

# Preface

In this book, we lead you on a journey into the fun and exciting world of computer programming. Throughout your journey, we'll provide you with lots of problem-solving practice. After all, good programmers need to be good problem solvers. We'll show you how to implement your problem solutions with Java programs. We provide a plethora of examples, some short and focused on a single concept, some longer and more “real world.” We present the material in a conversational, easy-to-follow manner aimed at making your journey a pleasant one. When you're done with the book, you should be a proficient Java programmer.

Our textbook targets a wide range of readers. Primarily, it targets students in a standard college-level “Introduction to Programming” course or course sequence where no prerequisite programming experience is assumed.

In addition to targeting students with no prerequisite programming experience, our textbook also targets industry practitioners and college-level students who have some programming experience and want to learn Java. This second set of readers can skip the early chapters on general programming concepts and focus on the features of Java that differ from the languages that they already know. In particular, since C++ and Java are so similar, readers with a C++ background should be able to cover the textbook in a single three-credit-hour course. (But let us reiterate for those of you with no programming experience: You should be fine. No prerequisite programming experience is required.)

Finally, our textbook targets high school students and readers outside of academia with no programming experience. This third set of readers should read the entire textbook at a pace determined on a case-by-case basis.

## Textbook Cornerstone #1: Problem Solving

Being able to solve problems is a critical skill that all programmers must possess. We teach programmatic problem solving by emphasizing two of its key elements—algorithm development and program design.

### Emphasis on Algorithm Development

In Chapter 2, we immerse readers into algorithm development by using pseudocode for the algorithm examples instead of Java. In using pseudocode, students are able to work through non-trivial problems on their own without getting bogged down in Java syntax—no need to worry about class headings, semicolons, braces, and so on.<sup>1</sup> Working through non-trivial problems enables students to gain an early appreciation for creativity, logic, and organization. Without that appreciation, Java students tend to learn Java syntax with a rote-memory attitude. But with that appreciation, students tend to learn Java syntax more quickly and effectively because they have a motivational basis for learning it. In addition, they are able to handle non-

<sup>1</sup> Inevitably, we use a particular style for our pseudocode, but we repeatedly emphasize that other pseudocode styles are fine as long as they convey the intended meaning. Our pseudocode style is a combination of free-form description for high-level tasks and more specific commands for low-level tasks. For the specific commands, we use natural English words rather than cryptic symbols. We've chosen a pseudocode style that is intuitive, to welcome new programmers, and structured, to accommodate program logic.

trivial Java homework assignments fairly early because they have prior experience with similarly non-trivial pseudocode homework assignments.

In Chapter 3 and in later chapters, we rely primarily on Java for algorithm-development examples. But for the more involved problems, we sometimes use high-level pseudocode to describe first-cut proposed solutions. Using pseudocode enables readers to bypass syntax details and focus on the algorithm portion of the solution.

## Emphasis on Program Design

Problem solving is more than just developing an algorithm. It also involves figuring out the best implementation for the algorithm. That's program design. Program design is extremely important and that's why we spend so much time on it. We don't just present a solution. We explain the thought processes that arise when coming up with a solution. For example, we explain how to choose between different loop types, how to split up a method into multiple methods, how to decide on appropriate classes, how to choose between instance and class members, and how to determine class relationships using inheritance and composition. We challenge students to find the most elegant implementations for a particular task.

We devote a whole chapter to program design—Chapter 8, Software Engineering. In that chapter, we provide in-depth looks at coding-style conventions, modularization, and encapsulation. Also in the chapter, we describe alternative design strategies—top-down, bottom-up, case-based, and iterative enhancement.

## Problem-Solving Sections

We often address problem solving (algorithm development and program design) in the natural flow of explaining concepts. But we also cover problem solving in sections that are wholly devoted to it. In each problem-solving section, we present a situation that contains an unresolved problem. In coming up with a solution for the problem, we try to mimic the real-world problem-solving experience by using an iterative design strategy. We present a first-cut solution, analyze the solution, and then discuss possible improvements to it. We use a conversational trial-and-error format (e.g., “What type of layout manager should we use? We first tried the `GridLayout` manager. That works OK, but not great. Let's now try the `BorderLayout` manager.”). This casual tone sets the student at ease by conveying the message that it is normal, and in fact expected, that a programmer will need to work through a problem multiple times before finding the best solution.

## Additional Problem-Solving Mechanisms

We include problem-solving examples and problem-solving advice throughout the text (not just in Chapter 2, Chapter 8, and the problem-solving sections). As a point of emphasis, we insert a problem-solving box, with an icon and a succinct tip, next to the text that contains the problem-solving example and/or advice.

We are strong believers in learning by example. As such, our textbook contains a multitude of complete program examples. Readers are encouraged to use our programs as recipes for solving similar programs on their own.

## Textbook Cornerstone #2: Fundamentals First

### Postpone Concepts That Require Complex Syntax

We feel that many introductory programming textbooks jump too quickly into concepts that require complex syntax. In using complex syntax early, students get in the habit of entering code without fully understanding it or, worse yet, copying and pasting from example code without fully understanding the example code. That can lead to less-than-ideal programs and students who are limited in their ability to solve a wide variety of

problems. Thus, we prefer to postpone concepts that require complex syntax. We prefer to introduce such concepts later on when students are better able to fully understand them.

As a prime example of that philosophy, we cover the simpler forms of GUI programming early (in an optional graphics track), but we cover the more complicated forms of GUI programming late. Specifically, we postpone event-driven GUI programming until the end of the book. This is different from some other Java textbooks, which favor early full immersion into event-driven GUI programming. We feel that strategy is a mistake because proper event-driven GUI programming requires a great deal of programming maturity. By covering it at the end of the book, our readers are better able to fully understand it.

## Tracing Examples

To write code effectively, it's imperative to understand code thoroughly. We've found that step-by-step tracing of program code is an effective way to ensure thorough understanding. Thus, in the earlier parts of the textbook, when we introduce a new programming structure, we often illustrate it with a meticulous trace. The detailed tracing technique we use illustrates the thought process programmers employ while debugging. It's a printed alternative to the sequence of screen displays generated by debuggers in IDE software.

## Input and Output

In the optional GUI-track sections and in the GUI chapters at the end of the book, we use GUI commands for input and output (I/O). But because of our emphasis on fundamentals, we use console commands for I/O for the rest of the book.<sup>2</sup> For console input, we use the `Scanner` class. For console output, we use the standard `System.out.print`, `System.out.println`, and `System.out.printf` methods.

## Textbook Cornerstone #3: Real World

More often than not, today's classroom students and industry practitioners prefer to learn with a hands-on, real-world approach. To meet this need, our textbook includes:

- compiler tools
- complete program examples
- practical guidance in program design
- coding-style guidelines based on industry standards
- UML notation for class relationship diagrams
- practical homework-project assignments

## Compiler Tools

We do not tie the textbook to any particular compiler tool—you are free to use any compiler tool(s) that you like. If you do not have a preferred compiler in mind, then you might want to try out one or more of these:

- Java2 SDK toolkit, by Sun
- TextPad, by Helios

---

<sup>2</sup> We cover GUI I/O early on with the `JOptionPane` class. That opens up an optional door for GUI fans. If readers are so inclined, they can use `JOptionPane` to implement all of our programs with GUI I/O rather than console I/O. To do so, they replace all console I/O method calls with `JOptionPane` method calls.

- Eclipse, by the Eclipse Foundation
- Netbeans, backed by Sun
- BlueJ, by the University of Kent and Deaken University

To obtain the above compilers, visit our textbook Web site at <http://www.mhhe.com/dean>, find the appropriate compiler link(s), and download away for free.

## Complete Program Examples

In addition to providing code fragments to illustrate specific concepts, our textbook contains lots of complete program examples. With complete programs, students are able to (1) see how the analyzed code ties in with the rest of a program, and (2) test the code by running it.

## Coding-Style Conventions

We include coding-style tips throughout the textbook. The coding-style tips are based on Sun's coding conventions (<http://java.sun.com/docs/codeconv/>) and industry practice. In Appendix 5, we provide a complete reference for the book's coding-style conventions and an associated example program that illustrates the conventions.

## UML Notation

The Universal Modeling Language (UML) has become a standard for describing the entities in large software projects. Rather than overwhelm beginning programmers with syntax for the entire UML (which is quite extensive), we present a subset of the UML. Throughout the textbook, we incorporate UML notation to pictorially represent classes and class relationships. For those interested in more details, we provide additional UML notation in Appendix 7.

## Homework Problems

We provide homework problems that are illustrative, practical, and clearly worded. The problems range from easy to challenging. They are grouped into three categories—review questions, exercises, and projects. We include review questions and exercises at the end of each chapter, and we provide projects on our textbook's Web site.

The review questions tend to have short answers and the answers are in the textbook. The review questions use these formats: short-answer, multiple-choice, true/false, fill-in-the-blanks, tracing, debugging, write a code fragment. Each review question is based on a relatively small part of the chapter.

The exercises tend to have short to moderate-length answers, and the answers are not in the textbook. The exercises use these formats: short-answer, tracing, debugging, write a code fragment. Exercises are keyed to the highest prerequisite section number in the chapter, but they sometimes integrate concepts from several parts of the chapter.

The projects consist of problem descriptions whose solutions are complete programs. Project solutions are not in the textbook. Projects require students to employ creativity and problem-solving skills and apply what they've learned in the chapter. These projects often include optional parts, which provide challenges for the more talented students. Projects are keyed to the highest prerequisite section number in the chapter, but they often integrate concepts from several preceding parts of the chapter.

An important special feature of this book is the way it specifies project problems. "Sample sessions" show the precise output generated for a particular set of input values. These sample sessions include inputs that represent typical situations and sometimes also extreme or boundary situations.

## Academic-Area Projects

To enhance the appeal of projects and to show how the current chapter's programming techniques might apply to different areas of interest, we take project content from several academic areas:

- Computer Science and Numerical Methods
- Business and Accounting
- Social Sciences and Statistics
- Math and Physics
- Engineering and Architecture
- Biology and Ecology

The academic-area projects do not require prerequisite knowledge in a particular area. Thus, instructors are free to assign any of the projects to any of their students. To provide a general reader with enough specialized knowledge to work a problem in a particular academic area, we sometimes expand the problem statement to explain a few special concepts in that academic area.

Most of the academic-area projects do not require students to have completed projects from earlier chapters; that is, the projects do not build on each other. Thus, instructors are free to assign projects without worrying about prerequisite projects. In some cases, a project repeats a previous chapter's project with a different approach. The teacher may elect to take advantage of this repetition to dramatize the availability of alternatives, but this is not necessary.

Project assignments can be tailored to fit readers' needs. For example:

- For readers outside of academia—  
Readers can choose projects that match their interests.
- When a course has students from one academic area—  
Instructors can assign projects from the relevant academic area.
- When a course has students with diverse backgrounds—  
Instructors can ask students to choose projects from their own academic areas, or  
Instructors can ignore the academic-area delineations and simply assign projects that are most appealing.

To help you decide which projects to work on, we've included a "Project Summary" section after the preface. It lists all the projects by chapter, and for each project, it specifies:

- The associated section within the chapter
- The academic area
- The length and difficulty
- A brief description

After using the "Project Summary" section to get an idea of which projects you might like to work on, see the textbook's Web site for the full project descriptions.

## Organization

In writing this book, we lead readers through three important programming methodologies: structured programming, object-oriented programming (OOP), and event-driven programming. For our structured programming coverage, we introduce basic concepts such as variables and operators, `if` statements, and loops. For our OOP coverage, we start by showing readers how to call pre-built methods from Sun's Java Applica-

tion Programming Interface (API) library. We then introduce basic OOP concepts such as classes, objects, instance variables, and instance methods. Next, we move on to more advanced OOP concepts—class variables, arrays, and inheritance. Chapters on exception handling and files provide a transition into event-driven graphical user interface (GUI) programming. We cover event-driven GUI programming in earnest in the final two chapters.

The content and sequence we promote enable students to develop their skills from a solid foundation of programming fundamentals. To foster this fundamentals-first approach, our book starts with a minimum set of concepts and details. It then gradually broadens concepts and adds detail later. We avoid overloading early chapters by deferring certain less-important details to later chapters.

## GUI Track

Many programmers find GUI programming to be fun. As such, GUI programming can be a great motivational tool for keeping readers interested and engaged. That’s why we include graphics sections throughout the book, starting in Chapter 1. We call those sections our “GUI track.” For readers who do not have time for the GUI track, no problem. Any or all of the GUI track sections may be skipped as they cover material that is independent of later material.

## Chapter 1

In Chapter 1, we first explain basic computer terms—what are the hardware components, what is source code, what is object code, and so on. We then narrow our focus and describe the programming language we’ll be using for the remainder of the book—Java. Finally, we give students a quick view of the classic bare-bones “Hello World” program. We explain how to create and run the program using minimalist software—Microsoft’s Notepad text editor and Sun’s command-line Software Development Kit (SDK) tools.

## Chapter 2

In Chapter 2, we present problem-solving techniques with an emphasis on algorithmic design. In implementing algorithm solutions, we use generic tools—flowcharts and pseudocode—with pseudocode being given the greatest weight. As part of our algorithm-design explanation, we describe structured programming techniques. In order to give students an appreciation for semantic details, we show how to trace algorithms.

## Chapters 3–5

We present structured programming techniques using Java in Chapters 3–5. Chapter 3 describes sequential programming basics—variables, input/output, assignment statements, and simple method calls. Chapter 4 describes non-sequential program flow—`if` statements, `switch` statements, and loops. In Chapter 5 we explain methods in more detail and show readers how to use pre-built methods in the Java API library. In all three chapters, we teach algorithm design by solving problems and writing programs with the newly introduced Java syntax.

## Chapters 6–8

Chapter 6 introduces the basic elements of OOP in Java. This includes implementing classes and implementing methods and variables within those classes. We use UML class diagrams and object-oriented tracing techniques to illustrate these concepts.

Chapter 7 provides additional OOP details. It explains how reference variables are assigned, tested for equality, and passed as arguments to a method. It covers overloaded methods and constructors.

While the art of program design and the science of computerized problem-solving are developed throughout the textbook, in Chapter 8, we focus on these aspects in the context of OOP. This chapter begins with an organized treatment of programming style. It includes recommendations on how to use methods to further the goal of encapsulation. It describes the major programming paradigms—top-down design, bottom-up design, using pre-written software for low-level modules, and prototyping.

## Chapter 9

Some Java textbooks teach how to implement class members before they teach how to implement instance members. With that approach, students learn to write class members inappropriately, and that practice is hard to break later on when instance members are finally covered. Proper programming practice dictates that programmers (beginning programmers certainly included) should implement instance members more often than class members. Thus, we teach how to implement instance members early on, and we postpone how to implement class members until Chapter 9.

## Chapter 10

In Chapter 10, we describe different ways to store related data. We present array basics and several important array applications—searching, sorting, and histogram construction. We present more advanced array concepts using two-dimensional arrays and arrays of objects. Finally, we look at a more powerful form of an array—an `ArrayList`.

## Chapter 11

Early on, students need to be immersed in problem-solving activities. Covering too much syntax detail early can detract from that objective. Thus, we initially gloss over some less-important syntax details and come back to those details later on in Chapter 11. Chapter 11 provides more details on items such as these:

- the `byte` and `short` primitive types
- the Unicode character set
- type promotions
- postfix versus prefix modes for the increment and decrement operators
- the conditional operator
- short-circuit evaluation

## Chapters 12–13

We describe class relationships in Chapters 12 and 13. We spend two full chapters on class relationships because the subject matter is so important. We take the time to explain class relationship details in depth and provide numerous examples. In Chapter 12, we discuss aggregation, composition, and inheritance. In Chapter 13, we discuss more advanced inheritance-related details such as the `Object` class, polymorphism, abstract classes, and interfaces.

## Chapters 14–15

We cover exception handling in Chapter 14 and files in Chapter 15. We cover exception handling prior to files because file-handling code utilizes exception handling; for example, opening a file requires that you check for an exception.

## Chapters 16–17

We cover event-driven GUI programming at the end of the book in Chapters 16 and 17. By learning event-driven GUI programming late, students are better able to grasp its inherent complexities.

## Appendices

Most of the appendices cover reference material, such as the ASCII character set and the operator precedence table. But the last two appendices cover advanced Java material—recursion and multithreading.

## Subject-Matter Dependencies and Sequence-Changing Opportunities

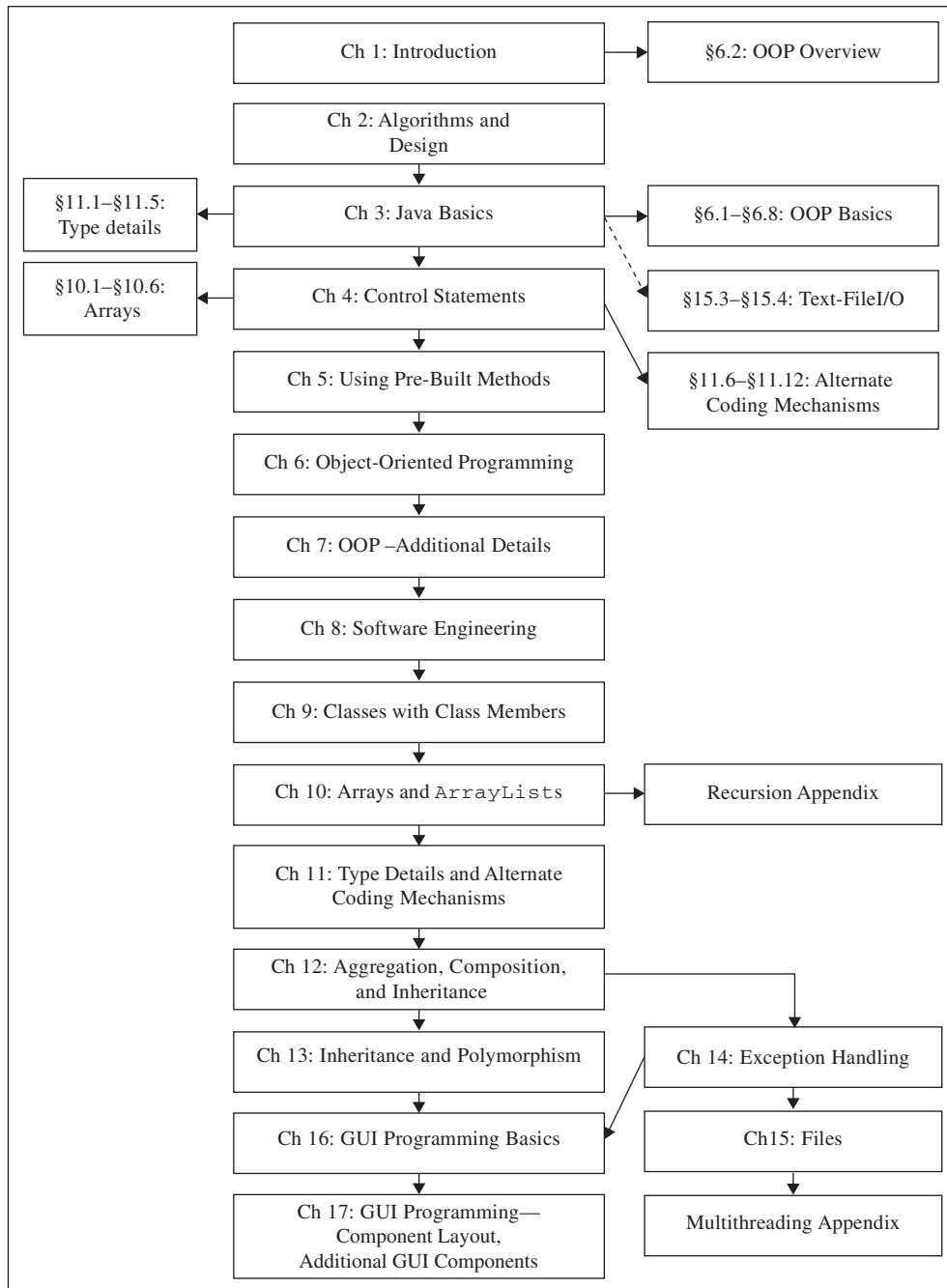
We’ve positioned the textbook’s material in a natural order for someone who wants fundamentals first and also wants an early introduction to OOP. We feel that our order is the most efficient and effective order for learning how to become a proficient OOP programmer. Nonetheless, we realize that different readers have different content-ordering preferences. To accommodate those different preferences, we’ve provided some built-in flexibility. Figure 0.1 illustrates that flexibility by showing chapter dependencies and, more importantly, chapter non-dependencies. For example, the arrow between Chapter 3 and Chapter 4 means that Chapter 3 must be read prior to Chapter 4. And the lack of an arrow between Chapters 1 and 2 means that Chapter 1 may be skipped.

Here are some sequence-changing opportunities revealed by Figure 0.1:

- Readers can skip Chapter 1 (Introduction to Computers and Programming).
- For an earlier introduction to OOP, readers can read the OOP overview section in Chapter 6 after reading Chapter 1. And they can learn OOP syntax and semantics in Chapter 6 after finishing Java basics in Chapter 3.
- For additional looping practice, readers can learn about arrays in Chapter 10 after finishing loops in Chapter 4.
- Readers can skip Chapter 15 (Files).

Note Figure 0.1’s dashed arrow that connects Chapter 3 to Chapter 15. We use a dashed arrow to indicate that the connection is partial. Some readers may wish to use files early on for input and output (I/O). Those readers should read Chapter 3 for Java basics and then immediately jump to Chapter 15, Sections 15.3 and 15.4 for text-file I/O. With a little work, they’ll then be able to use files for all their I/O needs throughout the rest of the book. We say “with a little work” because the text-file I/O sections contain some code that won’t be fully understood by someone coming directly from Chapter 3. To use the text-file I/O code, they’ll need to treat it as a template. In other words, they’ll use the code even though they probably won’t understand some of it.

To support content-ordering flexibility, the book contains “hyperlinks.” A hyperlink is an optional jump forward from one place in the book to another place. The jumps are legal in terms of prerequisite knowledge, meaning that the jumped-over (skipped) material is unnecessary for an understanding of the later material. We supply hyperlinks for each of the non-sequential arrows in Figure 0.1. For example, we supply hyperlinks that go from Chapter 1 to Chapter 6 and from Chapter 3 to Chapter 11. For each hyperlink tail end (in the earlier chapter), we tell the reader where they may optionally jump to. For each hyperlink target end (in the later chapter), we provide an icon at the side of the target text that helps readers find the place where they are to begin reading.



**Figure 0.1** Chapter dependencies

## Pedagogy

---

### Icons



Program elegance.

Indicates that the associated text deals with a program's coding style, readability, maintainability, robustness, and scalability. Those qualities comprise a program's elegance.



Problem solving.

Indicates that the associated text deals with problem-solving issues. Comments associated with this icon attempt to generalize highlighted material in the adjacent text.



Common errors.

Indicates that the associated text deals with common errors.



Hyperlink target.

Indicates the target end of a hyperlink.



Program efficiency.

Indicates that the associated text refers to program-efficiency issues.

## Student Resources

---

At the textbook Web site, <http://www.mhhe.com/dean>, students (and also teachers) can view and download these resources:

- Links to compiler software—for Sun's Java2 SDK toolkit, Helios's TextPad, Eclipse, NetBeans, and BlueJ
- TextPad tutorial
- Eclipse tutorials
- Textbook errata
- Student-version PowerPoint lecture slides without hidden notes
  - The student-version slides are identical to the teacher-version slides except that the hidden notes, hidden slides, and quizzes are omitted.
  - Omitting the hidden notes forces the students to go to lecture to hear the sage on the stage fill in the blanks.
- Project assignments
- All textbook example programs and associated resource files

## Instructor Resources

---

At the textbook Web site, <http://www.mhhe.com/dean>, instructors can view and download these resources:

- Teacher-version PowerPoint lecture slides with hidden notes
  - Hidden notes provide comments that supplement the displayed text in the lecture slides.
  - For example, if the displayed text asks a question, the hidden notes provide the answer.
- Exercise solutions
- Project solutions
- Test bank materials

## Acknowledgments

Anyone who has written a textbook can attest to what a large and well-orchestrated team effort it requires. Such a book can never be the work of only one person or even a few. We are deeply indebted to the team at McGraw-Hill Higher Education who have shown continued faith in our writing and invested generously in it.

It was a pleasure to work with Alan Apt during the book's two-year review period. He provided excellent guidance on several large design issues. We are grateful for the tireless efforts of Rebecca Olson. Rebecca did a tremendous job organizing and analyzing the book's many reviews. Helping us through the various stages of production were Project Manager Kay Brimeyer and Designer Laurie Janssen. We would also like to thank the rest of the editorial and marketing team, who helped in the final stages: Raghu Srinivasan, Global Publisher; Kristine Tibbetts, Director of Development; Heidi Newsom, Editorial Assistant; and Michael Weitz, Executive Marketing Manager.

All the professionals we have encountered throughout the McGraw-Hill organization have been wonderful to work with, and we sincerely appreciate their efforts.

We would like to acknowledge with appreciation the numerous and valuable comments, suggestions, and constructive criticisms and praise from the many instructors who have reviewed the book. In particular,

William Allen, *Florida Institute of Technology*  
 Robert Burton, *Brigham Young University*  
 Priscilla Dodds, *Georgia Perimeter College*  
 Jeanne M. Douglas, *University of Vermont*  
 Dr. H.E. Dunsmore, *Purdue University*  
 Deena Engel, *New York University*  
 Michael N. Huhns, *University of South Carolina*  
 Ibrahim Imam, *University of Louisville*  
 Andree Jacobson, *University of New Mexico*  
 Lawrence King, *University of Texas, Dallas*  
 Mark Llewellyn, *University of Central Florida*  
 Blayne E. Mayfield, *Oklahoma State University*  
 Mary McCollam, *Queen's University*  
 Hugh McGuire, *Grand Valley State University*  
 Jeanne Milostan, *Vanderbilt University*  
 Shyamal Mitra, *University of Texas, Austin*  
 Benjamin B. Nystuen, *University of Colorado, Colorado Springs*  
 Richard E. Pattis, *Carnegie Mellon University*  
 Tom Stokke, *University of North Dakota*  
 Ronald Taylor, *Wright State University*  
 Timothy A. Terrill, *University at Buffalo, The State University of New York*  
 Ping Wu, *Dell Inc*

We would also like to thank colleagues Wen Hsin, Kevin Burger, John Cigas, Bob Cotter, Alice Capson, and Mark Adams for helping with informal quick surveys and Barbara Kushan, Ed Tankins,

Mark Reith, and Benny Phillips for class testing. And a special debt of gratitude goes to colleague and grammarian nonpareil Jeff Glauner, who helped with subtle English syntax nuances.

Finally, thanks to the students. To the ones who encouraged the initial writing of the book, and to the ones who provided feedback and searched diligently for mistakes in order to earn bonus points on the homework. In particular, thank you Aris Czamanske, Malallai Zalmal, Paul John, Joby John, Matt Thebo, Josh McKinzie, Carol Liberty, Adeeb Jarrah, and Virginia Maikweki.

Sincerely,  
John and Ray