called _____.
 - What type of computer language uses English-like abbreviations for machine language instructions? _____.
 - Which logical unit of the computer sends information that has already been processed by the computer to various devices so that the information may be used outside the computer? _____.
 - Which logical unit of the computer retains information? _____.
 - Which logical unit of the computer performs calculations? _____.
 - Which logical unit of the computer makes logical decisions? _____.
 - The level of computer language most convenient to the programmer for writing programs quickly and easily is _____.

- i) The only language that a computer can directly understand is called that computer's _____.
- j) Which logical unit of the computer coordinates the activities of all the other logical units? _____.

1.8 Distinguish between the terms fatal error and nonfatal error. Why might you prefer to experience a fatal error rather than a nonfatal error?

1.9 Fill in the blanks in each of the following statements:

- a) Java _____ are designed to be transported over the Internet and executed in World Wide Web browsers.
- b) _____ programming causes a program to perform a task in response to user interactions with graphical user interface (GUI) components.
- c) Java's graphics capabilities are _____ and, hence portable.
- d) The standard _____ can be used to provide identical user interfaces across all computer platforms.
- e) Languages that cannot perform multiple activities in parallel are called _____ languages or _____ languages.
- f) Aggregations of data such as linked lists, stacks, queues and trees are called _____.

1.10 Fill in the blanks in each of the following statements (based on Sections 1.15 and 1.16):

- a) _____ design patterns describe techniques to instantiate objects (or groups of objects).
- b) The _____ is now the most widely used graphical representation scheme for modeling object-oriented systems.
- c) Java classes contain _____ (which implement class behaviors) and _____ (which implement class data).
- d) _____ design patterns allow designers to organize classes and objects into larger structures.
- e) _____ design patterns assign responsibilities to objects.
- f) In Java, the unit of programming is the _____, from which _____ are eventually instantiated.