Java provides programmers with an immediately attractive forum. With Java and the World Wide Web, a programmer’s work can have quick global distribution with appealing graphics and a marvelous capability for feedback, modification, software reuse, and growth. Certainly, Java’s other qualities have also contributed to its rapid adoption: the capability to implement clean, object-oriented designs; the familiarity of Java’s syntactic constructs; the good collection of ready-to-use features in the Java Class Libraries. But it’s the winsome ways of the World Wide Web that have pulled Java to the front for both experienced programmers and the newcomer entering a first-year computer science course.

With that said, it’s satisfying to see that the core of the first-year courses remains solid even as courses start using Java. The proven methods of representing and using data structures remain vital to beginning programmers in a data structures course, even as the curriculum is shifting to accommodate an object-oriented approach.

This book is written for just such an evolving data structures course, with the students’ programming carried out in Java. The text’s emphasis is on the specification, design, implementation, and use of the basic data types that are normally covered in a second-semester, object-oriented course. There is wide coverage of important programming techniques in recursion, searching, and sorting. The text also provides coverage of big-$O$ time analysis of algorithms, an optional appendix on writing an interactive applet to test an implementation of a data structure, using Javadoc to specify precondition/postcondition contracts, and a new introduction to the increasingly important topic of concurrency.

The text assumes that the student has already had an introductory computer science and programming class, but we do include coverage of those topics (such as the Java `Object` type and a precise description of parameter passing) that are not always covered completely in a first course. The rest of this preface discusses ways that this material can be covered, starting with a brief review of how con-
temporary topics from earlier editions are intermixed with new material from the new Java 2 Standard Edition 7.0.

The Fourth Edition’s Retained Contemporary Topics and Java 2 SE 7.0

This edition retains contemporary topics from earlier editions that define a coherent approach to the implementation and use of data structures in Java. These topics, listed below, have been updated to align with the latest software engineering practices suggested by the designers and implementors of Java 2 Standard Edition 7.0. The most important of these topics are listed here:

- **Better input/output support** (Chapter 1 and Appendix B) continues in this edition, including formatted output (with the `System.out.printf` method) and the input of primitive values (with the `java.util.Scanner` class). Our earlier input and output classes (`EasyReader` and `FormatWriter`) are still available online at www.cs.colorado.edu/~main/edu/colorado/io.

- **Earlier integration of the Java Class Libraries** begins in Chapter 2 of this edition. Although the emphasis of this textbook remains on the underlying design and implementation of classic data structures, we’ve found it increasingly important to give students significant exposure to the elements of class libraries. We begin this process in Section 2.5 with the easy-to-use `BigInteger` class. Chapters 3 to 5 give more examples that use the Java Class Libraries, including `Arrays`, `HashSet`, `ArrayList`, `LinkedList`, `ListIterator`, and others.

- **A modified approach to generic collection classes** occurs in Chapter 5 of this edition. Starting with J2SE 5.0, generic methods and generic classes allowed libraries to depend on an unspecified underlying data type (that we’ll call `E`). Originally, collection classes were often implemented with arrays in which the component type was an unspecified generic type (i.e., a private member variable of type `E[]`).

  Over the years, it was recognized that the `E[]` practice had a potential pitfall: We could use a typecast to pretend that we had that kind of array, but at run time, the program really had an array of Java objects (i.e., a private member variable of type `Object[]`, which by design contains only `E` objects). Within the generic class itself, this type mismatch did not cause a problem (because generic types are not checked at run time). But if the generic class provides the array to an outside application, then the possibility of a runtime type error existed.

  The solution, eloquently written about by Neal Gafter, is for the generic class to use an array of Java objects (rather than an array of type `E[]`) along with specific programming practices (described in Chapter 5) that ensure that unchecked typecasts cannot lead to runtime type errors.
Starting in Chapter 5 and continuing throughout the rest of the textbook, we now use this new approach for all of our generic classes.

- **Variable arity methods** (Chapter 3) allow a method to have a variable number of arguments. Generic classes have special considerations for variable arity methods that are newly discussed in Chapter 5 of this edition.

- **Enhanced for-loops** (Chapters 3 and 5) were introduced in the previous edition of this textbook. They allow easy iteration through the elements of an array, through a collection of elements that has implemented the `Iterable` interface, or through the elements of an enum type.

- **Autoboxing and auto-unboxing** of primitive values (Chapter 5) continue to be used in this edition to allow the storage and retrieval of primitive data values (such as an `int`) in Java’s generic collections (such as the `Vector`).

- **Java’s priority queue** (Chapter 7) has been added as a new topic in the fourth edition.

- **The binary search and sort methods from the Java Class Libraries** have a new presentation in Chapters 11 (searching) and 12 (sorting).

- **The important new topic of concurrency** is introduced through a concurrent sort in Chapter 12. The approach uses the `RecursiveAction` and `ForkJoinPool` classes from Java2SE 7, which provides an extremely clean introduction with very few distractions for the first-year student.

- **Covariant return types** allow the return type of an overridden inherited method to be any descendant class of the original return type. We use this idea for clone methods throughout the text, thereby avoiding a typecast with each use of a clone method and increasing type security. The larger use of the technique is postponed until the chapter on inheritance (Chapter 13).

- **Enum types** (Chapter 13) provide a convenient way to define a new data type whose objects may take on a small set of discrete values.

Beyond these new, coherent language features, the fourth edition of this text presents the same foundational data structures material as the earlier editions. In particular, the data structures curriculum emphasizes the ability to specify, design, and analyze data structures independently of any particular language, as well as the ability to implement new data structures and use existing data structures in any modern language. You’ll see this approach in the first four chapters as students review Java and then go on to study and implement the most fundamental data structures (bags and sequence classes) using both arrays and linked-list techniques that are easily applied to any high-level language.
Chapter 5 is a bit of a departure, bringing forward several Java-specific techniques that are particularly relevant to data structures: how to build generic collection classes based on Java’s new generic types, using Java interfaces and the API classes, and building and using Java iterators. Although much of the subsequent material can be taught without these Java features, this is a natural time for students to learn them.

Later chapters focus on particular data structures (stacks, queues, trees, hash tables, graphs) or programming techniques (recursion, sorting techniques, inheritance). The introduction of recursion is delayed until just before the study of trees. This allows us to introduce recursion with some simple but meaningful examples that do more than just tail recursion. This avoids the mistaken first impression that some students get of recursion being some kind of magic loop. After the first simple examples (including one new example for this edition), students see significant examples involving two recursive calls in common situations with binary trees.

Trees are particularly emphasized in this text, with Chapter 10 taking the students through two meaningful examples of data structures that use balanced trees (the heap project and the B-tree project). Additional projects for trees and other areas have also been added online at http://cssupport.pearsoncmg.com.

Other new Java features have been carried over from the third edition of the text as needed: the use of assertions to check preconditions, a more complete coverage of inheritance, a new example of an abstract class to build classes that play two-player strategy games such as Othello or Connect Four, an introduction to new classes of the Java API (now in the generic form of ArrayList, Vector, HashMap, and Hashtable), and new features of Javadoc.

All these new bells and whistles of Java are convenient, and students need to be up-to-date—but it’s the approaches to designing, specifying, documenting, implementing, using, and analyzing the data structures that will have the most enduring effect on your students.

The Five Steps for Each Data Type

The book’s core consists of the well-known data types: sets, bags (or multisets), sequential lists, stacks, queues, tables, and graphs. There are also additional supplemental data types, such as a priority queue. Some of the types are approached in multiple ways, such as the bag class that is first implemented by storing the elements in an array and is later reimplemented using a binary search tree. Each of the data types is introduced following a pattern of five steps.

Step 1: Understand the Data Type Abstractly. At this level, a student gains an understanding of the data type and its operations through concepts and pictures. For example, a student can visualize a stack and its operations of pushing and popping elements. Simple applications are understood and can be carried out by hand, such as using a stack to reverse the order of letters in a word.
Step 2: Write a Specification of the Data Type as a Java Class. In this step, the student sees and learns how to write a specification for a Java class that can implement the data type. The specification, written using the Javadoc tool, includes headings for the constructors, public methods, and sometimes other public features (such as restrictions tied to memory limitations). The heading of each method is presented along with a precondition/postcondition contract that completely specifies the behavior of the method. At this level, it’s important for the students to realize that the specification is not tied to any particular choice of implementation techniques. In fact, this same specification may be used several times for several different implementations of the same data type.

Step 3: Use the Data Type. With the specification in place, students can write small applications or applets to show the data type in use. These applications are based solely on the data type’s specification because we still have not tied down the implementation.

Step 4: Select Appropriate Data Structures and Proceed to Design and Implement the Data Type. With a good abstract understanding of the data type, we can select an appropriate data structure, such as an array, a linked list of nodes, or a binary tree of nodes. For many of our data types, a first design and implementation will select a simple approach, such as an array. Later, we will redesign and reimplement the same data type with a more complicated underlying structure.

Because we are using Java classes, an implementation of a data type will have the selected data structures (arrays, references to other objects, etc.) as private instance variables of the class. In my own teaching, I stress the necessity for a clear understanding of the rules that relate the private instance variables to the abstract notion of the data type. I require each student to write these rules in clear English sentences that are called the invariant of the abstract data type. Once the invariant is written, students can proceed to implementing various methods. The invariant helps in writing correct methods because of two facts: (a) Each method (except the constructors) knows that the invariant is true when the method begins its work; and (b) each method is responsible for ensuring that the invariant is again true when the method finishes.

Step 5: Analyze the Implementation. Each implementation can be analyzed for correctness, flexibility, and time analysis of the operations (using big-$O$ notation). Students have a particularly strong opportunity for these analyses when the same data type has been implemented in several different ways.

Where Will the Students Be at the End of the Course?

At the end of our course, students understand the data types inside out. They know how to use the data types and how to implement them in several ways. They know the practical effects of the different implementation choices. The
students can reason about efficiency with a big-\(O\) analysis and can argue for the correctness of their implementations by referring to the invariant of the ADT.

One of the lasting effects of the course is the specification, design, and implementation experience. The improved ability to reason about programs is also important. But perhaps most important of all is the exposure to classes that are easily used in many situations. The students no longer have to write everything from scratch. We tell our students that someday they will be thinking about a problem, and they will suddenly realize that a large chunk of the work can be done with a bag, a stack, a queue, or some such. And this large chunk of work is work that they won’t have to do. Instead, they will pull out the bag or stack or queue that they wrote this semester—using it with no modifications. Or, more likely, they will use the familiar data type from a library of standard data types, such as the proposed \textit{Java Class Libraries}. In fact, the behavior of some data types in this text is a cut-down version of the JCL, so when students take the step to the real JCL, they will be on familiar ground—from the standpoint of how to use the class and also having a knowledge of the considerations that went into building the class.

\section*{Other Foundational Topics}

Throughout the course, we also lay a foundation for other aspects of “real programming,” with coverage of the following topics beyond the basic data structures material.

\textbf{Object-Oriented Programming.} The foundations of object-oriented programming are laid by giving students a strong understanding of Java classes. The important aspects of classes are covered early: the notion of a method, the separation into private and public members, the purpose of constructors, and a small exposure to cloning and testing for equality. This is primarily covered in Chapter 2, some of which can be skipped by students with a good exposure to Java classes in the CS1 course.

Further aspects of classes are introduced when the classes first use dynamic arrays (Chapter 3). At this point, the need for a more sophisticated \texttt{clone} method is explained. Teaching this OOP method with the first use of dynamic memory has the effect of giving the students a concrete picture of how an instance variable is used as a reference to a dynamic object such as an array.

Conceptually, the largest innovation of OOP is the software reuse that occurs via inheritance. There are certainly opportunities for introducing inheritance right from the start of a data structures course (such as implementing a set class as a descendant of a bag class). However, an early introduction may also result in students juggling too many new concepts at once, resulting in a weaker understanding of the fundamental data structures. Therefore, in my own course, I introduce inheritance at the end as a vision of things to come. But the introduction to inheritance (Sections 13.1 and 13.2) could be covered as soon as classes are understood. With this in mind, some instructors may wish to cover Chapter 13
earlier, just before stacks and queues, so that stacks and queues can be derived from another class.

Another alternative is to identify students who already know the basics of classes. These students can carry out an inheritance project (such as the ecosystem of Section 13.3), while the rest of the students first learn about classes.

**Java Objects.** The Java Object type lies at the base of all the other Java types—or at least almost all the other types. The eight primitive types are not Java objects, and for many students, the CS1 work has been primarily with the eight primitive types. Because of this, the first few data structures are collections of primitive values, such as a bag of integers or a sequence of double numbers.

**Iterators.** Iterators are an important part of the Java Class Libraries, allowing a programmer to easily step through the elements in a collection class. The Iterable interface is introduced in Chapter 5. Throughout the rest of the text, iterators are not directly used, although they provide a good opportunity for programming projects, such as using a stack to implement an iterator for a binary search tree (Chapter 9).

**Recursion.** First-semester courses often introduce students to recursion. But many of the first-semester examples are tail recursion, where the final act of the method is the recursive call. This may have given students a misleading impression that recursion is nothing more than a loop. Because of this, I prefer to avoid early use of tail recursion in a second-semester course.

So, in our second-semester course, we emphasize recursive solutions that use more than tail recursion. The recursion chapter provides four examples along these lines. Two of the examples—generating random fractals and traversing a maze—are big hits with the students. The fractal example runs as a graphical applet, and although the maze example is text based, an adventurous student can convert it to a graphical applet. These recursion examples (Chapter 8) appear just before trees (Chapter 9) since it is within recursive tree algorithms that recursion becomes vital. However, instructors who desire more emphasis on recursion can move that topic forward, even before Chapter 2.

In a course that has time for advanced tree projects (Chapter 10), we analyze the recursive tree algorithms, explaining the importance of keeping the trees balanced—both to improve worst-case performance and to avoid potential execution stack overflow.

**Searching and Sorting.** Chapters 11 and 12 provide fundamental coverage of searching and sorting algorithms. The searching chapter reviews binary search of an ordered array, which many students will have seen before. Hash tables are also introduced in the search chapter by implementing a version of the JCL hash table and also a second hash table that uses chaining instead of open addressing. The sorting chapter reviews simple quadratic sorting methods, but the majority of the chapter focuses on faster algorithms: the recursive merge sort (with worst-case time of $O(n \log n)$), Tony Hoare’s recursive quicksort (with average-time $O(n \log n)$), and the tree-based heapsort (with worst-case time of $O(n \log n)$).
Advanced Projects, Including Concurrency

The text offers good opportunities for optional projects that can be undertaken by a more advanced class or by students with a stronger background in a large class. Particular advanced projects include the following:

- Interactive applet-based test programs for any of the data structures (outlined in Appendix I).
- Implementing an iterator for the sequence class (see Chapter 5 Programming Projects).
- Writing a deep clone method for a collection class (see Chapter 5 Programming Projects).
- Writing an applet version of an application program (such as the maze traversal in Section 8.2 or the ecosystem in Section 13.3).
- Using a stack to build an iterator for the binary search tree (see Chapter 9 Programming Projects).
- A priority queue implemented as an array of ordinary queues (Section 7.4) or implemented using a heap (Section 10.1).
- A set class implemented with B-trees (Section 10.2). I have made a particular effort on this project to provide sufficient information for students to implement the class without need of another text. Advanced students have successfully completed this project as independent work.
- Projects to support concurrent sorting in the final section of Chapter 12.
- An inheritance project, such as the ecosystem of Section 13.3.
- A graph class and associated graph algorithms in Chapter 14. This is another case in which advanced students may do work on their own.

Java Language Versions

All the source code in the book has been tested to work correctly with Java 2 Standard Edition Version 7.0, including new features such as generics and new concurrency support. Information on all of the Java products from Sun Microsystems is available at http://java.sun.com/products/index.html.

Flexibility of Topic Ordering

This book was written to give instructors latitude in reordering the material to meet the specific background of students or to add early emphasis to selected topics. The dependencies among the chapters are shown on the next page. A line joining two boxes indicates that the upper box should be covered before the lower box.

Here are some suggested orderings of the material:

Typical Course. Start with Chapters 1–9, skipping parts of Chapter 2 if the students have a prior background in Java classes. Most chapters can be covered in a week, but you may want more time for Chapter 4 (linked lists), Chapter 8 (recursion), or Chapter 9 (trees). Typically, I cover the material in 13 weeks,
Chapter Dependencies

At the start of the course, students should be comfortable writing application programs and using arrays in Java.

Chapter 1
Introduction

Chapters 2–3
Classes
Reference variables
Collection classes
Chapter 2 can be skipped by students with a good background in Java classes.

Chapter 2–3
Classes
Reference variables
Collection classes

Chapter 4
Linked lists

Sections 5.1–5.4
Generic programming

Chapter 5
Stacks

Chapter 6
Queues

Chapter 7
Trees

Section 10.1
Heaps

Section 10.2
B-trees

Section 10.3
Java trees

Section 10.4
Detailed tree analysis

Section 11.1
Binary search

Sec. 11.2–11.3
Hash tables
(Also requires Chapter 5)

Chapter 12
Sorting
(Heapsort also needs Section 10.1)

Chapter 13
Extended classes

Sections 5.5–5.7
The Java API
Iterators
Java collections
Java maps

Chapter 8
Recursion

Chapter 9
Trees

Chapter 14
Graphs

The shaded boxes provide good opportunities for advanced work.

including time for exams and extra time for linked lists and trees. Remaining weeks can be spent on a tree project from Chapter 10 or on binary search (Section 11.1) and sorting (Chapter 12).

**Heavy OOP Emphasis.** If students will cover sorting and searching elsewhere, then there is time for a heavier emphasis on object-oriented programming. The first three chapters are covered in detail, and then derived classes (Section 13.1) are introduced. At this point, students can do an interesting OOP project, perhaps based on the ecosystem of Section 13.3. The basic data structures (Chapters 4–7) are then covered, with the queue implemented as a derived class (Section 13.4). Finish up with recursion (Chapter 8) and trees (Chapter 9), placing special emphasis on recursive methods.

**Accelerated Course.** Assign the first three chapters as independent reading in the first week and start with Chapter 4 (linked lists). This will leave two to three extra weeks at the end of the term so that students can spend more time on searching, sorting, and the advanced topics (shaded in the chapter dependencies list).

I also have taught the course with further acceleration by spending no lecture time on stacks and queues (but assigning those chapters as reading).

**Early Recursion / Early Sorting.** One to three weeks may be spent at the start of class on recursive thinking. The first reading will then be Chapters 1 and 8, perhaps supplemented by additional recursive projects.

If the recursion is covered early, you may also proceed to cover binary search (Section 11.1) and most of the sorting algorithms (Chapter 12) before introducing collection classes.

**Supplements Via the Internet**

The following materials are available to all readers of this text at cssupport.pearsoncmg.com (or alternatively at www.cs.colorado.edu/~main/dsoj.html):

- Source code
- Errata

In addition, the following supplements are available to qualified instructors. Visit Addison-Wesley’s Instructor Resource Center (www.aw.com/irc) or contact your local Addison-Wesley representative for access to these:

- PowerPoint® presentations
- Exam questions
- Solutions to selected programming projects
- Speaker notes
- Sample assignments
- Suggested syllabi
Acknowledgments

This book grew from joint work with Walter Savitch, who continues to be an ever-present and enthusiastic supporter, colleague, and friend. My students from the University of Colorado at Boulder serve to provide inspiration and joy at every turn, particularly the spring seminars in Natural Computing and Ideas in Computing. During the past few years, the book has also been extensively reviewed by J.D. Baker, Philip Barry, Arthur Crummer, Herbert Dershem, Greg Dobbins, Zoran Duric, Dan Grecu, Scott Grissom, Bob Holloway, Rod Howell, Danny Krizanc, Ran Libeskind-Hadas, Meiliu Lu, Catherine Matthews, Robert Moll, Robert Pastel, Don Slater, Ryan Stansifer, Deborah Trytten, and John Wegis. I thank these colleagues for their excellent critique and their encouragement.

At Addison-Wesley, I thank Tracy Dunkelberger, Michael Hirsch, Bob Engelhardt, and Stephanie Sellinger, who have provided continual support and knowledgeable advice.

I also thank my friends and colleagues who have given me daily encouragement and friendship during the writing of this fourth edition: Andrzej Ehrenfeucht, Marga Powell, Grzegorz Rozenberg, and Allison Thompson-Brown, and always my family: Janet, Tim, Hannah, Michelle, and Paul.

Michael Main (main@colorado.edu)
Boulder, Colorado