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 components 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 **`Error`**s (normally beyond the control of most programs) and **`Exception`**s (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 **`throw`**s 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 **`Error`**s and **`Exception`**s 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 *multiprocessors*. 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 "sequential" 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:

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

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 **`Collection`**s 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 **`Hashtable`**s), **`Random`** and **`BitSet`**. The discussion of **`BitSet`** includes live code for one of the classic applications of **`BitSet`**s, 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:

Bouncing Ball Physics Demo	*Knight's Tour Walker*	*Story Teller*
Craps	*Morse Code*	*Tic-Tac-Toe*

Digital Clock
Flight Simulator
Karaoke
Kinetics Physics Demo
MP3 Player
Multimedia Authoring System
Pinball Machine
Roulette
Tortoise and the Hare
Towers of Hanoi
Video Conferencing
Video Games

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):

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

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
applets/applications
arts and entertainment
audio sites
books
Borland JBuilder IDE
conferences
consultants
contests
CORBA homepage
current information
databases
demos (many with source code)
developer's kit
development tools
discussion groups
documentation
downloadable applets
FAQs (frequently asked ?s)
games
graphics
IBM Developers Java Zone
Intelligence.com
Java Applet Rating Service
Java Developer Connection
Java Developer's Journal
Java Media Framework
Java Report
Java tools
Java Toys
Java Users Group (JUGs)
Java Woman
`java.sun.com`
JavaWorld on-line magazine
learning Java
links to Java sites
lists of resources
lists of what is new and cool
live chat sessions on Java
multimedia collections
NASA multimedia gallery
NetBeans IDE
news
`news:comp.lang.java`
newsgroups
newsletters
Object Management Group
products
projects
publications
puzzles
reference materials
resources
seminars
sites
software
Sun Microsystems
SunWorld on-line magazine
Team Java
The Java Tutorial
trade shows
training (please call us!)
tutorials for learning java
URLs for Java applets
`www.javaworld.com`
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 *statechart 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 3,465-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.