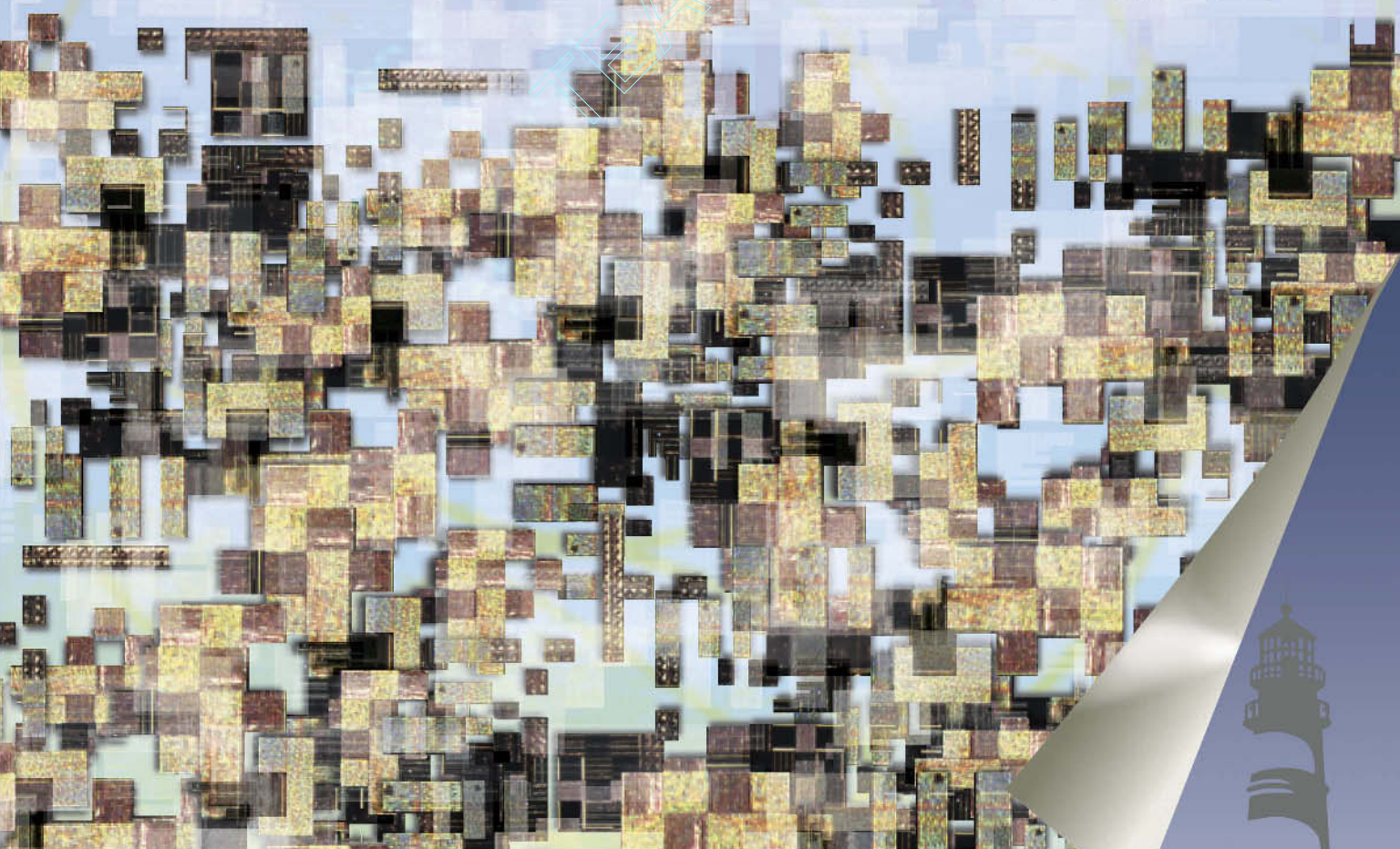
Object-Oriented Data Structures Using Java™

Nell Dale
Daniel T. Joyce
Chip Weems



JONES AND BARTLETT COMPUTER SCIENCE

Object-Oriented Data Structures Using Java™

Nell Dale
University of Texas, Austin

Daniel T. Joyce
Villanova University

Chip Weems
University of Massachusetts, Amherst

JONES AND BARTLETT PUBLISHERS
Sudbury, Massachusetts
BOSTON TORONTO LONDON SINGAPORE

World Headquarters
Jones and Bartlett Publishers
40 Tall Pine Drive
Sudbury, MA 01776
978-443-5000
info@jbpub.com
www.jbpub.com

Jones and Bartlett Publishers Canada
2406 Nikanna Road
Mississauga, Ontario
Canada L5C 2W6

Jones and Bartlett Publishers International
Barb House, Barb Mews
London W6 7PA
UK

Library of Congress Cataloging-in-Publication Data

Dale, Nell B.
Object-oriented data structures using Java / Nell Dale, Daniel T. Joyce, Chip Weems.
p. cm.
ISBN 0-7637-1079-2
1. Object-oriented programming (Computer science) 2. Data structures (Computer science) 3. Java (Computer program language) I. Joyce, Daniel T. II. Weems, Chip. III. Title.
QA76.64 .D35 2001
005.13'3—dc21

2001050374

Cover art courtesy of June Dale

Chief Executive Officer: Clayton Jones
Chief Operating Officer: Don W. Jones, Jr.
Executive V.P., and Publisher: Robert W. Holland, Jr.
V.P., Managing Editor: Judith H. Hauck
V.P., Design and Production: Anne Spencer
V.P., Manufacturing and Inventory Control: Therese Bräuer
Editor-in-Chief: J. Michael Stranz
Development and Product Manager: Amy Rose
Marketing Manager: Nathan Schultz
Production Assistant: Tara McCormick
Cover Design: Kristin E. Ohlin
Composition: Northeast Compositors, Inc.
Text Design: Anne Spencer
Printing and Binding: Courier Westford
Cover printing: John Pow Company, Inc.

This book was typeset in Quark 4.1 on a Macintosh G4. The font families used were Rotis Sans Serif, Rotis Serif, Industria, and Prestige Elite. The first printing was printed on 45# Highland Plus.

Printed in the United States of America
05 04 03 02 10 9 8 7 6 5 4 3 2

To Al, my husband and best friend.

N.D.

To Mike, Pat, Pete, Chris, Phil, Paul, Mary Anne.
"What a family!"

D.J.

To Lisa, Charlie, and Abby with love.

C.W.

preface

Welcome to the first edition of *Object-Oriented Data Structures using Java*. This book has been written to present the algorithmic, programming, and structuring techniques of a traditional data structures course in an object-oriented context. You'll find that all of the familiar topics of lists, stacks, queues, trees, graphs, sorting, searching, Big-O complexity analysis, and recursion are still here, but covered from an object-oriented point of view using Java. Thus, our structures are defined with Java interfaces and encapsulated as Java classes. We use abstract classes and inheritance, as appropriate, to take advantage of the relationships among various versions of the data structures. We use design aids, such as Class-Responsibility-Collaborator (CRC) Cards and Universal Modeling Language (UML) diagrams, to help us model and visualize our classes and their interrelationships. We hope that you enjoy this modern and up-to-date approach to the traditional data structures course.

Abstract Data Types

Over the last 16 years, the focus of the data structures course has broadened considerably. The topic of data structures now has been subsumed under the broader topic of *abstract data types (ADTs)*—the study of classes of objects whose logical behavior is defined by a set of values and a set of operations.

The term *abstract data type* describes a domain of values and set of operations that are specified independently of any particular implementation. The shift in emphasis is representative of the move towards more abstraction in computer science education. We now are interested in the study of the abstract properties of classes of data objects in addition to how the objects might be represented in a program.

The data abstraction approach leads us, throughout the book, to view our data structures from three different perspectives: their specification, their application, and their implementation. The specification describes the logical or abstract level. This level is concerned with *what* the operations are and *what* they do. The application level, sometimes called the user level, is concerned with how the data type might be used to solve a problem. This level is concerned with *why* the operations do what

they do. The implementation level is where the operations are actually coded. This level is concerned with the *how* questions.

Using this approach, we stress computer science theory and software engineering principles, including modularization, data encapsulation, information hiding, data abstraction, stepwise refinement, visual aids, the analysis of algorithms, and software verification methods. We feel strongly that these principles should be introduced to computer science students early in their education so that they learn to practice good software techniques from the beginning.

An understanding of theoretical concepts helps students put the new ideas they encounter into place, and practical advice allows them to apply what they have learned. To teach these concepts we consistently use intuitive explanations, even for topics that have a basis in mathematics, like the analysis of algorithms. In all cases, our highest goal has been to make our explanations as readable and as easily understandable as possible.

Prerequisite Assumptions

In this book, we assume that readers are familiar with the following Java constructs.

- Built-in simple data types
- Control structures *while*, *do*, *for*, *if*, and *switch*
- Creating and instantiating objects
- Basic user-defined classes
  - variables and methods
  - constructors, method parameters, and the *return* statement
  - visibility modifiers
- Built-in array types
- Basic string operations

We have included a review within the text to refresh the student's memory concerning some of the details of these topics (for example, defining/using classes and using strings).

Input/Output

It is difficult to know what background the students using a data structures textbook will have in Java I/O. Some may have learned Java in an environment where the Java input/output statements were "hidden" behind a package provided with their introductory textbook. Others may have learned graphical input/output techniques, but never learned how to do file input/output. Some have learned how to create graphical interfaces using the Java AWT; others have learned Swing; others have learned neither. Therefore, we have taken the following approach to I/O:

We assume the student has very little background.

We establish our "standard" I/O approach early—in the test driver developed at the end of the first chapter. The test driver uses command line parameters for input, basic text file input and output, and simple screen output based on Java's Swing classes.

Except for the case studies, we restrict our use of I/O throughout the text to the set of techniques used in the test driver.

We explain the I/O techniques used in the test driver in the *Java Input/Output I* feature section at the end of Chapter 1.

The only places in the text where more advanced I/O approaches are used are in the case studies. Beginning with Chapter 3, we develop case studies as examples of "real" programs that use the data structures we are studying. These case studies use progressively more advanced graphical interfaces, and are accompanied by additional feature sections as needed to explain any new constructs. Therefore, the case studies not only provide examples of object-oriented design and uses of data structures, they progressively introduce the student to user interface design techniques.

Content and Organization

We like to think that the material in Chapters 1 and 2 is a review for most students. However, the concepts in these two chapters are so crucial to the future of any and all students that we cannot rely on their having seen the material before. Even students who are familiar with the topics in these chapters can benefit from a review of the material since it is usually beneficial to see things from more than one perspective.

Here is a chapter-by-chapter overview of the textbook contents:

Chapter 1 outlines the basic goals of high-quality software and the basic principles of software engineering for designing and implementing programs to meet these goals. Abstraction, stepwise refinement, and object-oriented design are discussed. Some principles of object-oriented programming—encapsulation and inheritance—are introduced here. The UML class diagram is used as a tool for visualizing class characteristics and relationships. CRC cards are used in an introductory design example. This chapter also addresses what we see as a critical need in software education: the ability to design and implement correct programs and to verify that they are actually correct. Topics covered include the concept of "life-cycle" verification; designing for correctness using preconditions and postconditions; the use of deskchecking and design/code walk-throughs and inspections to identify errors before testing; debugging techniques, data coverage (black box), and code coverage (clear or white box) approaches; and test plans. As we develop ADTs in subsequent chapters, we discuss the construction of an appropriate test plan for each. The chapter culminates with the development of a test driver to aid in the testing of a simple programmer-defined class. The test driver has the additional benefit of introducing the basic I/O techniques used throughout the rest of the text.

Chapter 2 presents data abstraction and encapsulation, the software engineering concepts that relate to the design of the data structures used in programs. Three perspectives of data are discussed: abstraction, implementation, and application. These perspectives are illustrated using a real-world example (a library), and then are applied to built-in data structures that Java supports: primitive types, classes, interfaces, and arrays. The Java class type is presented as the way to represent the abstract data types we examine in subsequent chapters. We also look at several useful Java library classes,

including exceptions, wrappers, and strings. A feature section warns of the pitfalls of using references, which are the only means available to us for manipulating objects in Java.

Chapter 3 introduces a fundamental abstract data type: the list. The chapter begins with a general discussion of lists and then presents lists using the framework with which all of the other data structures are examined: a presentation and discussion of the specification, a brief application using the operations, and the design and coding of the operations. Both the unsorted and the sorted lists are presented with an array-based implementation. The binary search is introduced as a way to improve the performance of the search operation in the sorted list. Because there is more than one way to solve a problem, we discuss how competing solutions can be compared through the analysis of algorithms, using Big-O notation. This notation is then used to compare the operations in the unsorted list and the sorted list. The chapter begins with the presentation of an unsorted string list ADT. However, by the end of the chapter we have introduced *abstract classes* to allow us to take advantage of the common features of sorted and unsorted lists, and *interfaces* to enable us to implement generic lists. The chapter case study takes a simple real estate database, demonstrates the object-oriented design process, and concludes with the actual coding of a problem in which the sorted list is the principal data object. The development of the code for the case study introduces the use of interactive frame-based input.

Chapter 4 presents the stack and the queue data types. Each data type is first considered from its abstract perspective, and the idea of recording the logical abstraction in an ADT specification as a Java *interface* is stressed. The Stack ADT is implemented in Java using both an array-based approach and an array-list based approach. The Queue ADT is implemented using the array-based approach. A feature section discusses the options of implementing data structures "by copy" or "by reference." Example applications using both stacks (checking for balanced parenthesis) and queues (checking for palindromes), plus a case study using stacks (postfix expression evaluator) are presented. The chapter also includes a section devoted to the Java library's collection framework; that is, the lists, stacks, queues and so on that are available in the standard Java library.

Chapter 5 reimplements the ADTs from Chapters 3 and 4 as linked structures. The technique used to link the elements in dynamically allocated storage is described in detail and illustrated with figures. The array-based implementations and the linked implementations are then compared using Big-O notation. The chapter culminates with a review of our list framework, as it evolved in Chapters 3, 4, and 5, to use two interfaces, two abstract classes, and four concrete classes.

Chapter 6 looks at some alternate approaches for lists: circular linked lists, doubly linked lists, and lists with headers and trailers. An alternative representation of a linked structure, using static allocation (an array of nodes), is designed. The case study uses a list ADT developed specifically to support the implementation of large integers.

Chapter 7 discusses recursion, first providing an intuitive view of the concept, and then showing how recursion can be used to solve programming problems. Guidelines for writing recursive methods are illustrated with many examples. After demonstrating that

a by-hand simulation of a recursive routine can be very tedious, a simple three-question technique is introduced for verifying the correctness of recursive methods. Because many students are wary of recursion, the introduction to this material is deliberately intuitive and nonmathematical. A more detailed discussion of how recursion works leads to an understanding of how recursion can be replaced with iteration and stacks.

Chapter 8 introduces binary search trees as a way to arrange data, giving the flexibility of a linked structure with $O(\log_2 N)$ insertion and deletion time. We build on the previous chapter and exploit the inherent recursive nature of binary trees, by presenting recursive algorithms for many of the operations. We also address the problem of balancing binary search trees and implementing them with an array. The case study discusses the process of building an index for a manuscript and implements the first phase.

Chapter 9 presents a collection of other ADTs: priority queues, heaps, and graphs. The graph algorithms make use of stacks, queues, and priority queues, thus both reinforcing earlier material and demonstrating how general these structures are. The chapter ends with a section discussing how we can store objects (that could represent data structures) in files for later use.

Chapter 10 presents a number of sorting and searching algorithms and asks the question: which are better? The sorting algorithms that are illustrated, implemented, and compared include straight selection sort, two versions of bubble sort, insertion sort, quick sort, heap sort, and merge sort. The sorting algorithms are compared using Big-O notation. The discussion of algorithm analysis continues in the context of searching. Previously presented searching algorithms are reviewed and new ones are described. Hashing techniques are discussed in some detail.

Additional Features

Chapter Goals A set of goals presented at the beginning of each chapter helps the students assess what they have learned. These goals are tested in the exercises at the end of each chapter.

Chapter Exercises Most chapters have 30 or more exercises, organized by chapter sections to make it easy to assign the exercises. They vary in levels of difficulty, including short and long programming problems, the analysis of algorithms, and problems to test the student's understanding of concepts. Approximately one-third of the exercises are answered in the back of the book.

Chapter Summaries Each chapter concludes with a summary section that reviews the most important topics of the chapter and ties together related topics.

Chapter Summary of Classes and Support Files The end of each chapter also includes a table showing the set of author-defined classes/interfaces and support files introduced in the chapter and another table showing the set of Java library classes/interfaces/methods used in the chapter for the first time.

Sample Programs There are many sample programs and program segments illustrating the abstract concepts throughout the text.

Case Studies There are four major case studies. Each includes a problem description, an analysis of the problem input and required output, and a discussion of the appropriate data structures to use. The case studies are completely coded and tested.

Appendices The appendices summarize the Java reserved word set, operator precedence, primitive data types, and the ASCII subset of Unicode.

Web Site Jones and Bartlett has designed a web site to support this text. At http://oodatastructures.jbpub.com, students will find a glossary and most of the source code presented in the text. Instructors will find teaching notes, in-class activity suggestions, answers to those questions that are not in the back of the book, and PowerPoint presentations for each chapter. To obtain a password for this site, please contact Jones and Bartlett at 1-800-832-0034. Please contact the authors if you have material related to the text that you would like to share with others.

Acknowledgments

We would like to thank the following people who took the time to review this manuscript: John Amanatides, York University; Ric Heishman, North Virginia Community College; Neal Alderman, University of Connecticut; and Vladan Jovanovic, University of Detroit Mercy

Also, thanks to John Lewis and Maulan Bryon, both of Villanova University. John was always happy to discuss interesting design and coding problems and Maulan helped with programming.

A virtual bouquet of roses to the people who have worked on this book: Mike and Sigrid Wile along with the many people at Jones and Bartlett who contributed so much, especially J. Michael Stranz, Amy Rose, and Tara McCormick.

Nell thanks her husband Al, their children and grandchildren too numerous to name, and their dogs Maggie and Bear.

Dan thanks his wife Kathy for putting up with the extra hours of work and the disruption in the daily routine. He also thanks Tom, age 11, for helping with proofreading and Julie, age 8, for lending her gel pens for use during the copyediting process.

Chip thanks Lisa, Charlie, and Abby for being understanding of all the times he has been late for dinner, missed saying goodnight, couldn't stop to play, or had to skip a bike ride. The love of a family is fuel for an author.

N. D.
D. J.
C. W.

Contents

Chapter 1

Software Engineering

Goals

Measurable goals for this chapter include that you should be able to

- describe software life cycle activities
- describe the goals for "quality" software
- explain the following terms: software requirements, software specifications, algorithm, information hiding, abstraction, stepwise refinement
- describe four variations of stepwise refinement
- explain the fundamental ideas of object-oriented design
- explain the relationships among classes, objects, and inheritance and show how they are implemented in Java
- explain how CRC cards are used to help with software design
- interpret a basic UML state diagram
- identify sources of software errors
- describe strategies to avoid software errors
- specify the preconditions and postconditions of a program segment or method
- show how deskchecking, code walk-throughs, and design and code inspections can improve software quality and reduce effort
- explain the following terms: acceptance tests, regression testing, verification, validation, functional domain, black box testing, white box testing
- state several testing goals and indicate when each would be appropriate
- describe several integration-testing strategies and indicate when each would be appropriate
- explain how program verification techniques can be applied throughout the software development process
- create a Java test driver program to test a simple class

At this point you have completed at least one semester of computer science course work. You can take a problem of medium complexity, design a set of objects that work together to solve the problem, code the method algorithms needed to make the objects work, and demonstrate the correctness of your solution.

In this chapter, we review the software process, object-oriented design, and the verification of software correctness.

1.1 The Software Process

When we consider computer programming, we immediately think of writing code in some computer language. As a beginning student of computer science, you wrote programs that solved relatively simple problems. Much of your effort went into learning the syntax of a programming language such as Java or C++: the language's reserved words, its data types, its constructs for selection and looping, and its input/output mechanisms.

You learned a programming methodology that takes you from a problem description all the way through to the delivery of a software solution. There are many design techniques, coding standards, and testing methods that programmers use to develop high-quality software. Why bother with all that methodology? Why not just sit down at a computer and enter code? Aren't we wasting a lot of time and effort, when we could just get started on the "real" job?

If the degree of our programming sophistication never had to rise above the level of trivial programs (like summing a list of prices or averaging grades), we might get away with such a code-first technique (or, rather, a *lack* of technique). Some new programmers work this way, hacking away at the code until the program works more or less correctly—usually less!

As your programs grow larger and more complex, you must pay attention to other software issues in addition to coding. If you become a software professional, you may work as part of a team that develops a system containing tens of thousands, or even millions, of lines of code. The activities involved in such a software project's whole "life cycle" clearly go beyond just sitting down at a computer and writing programs. These activities include:

- *Problem analysis* Understanding the nature of the problem to be solved
- *Requirements elicitation* Determining exactly what the program must do
- *Software specification* Specifying what the program must do (the functional requirements) and the constraints on the solution approach (nonfunctional requirements, such as what language to use)
- *High-* and *low-level design* Recording how the program meets the requirements, from the "big picture" overview to the detailed design
- *Implementation of the design* Coding a program in a computer language
- *Testing and verification* Detecting and fixing errors and demonstrating the correctness of the program
- *Delivery* Turning over the tested program to the customer or user (or instructor)

- *Operation* Actually using the program
- *Maintenance* Making changes to fix operational errors and to add or modify the function of the program

Software development is not simply a matter of going through these steps sequentially. Many activities take place concurrently. We may be coding one part of the solution while we're designing another part, or defining requirements for a new version of a program while we're still testing the current version. Often a number of people work on different parts of the same program simultaneously. Keeping track of all these activities requires planning.

We use the term software engineering to refer to the discipline concerned with all aspects of the development of high-quality software systems. It encompasses *all* variations of techniques used during the software life cycle plus supporting activities such as documentation and teamwork. A software process is a specific set of inter-related software engineering techniques used by a person or organization to create a system.

Software engineering The discipline devoted to the design, production, and maintenance of computer programs that are developed on time and within cost estimates, using tools that help to manage the size and complexity of the resulting software products

Software process A standard, integrated set of software engineering tools and techniques used on a project or by an organization

What makes our jobs as programmers or software engineers challenging is the tendency of software to grow in size and complexity and to change at every stage of its development. Part of a good software process is the use of tools to manage this size and complexity. Usually a programmer has several toolboxes, each containing tools that help to build and shape a software product.

Hardware

One toolbox contains the hardware itself: the computers and their peripheral devices (such as monitors, terminals, storage devices, and printers), on which and for which we develop software.

Software

A second toolbox contains various software tools: operating systems, editors, compilers, interpreters, debugging programs, test-data generators, and so on. You've used some of these tools already.

Ideaware

A third toolbox is filled with the knowledge that software engineers have collected over time. This box contains the algorithms that we use to solve common programming problems, as well as data structures for modeling the information processed by our programs. Recall that an algorithm is a step-by-step description of the solution to a problem.

Algorithm A logical sequence of discrete steps that describes a complete solution to a given problem computable in a finite amount of time and space

Ideaware contains programming methodologies, such as object-oriented design, and

software concepts, including information hiding, data encapsulation, and abstraction. It includes aids for creating designs such as CRC (Classes, Responsibilities, and Collaborations) cards and methods for describing designs such as the UML (Unified Modeling Language). It also contains tools for measuring, evaluating, and proving the correctness of our programs. We devote most of this book to exploring the contents of this third toolbox.

Some might argue that using these tools takes the creativity out of programming, but we don't believe that to be true. Artists and composers are creative, yet their innovations are grounded in the basic principles of their crafts. Similarly, the most creative programmers build high-quality software through the disciplined use of basic programming tools.

Goals of Quality Software

Quality software is much more than a program that accomplishes its task. A good program achieves the following goals:

1. It works.
2. It can be modified without excessive time and effort.
3. It is reusable.
4. It is completed on time and within budget.

It's not easy to meet these goals, but they are all important.

Goal 1: Quality Software Works

A program must accomplish its task, and it must do it correctly and completely. Thus, the first step is to determine exactly what the program is required to do. You need to have a definition of the program's requirements. For students, the requirements often are included in the instructor's problem description. For programmers on a government contract, the requirements document may be hundreds of pages long.

Requirements A statement of what is to be provided by a computer system or software product

Software specification A detailed description of the function, inputs, processing, outputs, and special requirements of a software product. It provides the information needed to design and implement the product.

We develop programs that meet the requirements by fulfilling software specifications. The specifications indicate the format of the input and output, details about processing, performance measures (how fast? how big? how accurate?), what to do in case of errors, and so on. The specifications tell *what* the program does, but not *how* it is done. Sometimes your instructor provides detailed specifications; other times you have to write them yourself, based on a problem description, conversations with your instructor, or intuition.

How do you know when the program is right? A program has to be

- *complete:* it should "do everything" specified
- *correct:* it should "do it right"
- *usable:* its user interface should be easy to work with
- *efficient:* at least as efficient as "it needs to be"

For example, if a desktop-publishing program cannot update the screen as rapidly as the user can type, the program is not as efficient as it needs to be. If the software isn't efficient enough, it doesn't meet its requirements, and thus, according to our definition, it doesn't work correctly.

Goal 2: Quality Software Can Be Modified

When does software need to be modified? Changes occur in every phase of its existence.

Software is changed in the design phase. When your instructor or employer gives you a programming assignment, you begin to think of how to solve the problem. The next time you meet, however, you may be notified of a change in the problem description.

Software is changed in the coding phase. You make changes in your program because of compilation errors. Sometimes you see a better solution to a part of the problem after the program has been coded, so you make changes.

Software is changed in the testing phase. If the program crashes or yields wrong results, you must make corrections.

In an academic environment, the life of the software typically ends when a program is turned in for grading. When software is developed for actual use, however, many changes can be required during the maintenance phase. Someone may discover an error that wasn't uncovered in testing, someone else may want to include additional functionality, a third party may want to change the input format, and a fourth party may want to run the program on another system.

The point is that software changes often and in all phases of its life cycle. Knowing this, software engineers try to develop programs that are easy to modify. Modifications to programs often are not even made by the original authors but by subsequent maintenance programmers. Someday you may be the one making the modifications to someone else's program.

What makes a program easy to modify? First, it should be readable and understandable to humans. Before it can be changed, it must be understood. A well-designed, clearly written, well-documented program is certainly easier for human readers to understand. The number of pages of documentation required for "real-world" programs usually exceeds the number of pages of code. Almost every organization has its own policy for documentation.

Second, it should be able to withstand small changes easily. The key idea is to partition your programs into manageable pieces that work together to solve the problem, yet are relatively independent. The design methodologies reviewed later in this chapter should help you write programs that meet this goal.

Goal 3: Quality Software Is Reusable

It takes time and effort to create quality software. Therefore, it is important to receive as much value from the software as possible.

One way to save time and effort when building a software solution is to reuse programs, classes, methods, and so on from previous projects. By using previously designed and tested code, you arrive at your solution sooner and with less effort. Alternatively, when you create software to solve a problem, it is sometimes possible to structure that software so it can help solve future, related problems. By doing this, you are gaining more value from the software created.

Creating reusable software does not happen automatically. It requires extra effort during the specification and design of the software. Reusable software is well documented and easy to read, so that it is easy to tell if it can be used for a new project. It usually has a simple interface so that it can easily be plugged into another system. It is modifiable (Goal 2), in case a small change is needed to adapt it to the new system.

When creating software to fulfill a narrow, specific function, you can sometimes make the software more generally useable with a minimal amount of extra effort. Therefore, you increase the chances that you will reuse the software later. For example, if you are creating a routine that sorts a list of integers into increasing order, you might generalize the routine so that it can also sort other types of data. Furthermore, you could design the routine to accept the desired sort order, increasing or decreasing, as a parameter.

One of the main reasons for the rise in popularity of object-oriented approaches is that they lend themselves to reuse. Previous reuse approaches were hindered by inappropriate units of reuse. If the unit of reuse is too small, then the work saved is not worth the effort. If the unit of reuse is too large, then it is difficult to combine it with other system elements. Object-oriented classes, when designed properly, can be very appropriate units of reuse. Furthermore, object-oriented approaches simplify reuse through class inheritance, which is described later in this chapter.

Goal 4: Quality Software Is Completed on Time and within Budget

You know what happens in school when you turn your program in late. You probably have grieved over an otherwise perfect program that received only half credit—or no credit at all—because you turned it in one day late. "But the network was down for five hours last night!" you protest.

Although the consequences of tardiness may seem arbitrary in the academic world, they are significant in the business world. The software for controlling a space launch must be developed and tested before the launch can take place. A patient database system for a new hospital must be installed before the hospital can open. In such cases, the program doesn't meet its requirements if it isn't ready when needed.

"Time is money" may sound trite but failure to meet deadlines is *expensive*. A company generally budgets a certain amount of time and money for the development of a piece of software. If part of a project is only 80% complete when the deadline arrives, the company must pay extra to finish the work. If the program is part of a contract with a customer, there may be monetary penalties for missed deadlines. If it is being developed for commercial sales, the company may be beaten to the market by a competitor and be forced out of business.

Once you know what your goals are, what can you do to meet them? Where should you start? There are many tools and techniques that software engineers use. In the next few sections of this chapter, we focus on a review of techniques to help you understand, design, and code programs.

Specification: Understanding the Problem

No matter what programming design technique you use, the first steps are the same. Imagine the following situation. On the third day of class, you are given a 12-page description of Programming Assignment 1, which must be running perfectly and turned

in by noon, a week from yesterday. You read the assignment and realize that this program is three times larger than any program you have ever written. Now, what is your first step?

The responses listed here are typical of those given by a class of students in such a situation:

1. Panic and do nothing 39%
2. Panic and drop the course 30%
3. Sit down at the computer and begin typing 27%
4. Stop and think 4%

Response 1 is a predictable reaction from students who have not learned good programming techniques. Students who adopt Response 2 find their education progressing rather slowly. Response 3 may seem to be a good idea, especially considering the deadline looming. Resist the temptation, though, to immediately begin coding; the first step is to *think*. Before you can come up with a program solution, you must understand the problem. Read the assignment, and then read it again. Ask questions of your instructor to clarify the assignment. Starting early affords you many opportunities to ask questions; starting the night before the program is due leaves you no opportunity at all.

One problem with coding first and thinking later is that it tends to lock you into the first solution you think of, which may not be the best approach. We have a natural tendency to believe that once we've put something in writing, we have invested too much in the idea to toss it out and start over.

Writing Detailed Specifications

Many writers experience a moment of terror when faced with a blank piece of paper—where to begin? As a programmer, however, you should always have a place to start. Using the assignment description, first write a complete definition of the problem, including the details of the expected inputs and outputs, the processing and error handling, and all the assumptions about the problem. When you finish this task, you have a *specification*—a definition of the problem that tells you what the program should do. In addition, the process of writing the specification brings to light any holes in the requirements. For instance, are embedded blanks in the input significant or can they be ignored? Do you need to check for errors in the input? On what computer system(s) is your program to run? If you get the answers to these questions at this stage, you can design and code your program correctly from the start.

Many software engineers make use of operational *scenarios* to understand requirements. A scenario is a sequence of events for *one* execution of the program. Here, for example, is a scenario that a designer might consider when developing software for a bank's automated teller machine (ATM).

1. The customer inserts a bankcard.
2. The ATM reads the account number on the card.
3. The ATM requests a PIN (personal identification number) from the customer.
4. The customer enters 5683.
5. The ATM successfully verifies the account number and PIN combination.

6. The ATM asks the customer to select a transaction type (deposit, show balance, withdrawal, or quit).
7. The customer selects show balance.
8. The ATM obtains the current account balance ($1,204.35) and displays it.
9. The ATM asks the customer to select a transaction type (deposit, show balance, withdrawal, or quit).
10. The customer selects quit.
11. The ATM returns the customer's bankcard.

Scenarios allow us to get a feel for the behavior expected from the system. A single scenario cannot show all possible behaviors, however, so software engineers typically prepare many different scenarios to gain a full understanding of the requirements.

Sometimes details that are not explicitly stated in the requirements may be handled according to the programmer's preference. In some cases you have only a vague description of a problem, and it is up to you to define the entire software specification; these projects are sometimes called *open problems*. In any case, you should always document assumptions that you make about unstated or ambiguous details.

The specification clarifies the problem to be solved. However, it also serves as an important piece of program documentation. Sometimes it acts as a contract between a customer and a programmer. There are many ways in which specifications may be expressed and a number of different sections that may be included. Our recommended program specification includes the following sections:

- processing requirements
- sample inputs with expected outputs
- assumptions

If special processing is needed for unusual or error conditions, it too should be specified. Sometimes it is helpful to include a section containing definitions of terms used. It is also useful to list any testing requirements so that verifying the program is considered early in the development process. In fact, a test plan can be an important part of a specification; test plans are discussed later in this chapter in the section on verification of software correctness.

1.2 Program Design

Remember, the specification of the program tells *what* the program must do, but not *how* it does it. Once you have clarified the goals of the program, you can begin the design phase of the software life cycle. In this section, we review some ideaware tools that are used for software design and present a review of object-oriented design constructs and methods.

Tools

Abstraction

The universe is filled with complex systems. We learn about such systems through *models*. A model may be mathematical, like equations describing the motion of satellites around the earth. A physical object such as a model airplane used in wind-tunnel tests is another form of model. Only the characteristics of the system that are essential to the problem being studied are modeled; minor or irrelevant details are ignored. For example, although the earth is an oblate ellipsoid, globes (models of the earth) are spheres. The small difference in shape is not important to us in studying the political divisions and physical landmarks on the earth. Similarly, in-flight movies are not included in the model airplanes used to study aerodynamics.

An abstraction is a model of a complex system that includes only the essential details. Abstractions are the fundamental way that we manage complexity. Different viewers use different abstractions of a particular system. Thus, while we see a car as a means of transportation, the automotive engineer may see it as a large mass with a small contact area between it and the road (Figure 1.1).

Abstraction A model of a complex system that includes only the details essential to the perspective of the viewer of the system

What does abstraction have to do with software development? The programs we write are abstractions. A spreadsheet program used by an accountant models the books used to record debits and credits. An educational computer game about wildlife models an ecosystem. Writing software is difficult because both the systems we model and the processes we use to develop the software are complex. One of our major goals is to convince you to use abstractions to manage the complexity of developing software. In nearly every chapter, we make use of abstractions to simplify our work.

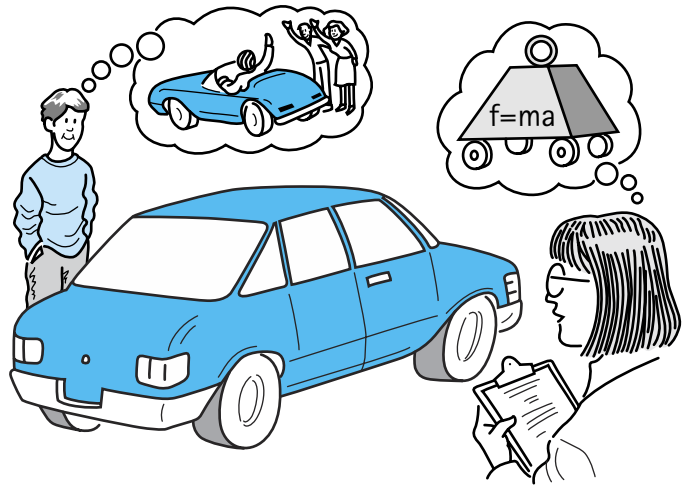


Figure 1.1 *An abstraction includes the essential details relative to the perspective of the viewer*

Information Hiding

Many design methods are based on decomposing a problem's solution into modules. By "module" we mean a cohesive system subunit that performs a share of the work. In Java, the primary module mechanism is the *class*. Decomposing a system into modules helps us manage complexity. Additionally, the modules can form the basis of assignments for different programming teams working separately on a large system.

Modules act as an abstraction tool. The complexity of their internal structure can be hidden from the rest of the system. This means that the details involved in implementing a module are isolated from the details of the rest of the system. Why is hiding the details desirable? Shouldn't the programmer know everything? *No!* Information hiding helps manage the complexity of a system since a programmer can concentrate on one module at a time.

Information hiding The practice of hiding the details of a module with the goal of controlling access to the details from the rest of the system

Of course, a program's modules are interrelated, since they work together to solve the problem. Modules provide services to each other through a carefully defined interface. The interface in Java is usually provided by the public methods of a class. Programmers of one module do not need to know the internal details of the modules it interacts with, but they do need to know the interfaces. Consider a driving analogy—you can start a car without knowing how many cylinders are in the engine. You don't need to know these lower-level details of the car's power subsystem in order to start it. You just have to understand the interface; that is, you only need to know how to turn the key.

Similarly, you don't have to know the details of other modules as you design a specific module. Such a requirement would introduce a greater risk of confusion and error throughout the whole system. For example, imagine what it would be like if every time we wanted to start our car, we had to think, "The key makes a connection in the ignition switch that, when the transmission safety interlock is in "park," engages the starter motor and powers up the electronic ignition system, which adjusts the spark and the fuel-to-air ratio of the injectors to compensate for…".

Besides helping us manage the complexity of a large system, abstraction and information hiding support our quality goals of modifiability and reusability. In a well-designed system, most modifications can be localized to just a few modules. Such changes are much easier to make than changes that permeate the entire system. Additionally, a good system design results in the creation of generic modules that can be used in other systems.

To achieve these goals, modules should be good abstractions with strong *cohesion*; that is, each module should have a single purpose or identity and the module should stick together well. A cohesive module can usually be described by a simple sentence. If you have to use several sentences or one very convoluted sentence to describe your module, it is probably *not* cohesive. Each module should also exhibit information hiding so that changes within it do not result in changes in the modules that use it. This independent quality of modules is known as *loose coupling*. If your module depends on the internal details of other modules, it is *not* loosely coupled.

But what should these modules be and how do we identify them? That question is addressed in the subsection on object-oriented design later in this chapter.

Stepwise Refinement

In addition to concepts such as abstraction and information hiding, software developers need practical approaches to conquer complexity. Stepwise refinement is a widely applicable approach. It has many variations such as top-down, bottom-up, functional decomposition and even "round-trip gestalt design." Undoubtedly, you have learned a variation of stepwise refinement in your studies, since it is a standard method for organizing and writing essays, term papers, and books. For example, to write a book an author first determines the main theme and the major subthemes. Next, the chapter topics can be identified, followed by section and subsection topics. Outlines can be produced and further refined for each subsection. At some point the author is ready to add detail—to actually begin writing sentences.

In general, with stepwise refinement, a problem is approached in stages. Similar steps are followed during each stage, with the only difference being the level of detail involved. The completion of each stage brings us closer to solving our problem. Let's look at some variations of stepwise refinement:

- Top-down: First the problem is broken into several large parts. Each of these parts is in turn divided into sections, then the sections are subdivided, and so on. The important feature is that *details are deferred as long as possible* as we move from a general to a specific solution. The outline approach to writing a book is a form of top-down stepwise refinement.
- Bottom-up: As you might guess, with this approach the details come first. It is the opposite of the top-down approach. After the detailed components are identified and designed, they are brought together into increasingly higher-level components. This could be used, for example, by the author of a cookbook who first writes all the recipes and then decides how to organize them into sections and chapters.
- Functional decomposition: This is a program design approach that encourages programming in logical action units, called functions. The main module of the design becomes the main program (also called the main function), and subsections develop into functions. This *hierarchy of tasks* forms the basis for functional decomposition, with the main program or function controlling the processing. Functional decomposition is not used for overall system design in the object-oriented world. However, it can be used to design the algorithms that implement object methods. The general function of the method is continually divided into sub-functions until the level of detail is fine enough to code. Functional decomposition is top-down stepwise refinement with an emphasis on functionality.
- Round-trip gestalt design: This confusing term is used to define the stepwise refinement approach to object-oriented design suggested by Grady Booch,[1] one of the leaders of the object movement. First, the tangible items and events in the problem domain are identified and assigned to candidate classes and objects.

---

[1]Grady Booch, *Object Oriented Design with Applications* (Redwood City, CA: Benjamin Cummings, 1991).

Next the external properties and relationships of these classes and objects are defined. Finally, the internal details are addressed, and unless these are trivial, the designer must return to the first step for another round of design. This approach is top-down stepwise refinement with an emphasis on objects and data.

Good designers typically use a combination of the stepwise refinement techniques described here.

Visual Aids

Abstraction, information hiding, and stepwise refinement are inter-related methods for controlling complexity during the design of a system. We will now look at some tools that we can use to help us visualize our designs. Diagrams are used in many professions. For example, architects use blueprints, investors use market trend graphs, and truck drivers use maps.

Software engineers use different types of diagrams and tables. Here, we introduce the *Unified Modeling Language* (*UML*) and *Class*, *Responsibility*, *and Collaboration* (*CRC*) *cards*, both of which are used throughout this text.

The UML is used to specify, visualize, construct, and document the components of a software system. It combines the best practices that have evolved over the past several decades for modeling systems, and is particularly well-suited to modeling object-oriented designs. UML diagrams are another form of abstraction. They hide implementation details and allow us to concentrate only on the major design components. UML includes a large variety of interrelated diagram types, each with its own set of icons and connectors. It is a very powerful development and modeling tool.

Covering all of UML is beyond the scope of this text.[2] We use only one UML diagram type, *detailed class diagrams*, to describe some of our designs. Examples are

[2]The official definition of the UML is maintained by the Object Management Group. Detailed information can be found at `http://www.omg.org/uml/`.

Class Name:	Superclass:	Subclassess:
Primary Responsibility		
Responsibilities	Collaborations	

Figure 1.2 *A blank CRC card*

shown beginning on page 16. The notation of the class diagrams is introduced as needed throughout the text.

UML class diagrams are good for modeling our designs after we have developed them. In contrast, CRC cards help us determine our designs in the first place. CRC cards were first described by Beck and Cunningham[3] in 1989 as a means of allowing object-oriented programmers to identify a set of cooperating classes to solve a problem.

A programmer uses a physical 4" × 6" index card to represent each class that has been identified as part of a problem solution. Figure 1.2 shows a blank CRC card. It contains room for the following information about a class:

1. Class name
2. Responsibilities of the class—usually represented by verbs and implemented by public methods
3. Collaborations—other classes/objects that are used in fulfilling the responsibilities

Thus the name CRC card. We have added fields to the original design of the card for the programmer to record superclass and subclass information, and the primary responsibility of the class.

[3]Beck and Cunningham: `http://c2.com/doc/oopsla89/paper.html`.

CRC cards are a great tool for refining an object-oriented design, especially in a team programming environment. They provide a physical manifestation of the building blocks of a system, allowing programmers to walk through user scenarios, identifying and assigning responsibilities and collaborations. The example in the next subsection demonstrates the use of CRC cards for design.

Object-Oriented Design

Review

Before describing approaches to object-oriented design, we present a short review of object-oriented programming. We use Java code to support this review.

The object-oriented paradigm is founded on three inter-related constructs: classes, objects, and inheritance. The inter-relationship among these constructs is so tight that it is nearly impossible to describe them separately. Objects are the basic run-time entities in an object-oriented system. An object is an instantiation of a class; or alternately, a class defines the structure of its objects. Classes are organized in an "is-a" hierarchy defined by inheritance. The definition of an object's behavior often depends on its position within this hierarchy. Let's look more closely at each of these constructs, using Java code to provide a concrete representation of the concepts. Java reserved words (when used as such), user-defined identifiers, class and method names, and so on appear in `this font` throughout the entire textbook.

Classes A class defines the structure of an object or a set of objects. A class definition includes variables (data) and methods (actions) that determine the behavior of an object. The following Java code defines a `Date` class that can be used to manipulate `Date` objects, for example, in a course scheduling system. The `Date` class can be used to create `Date` objects and to learn about the year, month, or day of any particular `Date` object.[4] Within the comments the word "this" is used to represent the current object.

```
public class Date
{
  protected int year;
  protected int month;
  protected int day;
  protected static final int MINYEAR = 1583;

  public Date(int newMonth, int newDay, int newYear)
  // Initializes this Date with the parameter values
```

[4]The Java library includes a `Date` class, `java.util.Date`. However, the familiar properties of dates make them a natural example to use in explaining object-oriented concepts. So we ignore the existence of the library class, as if we must design our own `Date` class.

```
  {
    month = newMonth;
    day = newDay;
    year = newYear;
   }

  public int yearIs()
  // Returns the year value of this Date
  {
    return year;
  }

  public int monthIs()
  // Returns the month value of this Date
  {
    return month;
  }

  public int dayIs()
  // Returns the day value of this Date
  {
    return day;
  }
}
```

The `Date` class demonstrates two kinds of variables: instance variables and class variables. The instance variables of this class are `year`, `month`, and `day`. Their values vary for each different instance of an object of the class. Instance variables represent the *attributes* of an object. `MINYEAR` is a class variable because it is defined to be static. It is associated directly with the `Date` class, instead of with objects of the class. A single copy of a static variable is maintained for all the objects of the class.

Remember that the `final` modifier states that a variable is in its final form and cannot be modified; thus `MINYEAR` is a constant. By convention, we use only capital letters when naming constants. It is standard procedure to declare constants as *static* variables. Since the value of the variable cannot change, there is no need to force every object of a class to carry around its own version of the value. In addition to holding shared constants, static variables can also be used to maintain information that is common to an entire class. For example, a Bank Account class may have a static variable that holds the number of current accounts.

In the above example, the `MINYEAR` constant represents the first full year that the widely used Gregorian calendar was in effect. The idea here is that programmers should not use the class to represent dates that predate that year. We look at ways to enforce this rule in Chapter 2.

The methods of the class are `Date`, `yearIs`, `monthIs`, and `dayIs`. Note that the `Date` method has the same name as the class. Recall that this means it is a special type

Observer A method that returns an observation on the state of an object.

of method, called a class constructor. Constructors are used to create new instances of a class—to instantiate objects of a class. The other three methods are classified as observer methods since they "observe" and return instance variable values. Another name for observer methods is "accessor" methods.

Once a class such as `Date` has been defined, a program can create and use objects of that class. The effect is similar to expanding the language's set of standard types to include a `Date` type—we discuss this idea further in Chapter 2. The UML class diagram for the `Date` class is shown in Figure 1.3. Note that the name of the class appears in the top section of the diagram, the variables appear in the next section, and the methods appear in the final section. The diagram includes information about the nature of the variables and method parameters; for example, we can see at a glance that `year`, `month`, and `day` are all of type `int`. Note that the variable `MINYEAR` is underlined, which indicates that it is a class variable rather than an instance variable. The diagram also indicates the visibility or protection associated with each part of the class (+ is public, # = protected)—we discuss visibility and protection in Chapter 2.

Objects Objects are created from classes at run-time. They can contain and manipulate data. You should view an object-oriented system as a set of objects, working together by sending each other messages to solve a problem.

To create an object in Java we use the *new* operator, along with the class constructor as follows:

```
Date myDate = new Date(6, 24, 1951);
Date yourDate = new Date(10, 11, 1953);
Date ourDate = new Date(6, 15, 1985);
```

We say that the variables `myDate`, `yourDate`, and `ourDate` reference "objects of the class `Date`" or simply "objects of type `Date`." We could also refer to them as "`Date` objects."

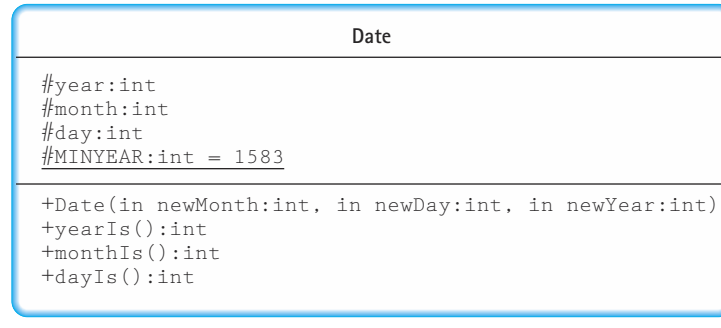


Figure 1.3 *UML class diagram for the* `Date` *class*

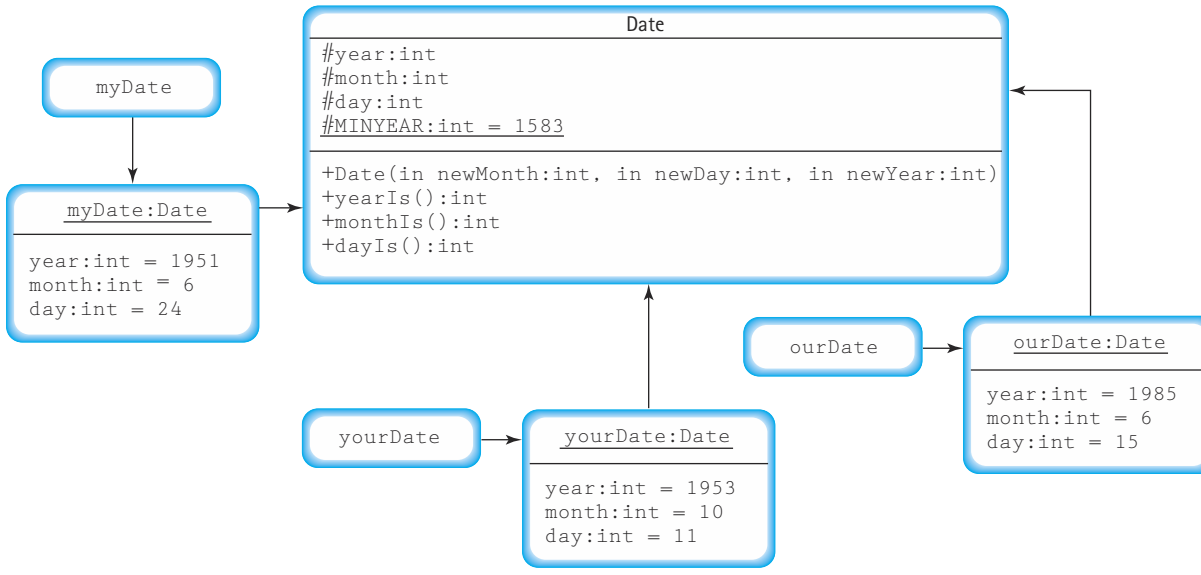


Figure 1.4 *Extended UML class diagram showing* `Date` *objects*

In Figure 1.4 we have extended the standard UML class diagram to show the relationship between the instantiated `Date` objects and the `Date` class.

As you can see, the objects are concrete instantiations of the class. Notice that the `myDate`, `yourDate`, and `ourDate` variables are not objects, but actually hold references to the objects. The references are shown by the pointers from the variable boxes to the objects. In reality, references are memory addresses. The memory address of the instantiated object is stored in the memory location assigned to the variable. If no object has been instantiated for a particular variable, then its memory location holds a `null` reference.

Object methods are invoked through the object upon which they are to act. For example, to assign the value of the `year` variable of `ourDate` to the integer variable `theYear`, a programmer would code

```
theYear = ourDate.yearIs();
```

Inheritance The object-oriented paradigm provides a powerful reuse tool called *inheritance*, which allows programmers to create a new class that is a specialization of an existing class. In this case, the new class is called a subclass of the existing class, which in turn is the superclass of the new class.

A subclass "inherits" features from its superclass. It adds new features, as needed, related to its specialization. It can also redefine inherited features as necessary. Contrary to the intuitive meaning of super and sub, a subclass usually has more variables and methods than its superclass. Super and sub refer to the relative positions of the classes

in a hierarchy. A subclass is below its superclass, and a superclass is above its subclasses.

Suppose we already have a `Date` class as defined above, and we are creating a new application to manipulate `Date` objects. Suppose also that in the new application we are often required to "increment" a `Date` variable—to change a `Date` variable so that it represents the next day. For example, if the `Date` object represents 7/31/2001, it would represent 8/1/2001 after being incremented. The algorithm for incrementing the date is not trivial, especially when you consider leap-year rules. But in addition to developing the algorithm, we must address another question: where to implement the algorithm. There are several options:

- Implement the algorithm within the new application. The code would need to obtain the month, day, and year from the `Date` object using the observer methods, calculate the new month, day, and year, instantiate a new `Date` object to hold the updated month, day, and year, and assign it to the same variable. This might appear to be a good approach, since it is the new application that requires the new functionality. However, if future applications also need this functionality, their programmers have to reimplement the solution for themselves. This approach does not support our goal of reusability.
- Add a new method, called `increment`, to the `Date` class. The code would use the incrementing algorithm to update the `month`, `year`, and `day` values of the current object. This approach is better than the previous approach because it allows any future programs that use the `Date` class to use the new functionality. However, this also means that *every* application that uses the `Date` class can use this method. In some cases, a programmer may have chosen to use the `Date` class because of its built-in protection against changes to the object variables. Such objects are said to be immutable. Adding an `increment` method to the `Date` class undermines this protection, since it allows the variables to be changed.
- Use inheritance. Create a new class, called `IncDate`, that inherits all the features of the current `Date` class, but that also provides the `increment` method. This approach resolves the drawbacks of the previous two approaches. We now look at how to implement this third approach.

We often call the inheritance relationship an *is a* relationship. In this case we would say that an object of the class `IncDate` is also a `Date` object, since it can do anything that a `Date` object can do—and more. This idea can be clarified by remembering that inheritance typically means specialization. `IncDate` is a special case of `Date`, but not the other way around.

To create `IncDate` in Java we would code:

```
public class IncDate extends Date
{
  public IncDate(int newMonth, int newDay, int newYear)
  // Initializes this IncDate with the parameter values
```

```
  {
    super(newMonth, newDay, newYear);
   }

  public void increment()
  // Increments this IncDate to represent the next day, i.e.,
  // this = (day after this)
  // For example if this = 6/30/2003 then this becomes 7/1/2003
  {
    // Increment algorithm goes here
  }
}
```

Note: sometimes in code listings we emphasize the sections of code most pertinent to the current discussion by underlining them.

Inheritance is indicated by the keyword `extends`, which shows that `IncDate` inherits from `Date`. It is not possible in Java to inherit constructors, so `IncDate` must supply its own. In this case, the `IncDate` constructor simply takes the month, day, and year parameters and passes them to the constructor of its superclass; it passes them to the `Date` class constructor using the `super` reserved word.

The other part of the `IncDate` class is the new `increment` method, which is classified as a transformer method, because it changes the internal state of the object. `increment` changes the object's `day` and possibly the `month` and `year` values. The `increment` transformer method is invoked through the object that it is to transform. For example, the statement

Transformer A method that changes the internal state of an object

```
ourDate.increment();
```

transforms the `ourDate` object.

Note that we have left out the details of the `increment` method since they are not crucial to our current discussion.

A program with access to both of the date classes can now declare and use both `Date` and `IncDate` objects. Consider the following program segment. (Assume `output` is one of Java's `PrintWriter` file objects.)

```
Date myDate = new Date(6, 24, 1951);
IncDate aDate = new IncDate(1, 11, 2001);

output.println("mydate day is:   " + myDate.dayIs());
output.println("aDate day is:    " + aDate.dayIs());

aDate.increment();
output.println("the day after is: " + aDate.dayIs());
```

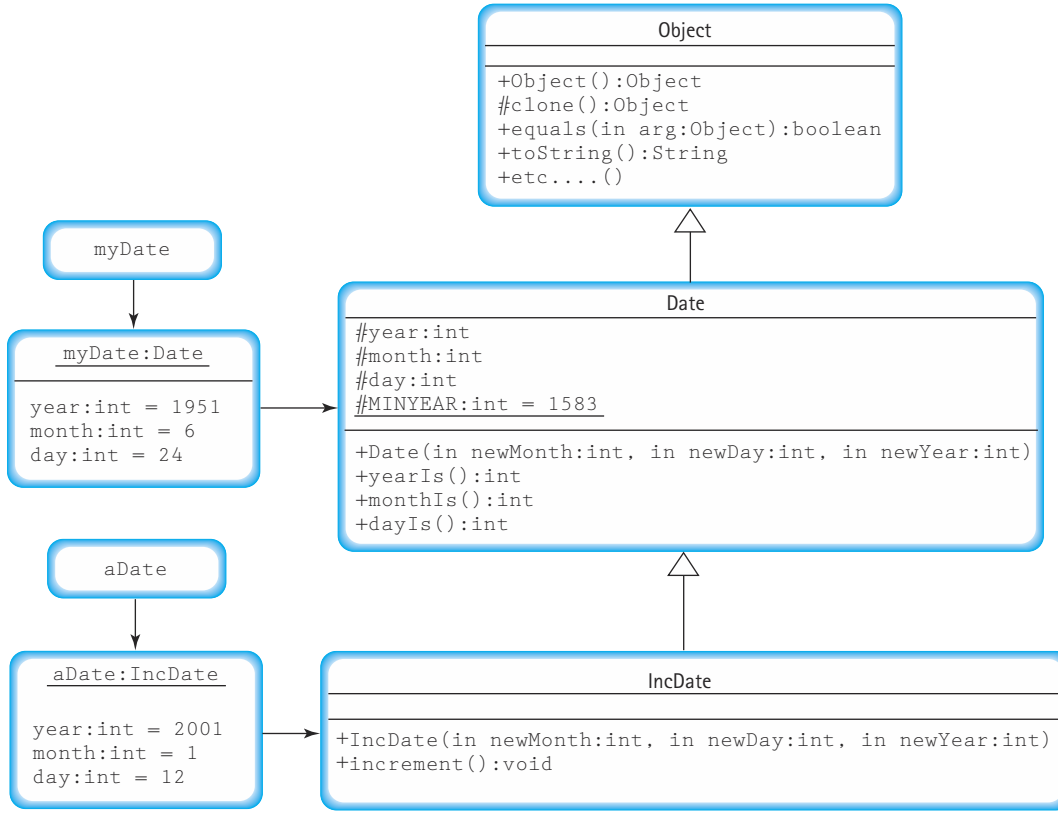


Figure 1.5 *Extended UML class diagram showing inheritance*

This program segment instantiates and initializes `myDate` and `aDate`, outputs the values of their days, increments `aDate` and finally outputs the new day value of `aDate`. You might ask, "How does the system resolve the use of the `dayIs` method by an `IncDate` object when `dayIs` is defined in the `Date` class?" Understanding how inheritance is supported by Java provides the answer to this question. The extended UML diagram in Figure 1.5 shows the inheritance relationships and captures the state of the system after the `aDate` object has been incremented. This figure helps us investigate the situation.

The compiler has available to it all the declaration information captured in the extended UML diagram. Consider the `dayIs` method call in the statement:

```
output.println("aDate day is:     " + aDate.dayIs());
```

To resolve this method call, the compiler follows the reference from the `aDate` variable to the `IncDate` class. Since it does not find a definition for a `dayIs` method in the `IncDate` class, it follows the inheritance link to the superclass `Date`, where it finds, and links to, the `dayIs` method. In this case, the `dayIs` method returns an `int` value that

represents the `day` value of the `aDate` object. During execution, the system changes the `int` value to a `String`, concatenates it to the string "aDate day is: " and prints it to `output`.

Note that because of the way method calls are resolved, by searching *up* the inheritance tree, only objects of the class `IncDate` can use the `increment` method. If you tried to use the `increment` method on an object of the class `Date`, such as the `myDate` object, there would be no definition available in either the `Date` class or any of the classes above `Date` in the inheritance tree. The compiler would report a syntax error in this situation.

Notice the `Object` class in the diagram. Where did it come from? In Java, any class that does not explicitly extend another class implicitly extends the predefined `Object` class. Since `Date` does not explicitly extend any other class, it inherits directly from `Object`. The `Date` class is a subclass of `Object`. The solid arrows with the hollow arrowheads indicate inheritance in a UML diagram.

All Java classes can trace their roots back to the `Object` class, which is so general that it does almost nothing; objects of the class `Object` are nearly useless by themselves. But `Object` does define several basic methods: comparison for equality (`equals`), conversion to a string (`toString`), and so on. Therefore, for example, any object in any Java program supports the method `toString`, since it is inherited from the `Object` class.

Just as Java automatically changes an integer value to a string in a statement like

```
output.println("aDate day is:    " + aDate.dayIs());
```

it automatically changes an object to a string in a statement like

```
output.println("tomorrow: " + aDate);
```

If you use an object as a string anywhere in a Java program, then the Java compiler automatically looks for a `toString` method for that object. In this case, the `toString` method is not found in the `IncDate` class, nor is it found in its superclass, the `Date` class. However, the compiler continues looking up the inheritance hierarchy, and finds the `toString` method in the `Object` class. Since all classes trace their roots back to `Object`, the compiler is always guaranteed to find a `toString` method eventually.

But, wait a minute. What does it mean to "change an object to a string"? Well, that depends on the definition of the `toString` method that is associated with the object. The `toString` method of the `Object` class returns a string representing some of the internal system implementation details about the object. This information is somewhat cryptic and generally not useful to us. This is an example of where it is useful to redefine an inherited method. We generally override the default `toString` method when creating our own classes, to return a more relevant string. For example, the following `toString` method could be added to the definition of the `Date` class:

```
public String toString()
{
  return(month + "/" + day + "/" + year);
}
```

Now, when the compiler needs a `toString` method for a `Date` object (or an `IncDate` object), it finds the method in the `Date` class and returns a more useful string. Figure 1.6 shows the output from the following program segment.

```
Date myDate = new Date(6, 24, 1951);
IncDate currDate = new IncDate(1, 11, 2001);

output.println("mydate:    " + myDate);
output.println("today:     " + currDate);

currDate.increment();
output.println("tomorrow: " + currDate);
```

The results on the left show the output generated if the `toString` method of the `Object` class is used by default; and on the right if the `toString` method above is added to the `Date` class:

Figure 1.6 *Output from program segment*

One last note: Remember that subclasses are assignment compatible with the superclasses above them in the inheritance hierarchy. Therefore, in our example, the statement

```
myDate = currDate;
```

would be legal, but the statement

```
currDate = myDate;
```

would cause an "incompatible type" syntax error.

Design

The *object-oriented design* (OOD) methodology originated with the development of programs to simulate physical objects and processes in the real world. For example, to simulate an electronic circuit, you could develop a class for simulating each kind of component in the circuit and then "wire-up" the simulation by having the modules pass information among themselves in the same pattern that wires connect the electronic components.

Identifying Classes The key task in designing object-oriented systems is identification of classes. Successful class identification and organization draws upon many of the tools that we discussed earlier in this chapter. Top-down stepwise refinement encourages us to start by identifying the major classes and gradually refine our system definition to identify all the classes we need. We should use abstraction and practice information hiding by keeping the interfaces to our classes narrow and hiding important design decisions and requirements likely to change within our classes. CRC cards can help us identify the responsibilities and collaborations of our classes, and expose holes in our design. UML diagrams let us record our designs in a form that is easy to understand.

When possible, we should organize our classes in an inheritance hierarchy, to benefit from reuse. Another form of reuse is to find prewritten classes, possibly in the standard Java library, that can be used in a solution.

There is no foolproof technique for identifying classes; we just have to start brainstorming ideas and see where they lead us. A large program is typically written by a team of programmers, so the brainstorming process often occurs in a team setting. Team members identify whatever objects they see in the problem and then propose classes to represent them. The proposed classes are all written on a board. None of the ideas for classes are discussed or rejected in this first stage.

After the brainstorming, the team goes through a process of filtering the classes. First they eliminate duplicates. Then they discuss whether each class really represents an object in the problem. (It's easy to get carried away and include classes, such as "the user," that are beyond the scope of the problem.) The team then looks for classes that seem to be related. Perhaps they aren't duplicates, but they have much in common, and so they are grouped together on the board. At the same time, the discussion may reveal some classes that were overlooked.

Usually it is not difficult to identify an initial set of classes. In most large problems we naturally find entities that we wish to represent as classes. For example, in designing a program that manages a checking account, we might identify checks, deposits, an account balance, and account statements as entities. These entities interact with each other through messages. For example, a check could send a message to the balance entity that tells it to deduct an amount from itself. We didn't list the amount in our initial set of objects, but it may be another entity that we need to represent.

Our example illustrates a common approach to OOD. We begin by identifying a set of objects that we think are important in a problem. Then we consider some scenarios in which the objects interact to accomplish a task. In the process of envisioning how a scenario plays out, we identify additional objects and messages. We keep trying new scenarios until we find that our set of objects and messages is sufficient to accomplish any task that the problem requires. CRC cards help us enact such scenarios.

A standard technique for identifying classes and their methods is to look for objects and operations in the problem statement. Objects are usually nouns and operations are usually verbs. For example, suppose the problem statement includes the sentence: "The student grades must be sorted from best to worst before being output." Potential objects are "student" and "grade," and potential operations are "sort" and "output." We propose

that on a printed copy of your requirements you circle the nouns and underline the verbs. The set of nouns are your candidate objects, and the verbs are your candidate methods. Of course, you have to filter this list, but at least it provides a good starting point for design.

Recall that in our discussion of abstraction and information hiding we stated that program modules should display strong cohesion. A good way to validate the cohesiveness of an identified class is to try to describe its main responsibility in a single coherent phrase. If you cannot do this, then you should reconsider your design. Some examples of cohesive responsibilities are:

- maintain a list of integers
- handle file interaction
- provide a date type

Some examples of "poor" responsibilities are:

- maintain a list of integers and provide special integer output routines
- handle file interaction and draw graphs on the screen

In summation, we have discussed the following approaches to identifying classes:

1. Start with the major classes and refine the design.
2. Hide important design decisions and requirements likely to change within a class.
3. Brainstorm with a group of programmers.
4. Make sure each class has one main responsibility.
5. Use CRC cards to organize classes and identify holes in the design.
6. Walk through user scenarios.
7. Look for nouns and verbs in the problem description.

Design Choices When working on design, keep in mind that there are many different correct solutions to most problems. The techniques we use may seem imprecise, especially in contrast with the precision that is demanded by the computer. But the computer merely demands that we express (code) a particular solution precisely. The process of deciding which particular solution to use is far less precise. It is our human ability to make choices without having complete information that enables us to solve problems. Different choices naturally lead to different solutions to a problem.

For example, in developing a simulation of an air traffic control system, we might decide that airplanes and control towers are objects that communicate with each other. Or we might decide that pilots and controllers are the objects that communicate. This choice affects how we subsequently view the problem, and the responsibilities that we assign to the objects. Either choice can lead to a working application. We may simply prefer the one with which we are most familiar.

Some of our choices lead to designs that are more or less efficient than others. For example, keeping a list of names in alphabetical rather than random order makes it possible for the computer to find a particular name much faster. However, choosing to leave the list randomly ordered still produces a valid (but slower) solution, and may even be the best solution if you do not need to search the list very often.

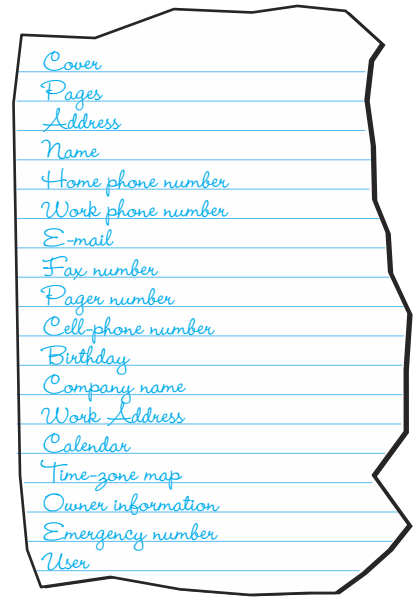
Other choices affect the amount of work that is required to develop the remainder of a problem solution. In creating a program for choreographing ballet movements, we might begin by recognizing a dancer as the important object and then create a class for each dancer. But in doing so, we discover that all of the dancers have certain common responsibilities. Rather than repeat the definition of those responsibilities for each class of dancer, we can change our initial choice and define a class for a generic dancer that includes all the common responsibilities and then develop subclasses that add responsibilities specific to each individual.

The point is, don't hesitate to begin solving a problem because you are waiting for some flash of genius that leads you to the perfect solution. There is no such thing. It is better to jump in and try something, step back, and see if you like the result, and then either proceed or make changes. In the example below we show how the CRC card technique helps you explore different design choices and keep track of them.

Design Example

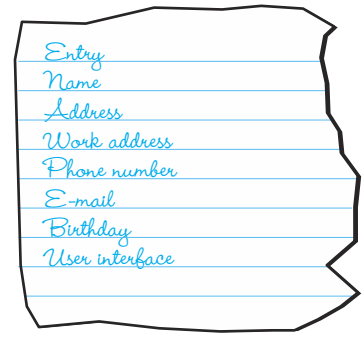
In this subsection we present a sample object-oriented design process that might be followed if we were on a small team of software engineers. Our purposes are to show the classes that might be identified for an object-oriented system, and to demonstrate the utility of CRC cards. We assume that our team of engineers has been given the task of automating an address book. A user should be able to enter and retrieve information from the address book. We have been given a sample physical address book on which to base their product.

First our team studies the problem, inspects the physical address book, and brainstorms that the application has the following potential objects:



Then we enter the filtering stage. Our application doesn't need to represent the physical parts of an address book, so we can delete Cover and Pages. However, we need something analogous to a page that holds all the same sort of information. Let's call it an Entry. The different telephone numbers can all be represented by the same kind of object. So we can combine Home, Work, Fax, Pager, and Cell-phone into a Phone number class. In consultation with the customer, we find that the electronic address book doesn't need the special pages that are often found in a printed address book, so we delete Calendar, Time-zone map, Owner information, and Emergency number.

Further thought reveals that the User isn't part of the application, although this does point to the need for a User interface that we did not originally list. A Work Address is a specific kind of address that has additional information, so we can make it a subclass of Address. Company names are just Strings, so there is no need to distinguish them, but Names have a first, last, and middle part. Our filtered list of classes now looks like this.



For each of these classes we create a CRC card. In the case of Work Address, we list Address as its Superclass, and on the Address card we list Work Address in its Subclasses space.

In doing coursework, you may be asked to work individually rather than in a collaborative team. You can still do your own brainstorming and filtering. However, we recommend that you take a break after the brainstorming and do the filtering once you have let your initial ideas rest for a while. An idea that seems brilliant in the middle of brainstorming may lose some of its attraction after a day or even a few hours.

Initial Responsibilities Once you (or your team) have identified the classes and created CRC cards for them, go over each card and write down its primary responsibility and an initial list of resultant responsibilities that are obvious. For example, a Name class manages a "Name" and has a responsibility to know its first name, its middle name, and its last name. We would list these three responsibilities in the left column of its card, as shown in Figure 1.7. In an implementation, they become methods that return the corresponding part of the name. For many classes, the initial responsibilities include knowing some value or set of values.

Class Name: Name	Superclass:	Subclassess:
Primary Responsibility: Manage a Name		
Responsibilities	Collaborations	
Know first		
Know middle		
Know last		

Figure 1.7 *A CRC card with initial responsibilities*

A First Scenario Walk-Through To further expand the responsibilities of the classes and see how they collaborate, we must pretend to carry out various processing scenarios by hand. This kind of role-playing is known as a *walk-through*. We ask a question such as, "What happens when the user wants to find an address that's in the book?" Then we answer the question by telling how each object is involved in accomplishing this task. In a team setting, the cards are distributed among the team members. When an object of a class is doing something, its card is held in the air to visually signify that it is active.

With this particular question, we might pick up the User Interface card and say, "I have a responsibility to get the person's name from the user." That responsibility gets written down on the card. Once the name is input, the User Interface must collaborate with other objects to look up the name and get the corresponding address. What object should it collaborate with? There is no identified object class that represents the entire set of address book entries.

We've found a hole in our list of classes! The Entry objects should be organized into a Book object. We quickly write out a Book CRC card. The User Interface card-holder then says, "I'm going to collaborate with the Book class to get the address." The collaboration is written in the right column of the card, and it remains in the air. The owner of the Book card holds it up, saying, "I have a responsibility to find an address in the list of Entry objects that I keep, given a name." That responsibility gets written on the

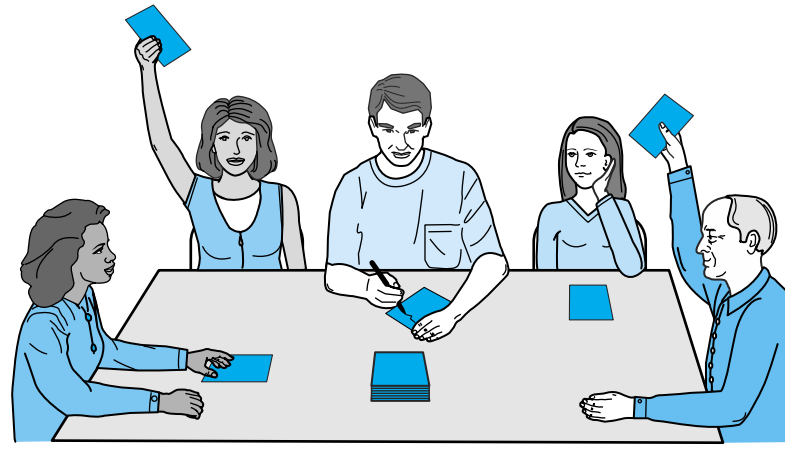


Figure 1.8 *A scenario walk-through in progress*

Book Card. Then the owner says, "I have to collaborate with each Entry to compare its name with the name sent to me by the User Interface." Figure 1.8 shows a team in the middle of a walk-through.

Now comes a decision. What are the responsibilities of Book and Entry for carrying out the comparison? Should Book get the name from Entry and do the comparison, or should it send the name to Entry and receive an answer that indicates whether they are equal? The team decides that Book should do the comparing, so the Entry card is held in the air, and its owner says, "I have a responsibility to provide the full name as a string. To do that I must collaborate with Name." The responsibility and collaboration are recorded and the Name card is raised.

Name says, "I have the responsibilities to know my first, middle, and last names. These are already on my card, so I'm done." And the Name card is lowered. Entry says, "I concatenate the three names into a string with spaces between them, and return the result to Book, so I'm done." The Entry card is lowered.

Book says, "I keep collaborating with Entry until I find the matching name. Then I must collaborate with Entry again to get the address." This collaboration is placed on its card and the Entry card is held up again, saying "I have a responsibility to provide an address. I'm not going to collaborate with Address, but am just going to return the object to Book." The Entry card has this responsibility added and then goes back on the table. Its CRC card is shown in Figure 1.9.

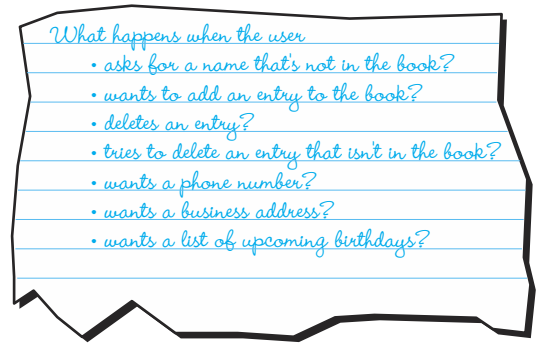
The scenario continues until the task of finding an address in the book and reporting it to the user is completed. Reading about the scenario makes it seem longer and more complex than it really is. Once you get used to role playing, the scenarios move quickly and the walk-through becomes more like a game. However, to keep things moving, it is important to avoid becoming bogged-down with implementation details. Book should not be concerned with how the Entry objects are organized on the list. Address doesn't need to think about whether the zip code is stored as an integer or a String.

Class Name: Entry	Superclass:	Subclassess:
Primary Responsibility: Manage a 'page' of information		
Responsibilities	Collaborations	
Provide name as a string	Get first from Name	
	Get middle from Name	
	Get last from Name	
Provide Address	None	

Figure 1.9 *The CRC card for Entry*

Only explore each responsibility far enough to decide whether a further collaboration is needed, or if it can be solved with the available information.

The next step is to brainstorm some additional questions that produce new scenarios. For example, here is list of some further scenarios.



We walk through each of the scenarios, adding responsibilities and collaborations to the CRC cards as necessary. After several scenarios have been tried, the number of

additions decreases. When one or more scenarios take place without adding to any of the cards, then we brainstorm further to see if we can come up with new scenarios that may not be covered. When all of the scenarios that we can envision seem to be doable with the existing classes, responsibilities, and collaborations, then the design is done.

The next step is to implement the responsibilities for each class. The implementation may reveal details of a collaboration that weren't obvious in the walk-through. But knowing the collaborating classes makes it easy to change their corresponding responsibilities. The implementation phase should also include a search of available class libraries to see if any existing classes can be used. For example, the `java.util.Calendar` class represents a date that can be used directly to implement Birthday.

Enhancing CRC Cards with Additional Information The CRC card design is informal. There are many ways that the card can be enhanced. For example, when a responsibility has obvious steps, we can write them below its name. Each step may have specific collaborations, and we write these beside the steps in the right column. We often recognize that certain data must be sent as part of the message that activates a responsibility, and we can record this in parentheses beside the calling collaboration and the responding responsibility. Figure 1.10 shows a CRC card that includes design information in addition to the basic responsibilities and collaborations.

To summarize the CRC card process, we brainstorm the objects in a problem and abstract them into classes. Then we filter the list of classes to eliminate duplicates. For each class, we create a CRC card and list any obvious responsibilities that it should support. We then walk through a common scenario, recording responsibilities and collaborations as they are discovered. After that we walk through additional scenarios, moving from common cases to special and exceptional cases. When it appears that we have all of the scenarios covered, we brainstorm additional scenarios that may need more responsibilities and collaborations. When our ideas for scenarios are exhausted, and all the scenarios are covered by the existing CRC cards, the design is done.

1.3 Verification of Software Correctness

At the beginning of this chapter, we discussed some characteristics of good programs. The first of these was that a good program works—it accomplishes its intended function. How do you know when your program meets that goal? The simple answer is, *test it*.

Testing The process of executing a program with data sets designed to discover errors

Let's look at testing as it relates to the rest of the software development process. As programmers, we first make sure that we understand the requirements, and then we come up with a general solution. Next we design the solution in terms of a system of classes, using good design principles, and finally we implement the solution, using well-structured code, with classes, comments, and so on.

Class Name: Entry	Superclass:	Subclassess:
Primary Responsibility: Manage a 'page' of information		
Responsibilities	**Collaborations**	
Provide name as a string		
Get first name	Name	
Get middle name	Name	
Get last name	Name	
Provide Address	None	
Change Name (name string)		
Break name into first, middle, last	String	
Update first name	Name, changeFirst(first)	
Update middle name	Name, changeMiddle(middle)	
Update last name	Name, changeLast(last)	

Figure 1.10 *A CRC card that is enhanced with additional information*

Once we have the program coded, we compile it repeatedly until the syntax errors are gone. Then we run the program, using carefully selected test data. If the program doesn't work, we say that it has a "bug" in it. We try to pinpoint the error and fix it, a process called debugging.

Debugging The process of removing known errors

Notice the distinction between testing and debugging. Testing is running the program with data sets designed to discover errors; debugging is removing errors once they are discovered.

When the debugging is completed, the software is put into use. Before final delivery, software is sometimes installed on one or more customer sites so that it can be tested in a real environment with real data. After passing this acceptance test phase, the software can be installed at all of the customer sites. Is the verification process now finished? Hardly! More than half of the total life-cycle costs and effort generally occur *after* the program becomes operational, in the maintenance phase. Some changes are made to correct errors in the original program; other changes are introduced to add new capabilities to the software system. In either case, testing must be done after any program modification. This is called regression testing.

Testing is useful for revealing the presence of bugs in a program, but it doesn't prove their absence. We can only say for sure that the program worked correctly for the cases we tested. This approach seems somewhat haphazard. How do we know which tests or how many of them to run? Debugging a whole program at once isn't easy. And fixing the errors found during such testing can sometimes be a messy task. Too bad we couldn't have detected the errors earlier—while we were designing the program, for instance. They would have been much easier to fix then.

Acceptance tests The process of testing the system in its real environment with real data

Regression testing Re-execution of program tests after modifications have been made in order to ensure that the program still works correctly

Program verification The process of determining the degree to which a software product fulfills its specifications

Program validation The process of determining the degree to which software fulfills its intended purpose

We know how program design can be improved by using a good design methodology. Is there something similar that we can do to improve our program verification activities? Yes, there is. Program verification activities don't need to start when the program is completely coded; they can be incorporated into the whole software development process, from the requirements phase on. Program verification is more than just testing.

In addition to program verification—fulfilling the requirement specifications—there is another important task for the software engineer: making sure the specified requirements actually solve the underlying problem. There have been countless times when a programmer finishes a large project and delivers the verified software, only to be told, "Well, that's what I asked for, but it's not what I need."

The process of determining that software accomplishes its intended task is called program validation. Program verification asks, "Are we doing the job right?" Program validation asks, "Are we doing the right job?"[5]

Can we really "debug" a program before it has ever been run—or even before it has been written? In this section, we review a number of topics related to satisfying the criterion "quality software works." The topics include:

- designing for correctness
- performing code and design walk-throughs and inspections
- using debugging methods
- choosing test goals and data
- writing test plans
- structured integration testing

[5]B. W. Boehm, *Software Engineering Economics* (Englewood Cliffs, N.J.: Prentice-Hall, 1981).

Origin of Bugs

When Sherlock Holmes goes off to solve a case, he doesn't start from scratch every time; he knows from experience all kinds of things that help him find solutions. Suppose Holmes finds a victim in a muddy field. He immediately looks for footprints in the mud, for he can tell from a footprint what kind of shoe made it. The first print he finds matches the shoes of the victim, so he keeps looking. Now he finds another, and from his vast knowledge of footprints, he can tell that it was made by a certain type of boot. He deduces that such a boot would be worn by a particular type of laborer, and from the size and depth of the print, he guesses the suspect's height and weight. Now, knowing something about the habits of laborers in this town, he guesses that at 6:30 P.M. the suspect might be found in Clancy's Pub.

In software verification we are often expected to play detective. Given certain clues, we have to find the bugs in programs. If we know what kinds of situations produce program errors, we are more likely to be able to detect and correct problems. We may even be able to step in and prevent many errors entirely, just as Sherlock Holmes sometimes intervenes in time to prevent a crime that is about to take place.

Let's look at some types of software errors that show up at various points in program development and testing and see how they might be avoided.

Specifications and Design Errors

What would happen if, shortly before you were supposed to turn in a major class assignment, you discovered that some details in the professor's program description were incorrect? To make matters worse, you also found out that the corrections were discussed at the beginning of class on the day you got there late, and somehow you never knew about the problem until your tests of the class data set came up with the wrong answers. What do you do now?

Writing a program to the wrong specifications is probably the worst kind of software error. How bad can it be? Most studies indicate that it costs 100 times as much to correct an error discovered after software delivery then it does if it is discovered early in the life cycle. Figure 1.11 shows how fast the costs rise in subsequent phases of software development. The vertical axis represents the relative cost of fixing an error; this cost

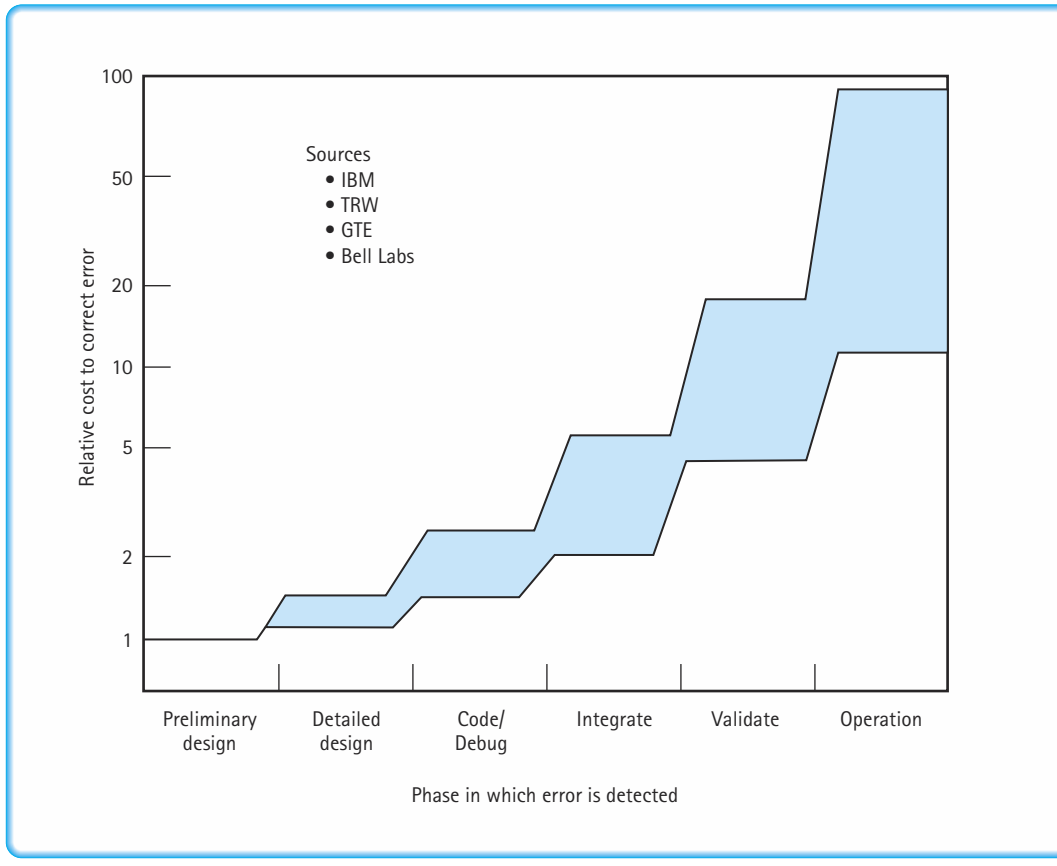


Figure 1.11 *Cost of a specification error based on when it is discovered*

might be in units of hours, or hundreds of dollars, or "programmer months" (the amount of work one programmer can do in a month). The horizontal axis represents the stages in the development of a software product. As you can see, an error that would have taken one unit to fix when you first started designing might take a hundred units to correct when the product is actually in operation!

Many specification errors can be prevented by good communication between the programmers (you) and the party who originated the problem (the professor, manager, or customer). In general, it pays to ask questions when you don't understand something in the program specifications. And the earlier you ask, the better.

A number of questions should come to mind as you first read a programming assignment. What error checking is necessary? What algorithm or data structure is supposed to be used in the solution? What assumptions are reasonable? If you obtain answers to these questions when you first begin working on an assignment, you can

incorporate them into your design and implementation of the program. Later in the program's development, unexpected answers to these questions can cost you time and effort. In short, in order to write a program that is correct, you must understand precisely what it is that your program is supposed to do.

Compile-Time Errors

In the process of learning your first programming language, you probably made a number of syntax errors. These resulted in error messages (for example, "TYPE MISMATCH," "ILLEGAL ASSIGNMENT," "SEMICOLON EXPECTED," and so on) when you tried to compile the program. Now that you are more familiar with the programming language, you can save your debugging skills for tracking down important logical errors. *Try to get the syntax right the first time.* Having your program compile cleanly on the first attempt is a reasonable goal. A syntax error wastes computing time and money, as well as programmer time, and it is preventable.

As you progress in your college career or move into a professional computing job, learning a new programming language is often the easiest part of a new software assignment. This does not mean, however, that the language is the least important part. In this book we discuss data structures and algorithms that we believe are language-independent. This means that they can be implemented in almost any general-purpose programming language. The success of the implementation, however, depends on a thorough understanding of the features of the programming language. What is considered acceptable programming practice in one language may be inadequate in another, and similar syntactic constructs may be just different enough to cause serious trouble.

It is, therefore, worthwhile to develop an expert knowledge of both the control and data constructs and the syntax of the language in which you are programming. In general, if you have a good knowledge of your programming language—and are careful—you can avoid syntax errors. The ones you might miss are relatively easy to locate and correct. Once you have a "clean" compilation, you can execute your program.

Run-Time Errors

Errors that occur during the execution of a program are usually harder to detect than syntax errors. Some run-time errors stop execution of the program. When this happens, we say that the program "crashed" or "abnormally terminated."

Run-time errors often occur when the programmer makes too many assumptions. For instance,

```
result = dividend / divisor;
```

is a legitimate assignment statement, if we can assume that `divisor` is never zero. If `divisor` is zero, however, a run-time error results.

Run-time errors also occur because of unanticipated user errors. If a user enters the wrong data type in response to a prompt, or supplies an invalid filename to a routine, most simple programs report a runtime error and halt; in other words, they crash.

Robustness The ability of a program to recover following an error; the ability of a program to continue to operate within its environment

Well-written programs should not crash. They should catch such errors and stay in control until the user is ready to quit.

The ability of a program to recover when an error occurs is called robustness. If a commercial program is not robust, people do not buy it. Who wants a word processor that crashes if the user says "SAVE" when there is no disk in the drive? We want the program to tell us, "Put your disk in the drive, and press Enter." For some types of software, robustness is a critical requirement. An airplane's automatic pilot system or an intensive care unit's patient-monitoring program just cannot afford to crash. In such situations, a defensive posture produces good results.

In general, you should actively check for error-creating conditions rather than let them abort your program. For instance, it is generally unwise to make too many assumptions about the correctness of input, especially interactive input from a keyboard. A better approach is to check explicitly for the correct type and bounds of such input. The programmer can then decide how an error should be handled (request new input, print a message, or go on to the next data) rather than leave the decision to the system. Even the decision to quit should be made by a program that is in control of its own execution. If worse comes to worst, let your program die gracefully.

This does not mean that everything that the program inputs must be checked for errors. Sometimes inputs are known to be correct—for instance, input from a file that has been verified. The decision to include error checking must be based upon the requirements of the program.

Some run-time errors do not stop execution but produce the wrong results. You may have incorrectly implemented an algorithm or initialized a variable to an incorrect value. You may have inadvertently swapped two parameters of the same type on a method call or used a less-than sign instead of a greater-than sign. These logical errors are often the hardest to prevent and locate. Later we talk about debugging techniques to help pinpoint run-time errors. We also discuss structured testing methods that isolate the part of the program being tested. But knowing that the earlier we find an error the easier it is to fix, we turn now to ways of catching run-time errors before run time.

Designing for Correctness

It would be nice if there were some tool that would locate the errors in our design or code without our even having to run the program. That sounds unlikely, but consider an analogy from geometry. We wouldn't try to prove the Pythagorean theorem by proving that it worked on every triangle; that would only demonstrate that the theorem works for every triangle we tried. We prove theorems in geometry mathematically. Why can't we do the same for computer programs?

The verification of program correctness, independent of data testing, is an important area of theoretical computer science research. The goal of this research is to establish a method for proving programs that is analogous to the method for proving theorems in geometry. The necessary techniques exist, but the proofs are often more complicated than the programs themselves. Therefore, a major focus of verification

research is to attempt to build automated program provers—verifiable programs that verify other programs. In the meantime, the formal verification techniques can be carried out by hand.[6]

Preconditions and Postconditions

Suppose we want to design a module (a logical chunk of the program) to perform a specific operation. To ensure that this module fits into the program as a whole, we must clarify what happens at its boundaries—what must be true when we enter the module and what is true when we exit.

To make the task more concrete, picture the design module as it is usually coded, as a method that is exported from a class. To be able to invoke the method, we must know its exact interface: the name and the parameter list, which indicates its inputs and outputs. But this isn't enough: We must also know any assumptions that must be true for the operation to function correctly.

We call the assumptions that must be true when invoking the method preconditions. The preconditions are like a product disclaimer:

Preconditions Assumptions that must be true on entry into an operation or method for the postconditions to be guaranteed

For example, the `increment` method of the `IncDate` class, described in the previous section, might have preconditions related to legal date values and the start of the Gregorian calendar. The preconditions should be listed with the method declaration:

```
public void increment()
// Preconditions: Values of day, month, and year represent a valid date
//                The represented date is not before minYear
```

Previously we discussed the quality of program robustness, the ability of a program to catch and recover from errors. While creating robust programs is an important goal,

---

[6] We do not go into this subject in detail here. If you are interested in this topic, you might start with David Gries' classic, *The Science of Programming* (NewYork: Springer-Verlag, (1981)).

it is sometimes necessary to decide at what level errors are caught and handled. Using preconditions for a method is similar to a contract between the programmer who creates the method and the programmers who use the method. The contract says that the programmer who creates the method is not going to try to catch the error conditions described by the preconditions, but as long as the preconditions are met, the method works correctly. It is up to the programmers who use the method to ensure that the method is never called without meeting the preconditions. In other words, the robustness of the system in terms of the method's preconditions is the responsibility of the programmers who use the class, and not the programmer who creates the class. This approach is sometimes called "programming by contract." It can save work because trapping the same error conditions at multiple levels of a hierarchical system is redundant and unnecessary.

Postconditions Statements that describe what results are to be expected at the exit of an operation or method, assuming that the preconditions are true

We must also know what conditions are true when the operation is complete. The postconditions are statements that describe the results of the operation. The postconditions do not tell us how these results are accomplished; they merely tell us what the results should be.

Let's consider what the preconditions and postconditions might be for another simple operation: a method that deletes the last element from a list. (We are using "list" in an intuitive sense; we formally define it in Chapter 3.) Assuming the method is defined within a class with the responsibility of maintaining a list, the specification for `RemoveLast` is as follows:

void RemoveLast()

Effect:	Removes the last element in this list.
Precondition:	This list is not empty.
Postcondition:	The last element has been removed from this list.

What do these preconditions and postconditions have to do with program verification? By making explicit statements about what is expected at the interfaces between modules, we can avoid making logical errors based on misunderstandings. For instance, from the precondition we know that we must check outside of this operation for the empty condition; this module *assumes* that there is at least one element.

Experienced software developers know that misunderstandings about interfaces to someone else's modules are one of the main sources of program problems. We use preconditions and postconditions at the method level in this book, because the information they provide helps us to design programs in a truly modular fashion. We can then use the classes we've designed in our programs, confident that we are not introducing errors by making mistakes about assumptions and about what the classes actually do.

Design Review Activities

When an individual programmer is designing and implementing a program, he or she can find many software errors with pencil and paper. Deskchecking the design solution is a very common method of manually verifying a program. The programmer writes down essential data (variables, input values, parameters, and so on) and walks through the design, marking changes in the data on the paper. Known trouble spots in the design or code should be double-checked. A checklist of typical errors (such as loops that do not terminate, variables that are used before they are initialized, and incorrect order of parameters on method calls) can be used to make the deskcheck more effective. A sample checklist for deskchecking a Java program appears in Figure 1.12. A few minutes spent deskchecking your designs can save lots of

Deskchecking Tracing an execution of a design or program on paper

The Design

1. Does each class in the design have a clear function or purpose?
2. Can large classes be broken down into smaller pieces?
3. Do multiple classes share common code? Is it possible to write more general classes to encapsulate the commonalities and then have the individual classes inherit from that general class?
4. Are all the assumptions valid? Are they well documented?
5. Are the preconditions and postconditions accurate assertions about what should be happening in the method they specify?
6. Is the design correct and complete as measured against the program specification? Are there any missing cases? Is there faulty logic?
7. Is the program designed well for understandability and maintainability?

The Code

1. Has the design been clearly and correctly implemented in the programming language? Are features of the programming language used appropriately?
2. Are methods coded to be consistent with the interfaces shown in the design?
3. Are the actual parameters on method calls consistent with the parameters declared in the method definition?
4. Is each data object to be initialized set correctly at the proper time? Is each data object set correctly before its value is used?
5. Do all loops terminate?
6. Is the design free of "magic" values? (A magic value is one whose meaning is not immediately evident to the reader. You should use constants in place of such values.)
7. Does each constant, class, variable, and method have a meaningful name? Are comments included with the declarations to clarify the use of the data objects?

Figure 1.12 *Checklist for deskchecking programs*

time and eliminate difficult problems that would otherwise surface later in the life cycle (or even worse, would not surface until after delivery).

Have you ever been really stuck trying to debug a program and showed it to a classmate or colleague who detected the bug right away? It is generally acknowledged that someone else can detect errors in a program better than the original author can. In an extension of deskchecking, two programmers can trade code listings and check each other's programs. Universities, however, frequently discourage students from examining each other's programs for fear that this exchange leads to cheating. Thus, many students become experienced in writing programs but don't have much opportunity to practice reading them.

Walk-through A verification method in which a *team* performs a manual simulation of the program or design

Inspection A verification method in which one member of a team reads the program or design line by line and the others point out errors

Most sizable computer programs are developed by *teams* of programmers. Two extensions of deskchecking that are effectively used by programming teams are design or code walk-throughs and inspections. These are formal team activities, the intention of which is to move the responsibility for uncovering bugs from the individual programmer to the group. Because testing is time-consuming and errors cost more the later they are discovered, the goal is to identify errors before testing begins.

In a *walk-through,* the team performs a manual simulation of the design or program with sample test inputs, keeping track of the program's data by hand on paper or a blackboard. Unlike thorough program testing, the walk-through is not intended to simulate all possible test cases. Instead, its purpose is to stimulate discussion about the way the programmer chose to design or implement the program's requirements.

At an *inspection*, a reader (never the program's author) goes through the requirements, design, or code line by line. The inspection participants are given the material in advance and are expected to have reviewed it carefully. During the inspection, the participants point out errors, which are recorded on an inspection report. Many of the errors have been noted by team members during their preinspection preparation. Other errors are uncovered just by the process of reading aloud. As with the walk-through, the chief benefit of the team meeting is the discussion that takes place among team members. This interaction among programmers, testers, and other team members can uncover many program errors long before the testing stage begins.

If you look back at Figure 1.11, you see that the cost of fixing an error is relatively inexpensive up through the coding phase. After that, the cost of fixing an error increases dramatically. Using the formal inspection process can clearly benefit a project.

Exception Associated with an unusual, often unpredictable event, detectable by software or hardware, that requires special processing. The event may or may not be erroneous.

Exceptions

At the design stage, you should plan how to handle exceptions in your program. Exceptions are just what the name implies: exceptional situations. They are situations that alter the flow of control of the program, usually resulting in a premature end to program execution. Working with exceptions begins at the design phase: What are the unusual situations that the program should recognize? Where in the program can the situations be detected? How should the situations be handled if they occur?

Where—indeed whether—an exception is detected depends on the language, the software package design, the design of the libraries being used, and the platform, that is, on the operating system and hardware. Where an exception *should* be detected depends on the type of exception, on the software package design, and on the platform. Where an exception *is* detected should be well documented in the relevant code segments.

An exception *may* be handled any place in the software hierarchy—from the place in the program module where the exception is first detected through the top level of the program. In Java, as in most programming languages, unhandled built-in exceptions carry the penalty of program termination. Where in an application an exception *should* be handled is a design decision; however, exceptions should be handled at a level that knows what the exception means.

An exception need not be fatal. For non-fatal exceptions, the thread of execution may continue. Although the thread of execution can continue from any point in the program, the execution should continue from the lowest level that can recover from the exception. When an error occurs, the program may fail unexpectedly. Some of the failure conditions may possibly be anticipated and some may not. All such errors must be detected and managed.

Exceptions can be written in any language. Java (along with some other languages) provides built-in mechanisms to manage exceptions. All exception mechanisms have three parts:

- Defining the exception
- Generating (raising) the exception
- Handling the exception

Once your exception plan is determined, Java gives you a clean way of implementing these three phases using the *try-catch* and *throw* statements. We cover these statements at the end of Chapter 2 after we have introduced some additional Java constructs.

Program Testing

Eventually, after all the design verification, deskchecking, and inspections have been completed, it is time to execute the code. At last, we are ready to start testing with the intention of finding any errors that may still remain.

The testing process is made up of a set of test cases that, taken together, allow us to assert that a program works correctly. We say "assert" rather than "prove" because testing does not generally provide a proof of program correctness.

The goal of each test case is to verify a particular program feature. For instance, we may design several test cases to demonstrate that the program correctly handles various classes of input errors. Or we may design cases to check the processing when a data structure (such as an array) is empty, or when it contains the maximum number of elements.

Within each test case, we must perform a series of component tasks:

- We determine inputs that demonstrate the goal of the test case.
- We determine the expected behavior of the program for the given input.
- We run the program and observe the resulting behavior.
- We compare the expected behavior and the actual behavior of the program. If they are the same, the test case is successful. If not, an error exists, either in the test case itself or in the program. In the latter case, we begin debugging.

Unit testing Testing a class or method by itself

For now we are talking about test cases at a class, or method, level. It's much easier to test and debug modules of a program one at a time, rather than trying to get the whole program solution to work all at once. Testing at this level is called unit testing.

How do we know what kinds of unit test cases are appropriate, and how many are needed? Determining the set of test cases that is sufficient to validate a unit of a program is in itself a difficult task. There are two approaches to specifying test cases: cases based on testing possible data inputs and cases based on testing aspects of the code itself.

Functional domain The set of valid input data for a program or method

Data Coverage

In those limited cases where the set of valid inputs, or the functional domain, is extremely small, one can verify a program unit by testing it against every possible input element. This approach, known as *exhaustive testing*, can prove conclusively that the software meets its specifications. For instance, the functional domain of the following method consists of the values `true` and `false`.

```
public void PrintBoolean(boolean boolValue)
// Prints the Boolean value to the output
{
  if (boolValue)
    output.println("true");
  else
    output.println("false");
}
```

It makes sense to apply exhaustive testing to this method, because there are only two possible input values. In most cases, however, the functional domain is very large, so exhaustive testing is almost always impractical or impossible. What is the functional domain of the following method?

```
public void PrintInteger(int intValue)
// Prints the integer value intValue to the output
{
  output.println(intValue);
}
```

It is not practical to test this method by running it with every possible data input; the number of elements in the set of `int` values is clearly too large. In such cases, we do not attempt exhaustive testing. Instead, we pick some other measurement as a testing goal.

You can attempt program testing in a haphazard way, entering data randomly until you cause the program to fail. Guessing doesn't hurt, but it may not help much either. This

approach is likely to uncover some bugs in a program, but it is very unlikely to find them all. Fortunately, however, there are strategies for detecting errors in a systematic way.

One goal-oriented approach is to cover general classes of data. You should test at least one example of each category of inputs, as well as boundaries and other special cases. For instance, in method `PrintInteger` there are three basic classes of `int` data: negative values, zero, and positive values. So, you should plan three test cases, one for each of these classes. You could try more than three, of course. For example, you might want to try `Integer.MAX_VALUE` and `Integer.MIN_VALUE`, but because all the program does is print the value of its input, the additional test cases don't accomplish much.

There are other cases of data coverage. For example, if the input consists of commands, you must test each command and varying sequences of commands. If the input is a fixed-sized array containing a variable number of values, you should test the maximum number of values; this is the boundary condition. A way to test for robustness is to try one more than the maximum number of values. It is also a good idea to try an array in which no values have been stored or one that contains a single element. Testing based on data coverage is called black-box testing. The tester must know the external interface to the module—its inputs and expected outputs—but does not need to consider what is being done inside the module (the inside of the black box). (See Figure 1.13)

Black-box testing Testing a program or method based on the possible input values, treating the code as a "black box"

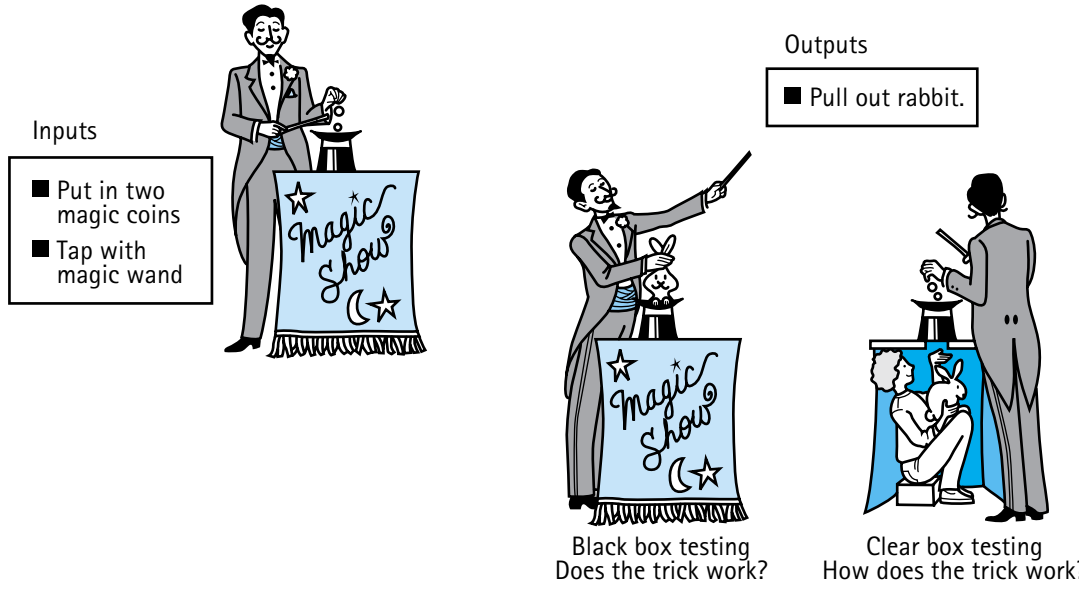


Figure 1.13 *Testing approaches*

Code Coverage

A number of testing strategies are based on the concept of code coverage, the execution of statements or groups of statements in the program. This testing approach is called clear (or white) box testing. The tester must look inside the module (through the clear box) to see the code that is being tested.

Clear (white) box testing Testing a program or method based on covering all of the branches or paths of the code

Branch A code segment that is not always executed; for example, a switch statement has as many branches as there are case labels

Path A combination of branches that might be traversed when a program or method is executed

Path testing A testing technique whereby the tester tries to execute all possible paths in a program or method

One approach, called *statement coverage*, requires that every statement in the program be executed at least once. Another approach requires that the test cases cause every branch, or code section, in the program to be executed. A single test case can achieve statement coverage of an *if-then* statement, but it takes two test cases to test both branches of the statement.

A similar type of code-coverage goal is to test program paths. A path is a combination of branches that might be traveled when the program is executed. In path testing, we try to execute all the possible program paths in different test cases.

Test Plans

Deciding on the goal of the test approach—data coverage, code coverage, or (most often) a mixture of the two, precedes the development of a test plan. Some test plans are very informal—the goal and a list of test cases, written by hand on a piece of paper. Even this type of test plan may be more than you have ever been required to write for a class programming project. Other test plans (particularly those submitted to management or to a customer for approval) are very formal, containing the details of each test case in a standardized format.

Test plan A document showing the test cases planned for a program or module, their purposes, inputs, expected outputs, and criteria for success

For program testing to be effective, *it must be planned*. You must design your testing in an organized way, and you must put your design in writing. You should determine the required or desired level of testing, and plan your general strategy and test cases before testing begins. In fact, you should start planning for testing before writing a single line of code.

Debugging

In the previous section we talked about checking the output from our test and debugging when errors were detected. We can debug "on the fly" by adding output statements in suspected trouble spots when problems are found. For example, if you suspect an error in the `IncDate increment` method, you could augment the method as follows:

```
public void increment()
{
  // For debugging
  output.println("IncDate method increment entered.");
  output.println("year = " + year);
  output.println("month = " + month);
  output.println("day = " + day);

  // Increment algorithm goes here
  // It updates the year, month, and day values

  // For debugging
  output.println("IncDate method increment exiting.");
  output.println("year = " + year);
  output.println("month = " + month);
  output.println("day = " + day);
  output.println("IncDate method increment terminated.");
}
```

Note that the new output is only for debugging; these output lines are meant to be seen only by the tester, not by the user of the program. But it's annoying for debugging output to show up mixed with your application's real output, and it's difficult to debug when the debugging output isn't collected in one place. One way to separate the debugging output from the "real" program output is to declare a separate file to receive these debugging lines.

Usually the debugging output statements are removed from the program, or "commented out," before the program is delivered to the customer or turned in to the professor. (To "comment out" means to turn the statements into comments by preceding them with `//` or enclosing them between `/*` and `*/`.) An advantage of turning the debugging statements into comments is that you can easily and selectively turn them back on for later tests. A disadvantage of this technique is that editing is required throughout the program to change from the testing mode (with debugging) to the operational mode (without debugging).

Another popular technique is to make the debugging output statements dependent on a Boolean flag, which can be turned on or off as desired. For instance, a section of code known to be error-prone may be flagged in various spots for trace output by using the Boolean value `debugFlag`:

```
// Set debugFlag to control debugging mode
static boolean debugFlag = true;
.
.
.
if (debugFlag)
  debugOutput.println("method Complex entered.");
```

This flag may be turned on or off by assignment, depending on the programmer's need. Changing to an operational mode (without debugging output) merely involves redefining `debugFlag` as `false` and then recompiling the program. If a flag is used, the debugging statements can be left in the program; only the *if* checks are executed in an operational run of the program. The disadvantage of this technique is that the code for the debugging is always there, making the compiled program larger and slower. If there are a lot of debugging statements, they may waste needed space and time in a large program. The debugging statements can also clutter up the program, making it harder to read. (This is another example of the tradeoffs we face in developing software.)

Some systems have online debugging programs that provide trace outputs, making the debugging process much simpler. If the system at your school or workplace has a run-time debugger, use it! Any tool that makes the task easier should be welcome, but remember that no tool replaces thinking.

A warning about debugging: Beware of the quick fix! Program bugs often travel in swarms, so when you find a bug, don't be too quick to fix it and run your program again. As often as not, fixing one bug generates another. A superficial guess about the cause of a program error usually does not produce a complete solution. In general, the time that it takes to consider all the ramifications of the changes you are making is time well spent.

If you constantly need to debug, there's a deficiency in your design process. The time that it takes to consider all the ramifications of the design you are making is time spent best of all.

Testing Java Data Structures

The major topic of this textbook is data structures: what they are, how we use them, and how we implement them using Java. This chapter has been an overview of software engineering. In Chapter 2 we begin our concentration on data and how to structure it. It seems appropriate to end this section about verification with a look at how we test the data structures we implement in Java.

In Chapter 2, we implement a data structure using a Java class, so that many different application programs can use the structure. When we first create the class that models the data structure, we do not necessarily have any application programs ready to use it. We need to test it by itself first, before creating the applications.

Every data structure that we implement supports a set of operations. For each structure, we would like to create a test driver program that allows us to test the operations in a variety of sequences. How can we write a single test driver that allows us to test numerous operation sequences? The solution is to separate the specific set of operations that we want to test from the test driver program itself. We list the operations, and the necessary parameters, in a text file. The test driver program reads the operations from the text file one line at a time, performs the listed operation by invoking the methods of the class being tested, and reports the results to an output file. The test program also reports its general results on the screen.

The testing approach described here allows us to easily change our test case—we just have to change the contents of the input file. However, it would be even easier if we could dynamically change the name of the input file, whenever we run the program. Then we could organize our test cases, one per file, and easily rerun a test case whenever we needed. Therefore, we construct our test driver to accept the name of the input file as a command line parameter; we do the same for the output file. Figure 1.14 displays a model of our test architecture.

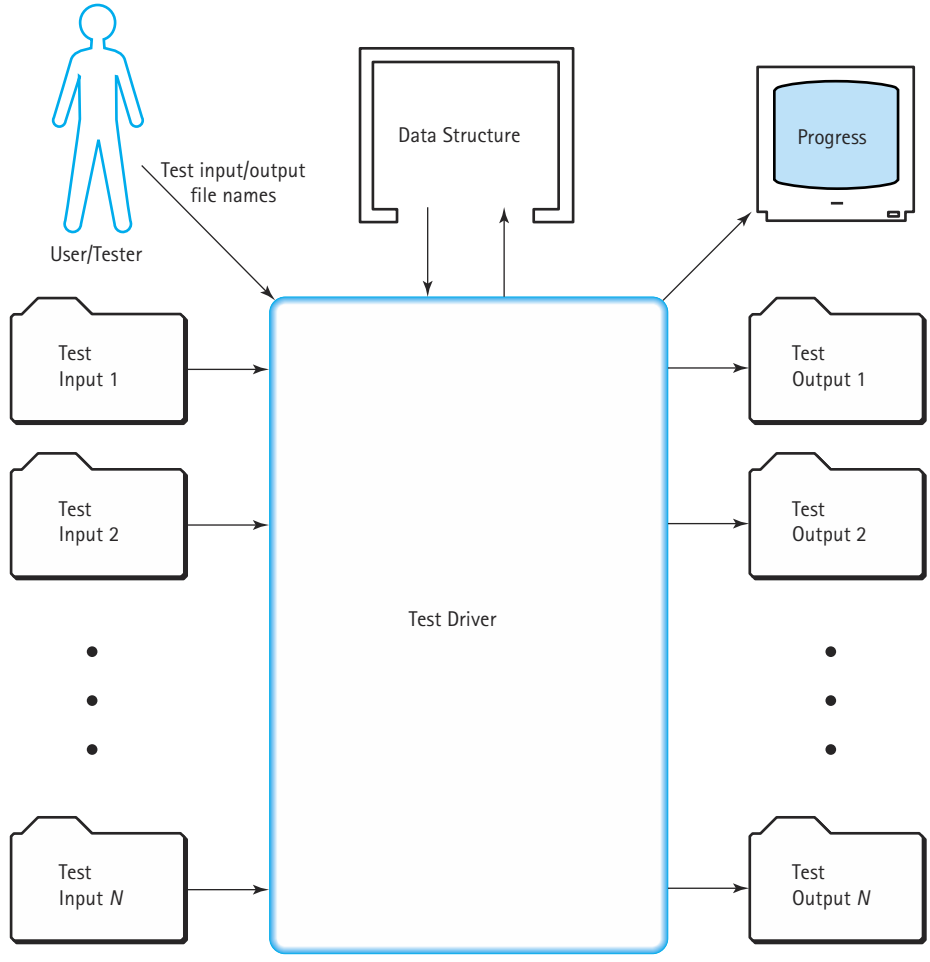


Figure 1.14 *Model of test architecture*

Our test drivers all follow the same basic algorithm; here is a pseudocode description:

```
Obtain the names of the input and output files from the command line
Open the input file for reading and the output file for writing
Read the first line from the input file
Print "Results " plus the first line of the input file to the output file
Print a blank line to the output file
Read a command line from the input file
Set numCommands to 0
While the command read is not 'quit'
    Execute the command by invoking the public methods of the data structure
    Print the results to the output file
    Print the data structure to the output file (if appropriate)
    Increment numCommands by 1
    Read the next command from the input file
Close the input and output files.
Print "Command " + numCommands + " completed" to the screen
Print "Testing completed" to the screen
```

This algorithm provides us with maximum flexibility for minimum extra work when we are testing our data structures. Once we implement the algorithm by creating a test driver for a specific data structure, we can easily create a test driver for a different data structure by changing only three steps.

Notice that the third and fourth commands copy a "header line" from the input test file to the output file. This helps us manage our test cases by allowing us to label each test case file with an identifying string on its first line; the same string always begins the corresponding output file.

Suppose we want to test the `IncDate` class that was defined earlier in this chapter. We first create a test plan. Let's use a goal-oriented approach. We first test the constructor and each of the observer methods. Next we test the transformer method `increment`. To test `increment` we identify general categories of dates, with respect to the effect of the `increment` method. We test dates that represent each of these categories, with special attention given to the boundaries of the categories. Thus, we test some dates in the middle of months, and at the beginning and end of months. We test the end of years also. We pay careful attention to testing how the method handles leap years, by including tests concentrated at the end of February in many different years. Several more test cases, besides those listed below, would be needed to ensure that the `increment` method works correctly.

Operation to be Tested and Description of Action	Input Values	Expected Output
Constructor		
`IncDate`	5, 6, 2000	
print		5/6/2000
Observers		
print `monthIs`		5
print `dayIs`		6
print `yearIs`		2000
Transformer		
`increment` and print		5/7/2000
`IncDate`	5,30,2000	
`increment` and print		5/31/2000
`IncDate`	5,31,2000	
`increment` and print		6/1/2000
`IncDate`	6,30,2000	
`increment` and print		7/1/2000
`IncDate`	2,28,2002	
`increment` and print		3/1/2002
etc.		

After identifying a test plan, we create a test driver using our algorithm. Then we use the test driver to carry out our plan. The `IncDate` class supports five operations: `IncDate` (the constructor), `yearIs`, `monthIs`, `dayIs`, and `increment`. We represent these operations in the test input file simply by using their names. In that file, the word `IncDate` is followed by three lines, each containing an integer, to supply the three *int* parameters of the constructor. Figure 1.15 shows an example of a test input file, the resulting output file, and the screen information that would be generated.

Study the test driver program on page 51 to make sure you understand our testing approach. You should be able to follow the control logic of the program. Note that we assume the inclusion of a reasonable `toString` method in the `Date` class, as described at the end of the Object-Oriented Design section. (The `Date.java` file on our web site includes a `toString` method.)

```
IncDate Test Data A
IncDate
5
6
2000
monthIs
dayIs
increment
dayIs
quit
```

File: TestDataA

```
Results IncDate Test Data A

Constructor invoked with 5 6 2000
theDate: 5/6/2000
Month is 5
theDate: 5/6/2000
Day is 6
theDate: 5/6/2000
increment invoked
theDate: 5/7/2000
Day is 7
theDate: 5/7/2000
```

File: TestOutputA

Screen

```
Command: java TDIncDate TestDataA TestOutputA
```

Figure 1.15 *Example of a test input file and resulting output file*

We realize that the students using this textbook come from a wide variety of Java backgrounds, especially with respect to the Java I/O approach. You may have learned Java in an environment where the Java input/output statements were "hidden" behind a package provided with your introductory textbook. Or you may have learned graphical input/output techniques, but never learned how to do file input/output. You may not be familiar with "command-line parameters;" or you might have been using command-line parameters since the first week you studied Java. You may have learned how to use the Java AWT; you may have learned Swing; you may have learned neither. Our approach to testing requires only simple file input and output, in addition to screen output. It does not require any direct user input during execution, which can be complicated in Java.

The feature section on Java Input/Output (after the following code) introduces the input/output techniques used for our test drivers. We use these same techniques in test drivers and example programs throughout the rest of the text, so it is a good idea for you to study them carefully now. The only places in the text where more advanced I/O approaches are used are in the chapter Case Studies. Beginning with Chapter 3, we develop case studies as examples of real programs that use the data structures you are studying. These case studies use progressively more advanced graphical interfaces, and are accompanied by additional feature sections as needed to explain any new constructs.

Therefore, the case studies not only provide examples of object-oriented design and uses of data structures, but they also progressively introduce you to user interface techniques.

Within the following test driver code we have emphasized, with underlining, all the commands related to input/output. As you can see, these statements make up a large percentage of the program; this is not unusual.

```
//--------------------------------------------------------------------------
// TDIncDate.java             by Dale/Joyce/Weems                  Chapter 1
//
// Test Driver for the IncDate class
//--------------------------------------------------------------------------

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import IncDate.*;

// Test Driver for the IncDate class
public class TDIncDate
{
  public static void main(String[] args) throws IOException
  {
    String testName  = "IncDate";
    String command   = null;
    int numCommands  = 0;
    IncDate  theDate = new IncDate(0,0,0);
    int month, day, year;

    //Get file name arguments from command line as entered by user
    String dataFileName = args[0];
    String outFileName  = args[1];

    //Prepare files
    BufferedReader dataFile = new BufferedReader(new FileReader(dataFileName));
    PrintWriter outFile     = new PrintWriter(new FileWriter(outFileName));

    //Get test file header line and echo print to outFile
    String testInfo = dataFile.readLine();
    outFile.println("Results " + testInfo);
    outFile.println();
    command = dataFile.readLine();

    //Process commands
    while(!command.equals("quit"))
```

```
{
  if (command.equals("IncDate"))
  {
    month  = Integer.parseInt(dataFile.readLine());
    day    = Integer.parseInt(dataFile.readLine());
    year   = Integer.parseInt(dataFile.readLine());
    outFile.println("Constructor invoked with " + month + " "
                    + day + " " + year);
    theDate = new IncDate(month, day, year);
  }
  else if (command.equals("yearIs"))

  {
    outFile.println("Year is " + theDate.yearIs());
  }
  else if (command.equals("monthIs"))

  {
    outFile.println("Month is " + theDate.monthIs());
  }
  else if (command.equals("dayIs"))

  {
    outFile.println("Day is " + theDate.dayIs());
  }
  else if (command.equals("increment"))

  {
    theDate.increment();
    outFile.println("increment invoked ");
  }

  outFile.println("theDate: " + theDate);
  numCommands++;
  command = dataFile.readLine();
}

//Close files
dataFile.close();
outFile.close();

//Set up output frame
JFrame outputFrame = new JFrame();
outputFrame.setTitle("Testing " + testName);
outputFrame.setSize(300,100);
outputFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
    // Instantiate content pane and information panel
    Container contentPane = outputFrame.getContentPane();
    JPanel infoPanel      = new JPanel();

    // Set layout
    infoPanel.setLayout(new GridLayout(2,1));

    // Create labels
    JLabel countInfo = new JLabel(numCommands + " commands completed.  ");
    JLabel finishedInfo = new JLabel("Testing completed. "
                                     + "Close window to exit program.");

    // Add information
    infoPanel.add(countInfo);
    infoPanel.add(finishedInfo);
    contentPane.add(infoPanel);

    // Show information
    outputFrame.show();
  }
}
```

Note that the test driver gets the test data and calls the methods to be tested. It also provides written output about the effects of the method calls, so that the tester can check the results. Sometimes test drivers are used to test hundreds or thousands of test cases. In such situations it is best if the test driver automatically verifies whether or not the test cases were handled successfully. Exercise 36 asks you to expand this test driver to include automatic test-case verification.

This test driver does not do any error checking to make sure that the inputs are valid. For instance, it doesn't verify that the input command code is really a legal command. Furthermore, it does not handle possible I/O exceptions; instead it just throws them out to the run-time environment (exception handling is discussed in Chapter 2). Remember that the goal of the test driver is to act as a skeleton of the real program, not to be the real program. Therefore, the test driver does not need to be as robust as the program it simulates.

Java Input/Output I

The Java class libraries provide varied and robust mechanisms for input and output. Hundreds of classes related to the user interface provide programmers with a multitude of options. I/O is not the topic of this textbook. We use straightforward I/O approaches that support the study of data structures.

In this feature section, we examine the I/O commands used in the `TDIncDate` program (we examine more I/O commands as needed later in the text). The relevant commands are highlighted

in the program text. As modeled in Figure 1.14, this program uses screen output and file input and output. The program also uses command-line arguments to obtain the names of the files—this is a form of input. Figure 1.15 shows an example of an input file, the resultant output file, the screen output, and the corresponding command line. If you're interested in learning more, you might begin by studying the documentation provided on the Sun Microsystems Inc. web site of the various classes and methods we use.

Command-Line Input

A simple way to pass string information to a Java program is with command-line arguments. Command-line arguments are read by the program each time it is run; a different set of arguments will invoke different behavior from the program. For example, suppose you want to run the `TDIncDate` program using a file called `TestDataA` as the input file and a file called `TestOutputA` as the output file. If you are working from the command line, you invoke the Java interpreter, asking it to "execute" the `TDIncDate.class` file using as arguments the strings "TestDataA" and "TestOutputA" by entering:

```
java TDIncDate TestDataA TestOutputA
```

The program runs; it takes its input from the `TestDataA` file; a small output window appears on your screen informing you when the program is finished; and the `TestOutputA` file holds the results of the test. You end the program by closing the output window. Now, if you want the program to run again using different input and output files, say, `TestDataB` and `TestOutputB`, you simply invoke the interpreter with a different command line:

```
java TDIncDate TestDataB TestOutputB
```

Note that if you are using an integrated development environment, instead of working from the command line, you compile and run your program using a pull-down menu or a shortcut key. Consult your environment's documentation to learn how to pass command-line arguments in this situation.

How do you access the command-line arguments within your program? Through the `main` method's array of strings parameter. By convention, this parameter is usually called `args`, to represent the command-line arguments. In our example, `args[0]` references the string "TestDataA" and `args[1]` references the string "TestOutputA". We use these string values to initialize string variables that represent the input and output files of the program:

```
String dataFileName = args[0];
String outFileName = args[1];
```

With this approach, we can change the test input and output files each time we run the program by simply entering a different command on the command line.

File Output

Java provides a stream output model. As an abstract concept, a stream is just a sequence of bytes. A Java program can direct an output stream to a file, a network connection, or even a specific block of memory. We use files.

The Java class library supports more than 60 different stream types. We use classes that inherit from the abstract class `Writer`. Abstract classes are discussed in Chapter 3. For now, all you need to know is that you cannot instantiate objects of abstract classes, but you can extend the classes. In our program we use the `PrintWriter` class and the `FileWriter` class, both of which are library subclasses of `Writer`. To make these classes available within our program, we must include the *import* statement:

```
import java.io.*;
```

The `Writer` class and its subclasses allow us to perform text output in a standard environment. You may recall from your previous studies that Java uses the Unicode character set as its base character set. A Unicode character uses 16 bits; therefore, the Unicode character set can represent 65,536 unique characters. This large character set helps make Java suitable as a programming language around the world, since there are many languages that do not use the standard Western alphabet. However, most of our environments do not yet support the Unicode character set. For example, text files, which we often use to provide input to a program or output from a program, are based on the much smaller ASCII character set. The `Writer` class provides methods to translate the Unicode characters used within a Java program to the ASCII characters required by text files.

To perform stream output using ASCII characters, we instantiate an object of the class `PrintWriter`. The `PrintWriter` class provides methods for printing all of Java's primitive types, strings, generic objects (using the object's `toString` method), and arrays of characters. It also provides a method to close the output stream (`close`), methods to check and set errors (`checkError` and `setError`), and a method to flush the stream (`flush`). The `flush` method is used to force all of the current output to go immediately to the file. In `TDIncDate` we only use `PrintWriter`, `println`, and `close` methods. The `println` method sends a textual representation of its parameter to the output stream, followed by a linefeed. For example, the code:

```
outFile.println("Month is " + theDate.monthIs());
```

transforms the `int` returned by the `monthIs` method into a string, concatenates that string to the string "Month is", transforms the entire string into an ASCII representation, appends a linefeed character, and sends the whole thing to the output stream. You can see many other uses of the `println` method throughout the rest of the program. The `close` method is invoked when processing is finished:

```
outFile.close();
```

Invoking `close` informs the system that we are finished using the file. It is important for system efficiency and stability for a program to close files when it is finished using them.

So far in this discussion, we have referred to sending textual information to the "output stream." But how is this output stream associated with the correct file? The answer to this question is found by looking at the declaration of the `PrintWriter` object used in the program:

```
PrintWriter outFile = new PrintWriter(new FileWriter(outFileName));
```

Embedded within the `PrintWriter` declaration is an invocation of a `FileWriter` constructor:

```
new FileWriter(outFileName)
```

The `FileWriter` class is another subclass of `Writer`. The code invokes the `FileWriter` constructor and instantiates an object of the class `Writer` that is associated with the file represented by the variable `outFileName`. Recall that `outFileName` is the name of the output file that was passed to the program as a command-line argument. By embedding this code within the `PrintWriter` declaration, we associate the `PrintWriter` object `outFile` with the text file represented by `outFileName`. In our example above this is the `OutFileA` file. Therefore, a command such as:

```
outFile.println("Month is " + theDate.monthIs());
```

sends its output to the `OutFileA` file.

File Input

Most of the previous discussion about file output can be applied to file input. Instead of using the abstract class `Writer` we use the abstract class `Reader`; instead of `PrintWriter` we use `BufferedReader`; instead of the `println` method we use the `readLine` method; instead of the `FileWriter` class we use the `FileReader` class. We leave it to the reader to look over the `TDIncDate` program to see how the various file reading statements interact with each other. We do, however, briefly discuss the `readLine` method.

The `BufferedReader readLine` method returns a string that holds the next line of characters from the input stream. Therefore, a statement such as:

```
command = dataFile.readLine();
```

sets the string variable `command` to reference the next line of characters from the file associated with the object `dataFile`. In some cases we need to transform this line of characters into an integer. To do this we use the `parseInt` method of the `Integer` wrapper class:

```
day = Integer.parseInt(dataFile.readLine());
```

An alternate approach is to use the `intValue` method of the `String` class, and the `valueOf` method of the `Integer` wrapper class as follows:

```
day = Integer.valueOf(dataFile.readLine()).intValue;
```

Wrapper classes are discussed in Chapter 2.

Frame Output

We really cannot do justice to the topic of graphical user interfaces (GUIs) in this textbook. The topic is a nontrivial, important area of computing and deserves serious study. Nevertheless, modern programming approaches demand the use of GUIs and we make moderate use of them in our programs. So, without trying to explain all of the underlying concepts and supporting classes, we look at the purpose of each of the statements related to frame output. (Figure 1.15 shows the displayed frame.)

Note that our `TDIncDate` class includes the following *import* statements:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

The first statement imports classes from the Java library `awt` package; the second imports classes related to event handling, also from the Java library `awt` package; the third imports the classes of the Java `swing` package. The AWT (Abstract Window Toolkit) was the set of graphical interface tools included with the original version of Java. Developers found that this set of tools was too limited for professional program development, so the Java designers included a new set of graphical components, called the "Swing" components, when they released the Java Foundation Classes in 1997. The Swing components are more portable and flexible than their AWT counterparts. We use Java Swing components throughout the text. Note that Java Swing is built on top of Java AWT, so we still need to import AWT classes.

The code related to the frame output begins with the comment:

```
//Set up output frame
```

and continues to the end of the program listing. First, let's address the set-up of the frame itself. A frame is a top-level window with a title, a border, a menu bar, a content pane, and more. We declare our frame with the statement:

```
JFrame outputFrame = new JFrame();
```

`JFrame` is the Java Swing frame component (you can recognize Java Swing components since they begin with the letter "J" to differentiate them from their AWT counterparts). Therefore, our `outputFrame` object is a `JFrame`, and can be manipulated with the library methods defined for `JFrames`.

We immediately make use of three of these methods to set up our frame:

```
outputFrame.setTitle("Testing " + testName);
outputFrame.setSize(300,100);
outputFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

These statements set the title and size for the instantiated frame, and define how the frame should react to the user closing the frame's window. Setting the title and size are very straightforward. The title of our frame is "Testing IncDate," since the variable `testname` was set to "IncDate" at the beginning of the main method. The size of the frame is set to 300 pixels wide by 100 pixels tall.

Defining how the frame reacts to the user closing the frame's window is a little more complicated. When the frame is eventually displayed, it appears in its own window. Normally, when you define a window from within a Java program, you must define how the window reacts to various events: closing the window, resizing the window, activating the window, and so on. You must define methods to handle all of these events. However, in our program we want to handle only one of these events, the window-closing event. Java provides a special method, just for handling this event; the `setDefaultCloseOperation` method. This method tells the `JFrame` what to do when its window is closed, as long the action is one of a small set of common choices. The `JFrame` class provides the following class constants that name these choices:

```
JFrame.DISPOSE_ON_CLOSE
JFrame.DO_NOTHING_ON_CLOSE
JFrame.HIDE_ON_CLOSE
JFrame.EXIT_ON_CLOSE
```

In our program we use the `EXIT_ON_CLOSE` option, so the program disposes of the window and exits when the user closes the window.

The following two lines set up our frame output:

```
Container contentPane = outputFrame.getContentPane();
JPanel infoPanel      = new JPanel();
```

The first line provides us a "handle" for the content pane of the new frame. Remember that frames have many parts; the part where we display information is called the "content pane." We now have access to the content pane of our frame through the `contentPane` variable. This variable is an object of the class `Container`, which means we can place other objects into it for display purposes. What can we place into it? We can place almost anything: buttons, labels, drawings, text boxes; but to help us organize our interfaces we prefer to place yet another container object, called a panel, into content panes. The second line instantiates a `JPanel` object (the Swing version of a panel) called `infoPanel`. It is here where we place the information we want to display.

We next set a particular layout scheme for the `infoPanel` panel with the command:

```
infoPanel.setLayout(new GridLayout(2,1));
```

When we add items to the panel, they are organized according to the layout scheme defined in the above statement. We have chosen to use the grid layout scheme with 2 rows and 1 column. The Java Library provides many other layout schemes.

Next we create a new "label," containing information we wish to display on the screen. A label is a component that can hold one line of text; nothing fancy, just a line of text. That is all we need here. This is accomplished by the statements:

```
JLabel countInfo = new JLabel(numCommands + " commands completed.  ");
JLabel finishedInfo = new JLabel("Testing completed. "
                                 + "Close window to exit program.");
```

Finally, we add our information to the panel and display it with:

```
infoPanel.add(countInfo);
infoPanel.add(finishedInfo);
contentPane.add(infoPanel);
outputFrame.show();
```

The first two `add` method invocations add the labels to the `infoPanel`. The third `add` method invocation adds the `infoPanel` to the `contentPane` (which is already associated with the `outputFrame`). The `show` method displays the `outputFrame` on the monitor. That's it.

In summation, to perform frame output, the `TDIncDate` program does the following:

1. Imports classes from the `awt` and `swing` packages
2. Instantiates a new `JFrame` object
3. Obtains the content pane of the new frame
4. Creates a panel to hold information
5. Defines the layout of the panel
6. Instantiates labels with the information to display
7. Adds these labels to the panel
8. Adds the panel to the content pane
9. Shows the frame

Using this frame output approach allows us to use window output without getting bogged down in too much detail. When we run our test driver program, it reads data from the input file and writes results to the output file. It then creates an output frame as a separate program thread and reports summary information about the test results there. Note that when the main thread of the program finishes, the frame thread is still running. It will run until the user closes the frame's window, activating the window-closing event that we defined through the `setDefaultCloseOperation` method.

Practical Considerations

It is obvious from this chapter that program verification techniques are time-consuming and, in a job environment, expensive. It would take a long time to do all of the things discussed in this chapter, and a programmer has only so much time to work on any par-

ticular program. Certainly not every program is worthy of such cost and effort. How can you tell how much and what kind of verification effort is necessary?

A program's requirements may provide an indication of the level of verification needed. In the classroom, your professor may specify the verification requirements as part of a programming assignment. For instance, you may be required to turn in a written, implemented test plan. Part of your grade may be determined by the completeness of your plan. In the work environment, the verification requirements are often specified by a customer in the contract for a particular programming job. For instance, a contract with a customer may specify that formal reviews or inspections of the software product be held at various times during the development process.

A higher level of verification effort may be indicated for sections of a program that are particularly complicated or error-prone. In these cases, it is wise to start the verification process in the early stages of program development in order to prevent costly errors in the design.

A program whose correct execution is critical to human life is obviously a candidate for a high level of verification. For instance, a program that controls the return of astronauts from a space mission would require a higher level of verification than a program that generates a grocery list. As a more down-to-earth example, consider the potential for disaster if a hospital's patient database system had a bug that caused it to lose information about patients' allergies to medications. A similar error in a database program that manages a Christmas card mailing list, however, would have much less severe consequences.

Summary

How are our quality software goals met by the strategies of abstraction and information hiding? When we hide the details at each level, we make the code simpler and more readable, which makes the program easier to write, modify, and reuse. Object-oriented design processes produce modular units that are also easier to test, debug, and maintain.

One positive side effect of modular design is that modifications tend to be localized in a small set of modules, and thus the cost of modifications is reduced. Remember that whenever we modify a module we must retest it to make sure that it still works correctly in the program. By localizing the modules affected by changes to the program, we limit the extent of retesting needed.

Finally, we increase reliability by making the design conform to our logical picture and delegating confusing details to lower levels of abstraction. By understanding the wide range of activities involved in software development—from requirements analysis through the maintenance of the resulting program—we gain an appreciation of a disciplined software engineering approach. Everyone knows some programming wizard who can sit down and hack out a program in an evening, working alone, coding without a formal design. But we cannot depend on wizardry to control the design, implementation, verification, and maintenance of large, complex software projects that involve the efforts of many programmers. As computers grow larger and more powerful, the problems that people want to solve on them also become larger and more complex. Some

people refer to this situation as a software *crisis*. We'd like you to think of it as a software *challenge*.

It should be obvious by now that program verification is not something you begin the night before your program is due. Design verification and program testing go on throughout the software life cycle.

Verification activities begin when we develop the software specifications. At this point, we formulate the overall testing approach and goals. Then, as program design work begins, we apply these goals. We may use formal verification techniques for parts of the program, conduct design inspections, and plan test cases. During the implementation phase, we develop test cases and generate test data to support them. Code inspections give us extra support in debugging the program before it is ever run. Figure 1.16 shows how the various types of verification activities fit into the software development cycle. Throughout the life cycle, one thing remains the same: the earlier in this cycle we can detect program errors, the easier (and less costly in time, effort, and money) they are to remove. Program verification is a serious subject; a program that doesn't work isn't worth the disk it's stored on.

Analysis	Make sure that requirements are completely understood.
	Understand testing requirements.
Specification	Verify the identified requirements.
	Perform requirements inspections with your client.
Design	Design for correctness (using assertions such as preconditions and postconditions).
	Perform design inspections.
	Plan testing approach.
Code	Understand programming language well.
	Perform code inspections.
	Add debugging output statements to the program.
	Write test plan.
	Construct test drivers.
Test	Unit test according to test plan.
	Debug as necessary.
	Integrate tested modules.
	Retest after corrections.
Delivery	Execute acceptance tests of complete product.
Maintenance	Execute regression test whenever delivered product is changed to add new functionality or to correct detected problems.

Figure 1.16 *Life-cycle verification activities*

Summary of Classes and Support Files

In this section at the end of each chapter we summarize, in tabular form, the classes defined in the chapter. The classes are listed in the order in which they appear in the text. We also include information about any other files, such as test input files, that support the material. The summary includes the name of the file, the page on which the class or support file is first referenced, and a few notes. The notes explain how the class or support file was used in the text, followed by additional notes if appropriate. The class and support files are available on our web site. They can be found in the `ch01` subdirectory of the `bookFiles` subdirectory.

Classes and Support Files Defined in Chapter 1

File	First Ref.	Notes
`Date.java`	page 14	Example of a Java class with instance and class variables.
		Unlike the original code in the text, the code on our web site includes a `toString` method.
`IncDate.java`	page 18	Demonstrates inheritance.
		The code for the `increment` command is not included (see Exercise 34).
`TDIncDate.java`	page 51	Example of a test driver; test driver for the `IncDate` class.
		In Exercise 36 we ask the student to enhance the code to include automated test verification.
`TestDataA`	page 50	Input file for `TDIncDate`.

We also include in this summary section a list of any Java library classes that were used for the first time for the classes defined in the chapter. For each library class we list its name, its package, any of its methods that are explicitly used, and the name of the program/class where they are first used. The classes are listed in the order in which they are first used. Note that in some classes the methods listed might not be defined directly in the class; they might be defined in one of its superclasses. With the classes we also list constructors, if appropriate. For more information about the library classes and methods, check the Sun Java documentation.

Library Classes Used in Chapter 1 for the First Time

Class Name	Package	Overview	Methods Used	Where Used
`JFrame`	`swing`	Manages a graphical window	`addWindowListener`, `getContentPane`, `show`, `setSize`, `setTitle`	`TDIncDate`
`String`	`lang`	Creates and parses strings	`equals`, `String`	`TDIncDate`
`BufferedReader`	`io`	Provides a buffered stream of character data	`BufferedReader`, `readLine`, `close`	`TDIncDate`
`FileReader`	`io`	Allows reading of characters from a file	`FileReader`	`TDIncDate`
`PrintWriter`	`io`	Outputs a buffered stream of character data	`PrintWriter`, `println`, `close`	`TDIncDate`
`FileWriter`	`io`	Allows reading of characters from a file	`FileWriter`	`TDIncDate`
`Container`	`awt`	Provides a container that can hold other containers	`add`	`TDIncDate`
`Jpanel`	`swing`	Provides a container for organizing display information	`add`, `JPanel`, `setLayout`	`TDIncDate`
`GridLayout`	`awt`	Creates a rectangular grid scheme for output	`GridLayout`	`TDIncDate`
`JLabel`	`swing`	Holds one line of text for display	`JLabel`	`TDIncDate`
`WindowAdapter`	`awt`	Provides null methods for window events	`WindowAdapter`	`TDIncDate`
`System`	`lang`	Various system-related methods	`exit`	`TDIncDate`
`Integer`	`lang`	Wraps the primitive `int` type	`parseInt`	`TDIncDate`

Exercises

1.1 The Software Process

1. Explain what we mean by "software engineering."
2. List four goals of quality software.
3. Which of these statements is always true?
   a. All of the program requirements must be completely defined before design begins.
   b. All of the program design must be complete before any coding begins.
   c. All of the coding must be complete before any testing can begin.
   d. Different development activities often take place concurrently, overlapping in the software life cycle.
4. Explain why software might need to be modified
   a. in the design phase.
   b. in the coding phase.
   c. in the testing phase.
   d. in the maintenance phase.
5. Goal 4 says, "Quality software is completed on time and within budget."
   a. Explain some of the consequences of not meeting this goal for a student preparing a class programming assignment.
   b. Explain some of the consequences of not meeting this goal for a team developing a highly competitive new software product.
   c. Explain some of the consequences of not meeting this goal for a programmer who is developing the user interface (the screen input/output) for a spacecraft launch system.
6. Name three computer hardware tools that you have used.
7. Name two software tools that you have used in developing computer programs.
8. Explain what we mean by "ideaware."

1.2 Program Design

9. For each of the following, describe at least two different abstractions for different viewers (see Figure 1.1).
   a. A dress
   b. An aspirin
   c. A carrot
   d. A key
   e. A saxophone
   f. A piece of wood
10. Describe four different kinds of stepwise refinement.
11. Explain how to use the nouns and verbs in a problem description to help identify candidate design classes and methods.
12. Find a tool that you can use to create UML class diagrams and recreate the diagram of the `Date` class shown in Figure 1.3.

13. What is the difference between an object and a class? Give some examples.
14. Describe the concept of inheritance, and explain how the inheritance tree is traversed to bind method calls with method implementations in an object-oriented system.
15. Make a list of potential objects from the description of the automated-teller-machine scenario given in this chapter.
16. Given the definition of the `Date` and `IncDate` classes in this chapter, and the following declarations:

```
int temp;
Date date1 = new Date(10,2,1989);
Date date2 = new Date(4,2,1992);
IncDate date3 = new IncDate(12,25,2001);
```

indicate which of the following statements are illegal, and which are legal. Explain your answers.

a. `temp = date1.dayIs();`
b. `temp = date3.yearIs();`
c. `date1.increment();`
d. `date3.increment();`
e. `date2 = date1;`
f. `date2 = date3;`
g. `date3 = date2;`

1.3 Verification of Software Correctness

17. Have you ever written a programming assignment with an error in the specifications? If so, at what point did you catch the error? How damaging was the error to your design and code?
18. Explain why the cost of fixing an error is increasingly higher the later in the software cycle the error is detected.
19. Explain how an expert understanding of your programming language can reduce the amount of time you spend debugging.
20. Explain the difference between program verification and program validation.
21. Give an example of a run-time error that might occur as the result of a programmer making too many assumptions.
22. Define "robustness." How can programmers make their programs more robust by taking a defensive approach?
23. The following program has two separate errors, each of which would cause an infinite loop. As a member of the inspection team, you could save the programmer a lot of testing time by finding the errors during the inspection. Can you help?

```
import java.io.PrintWriter;
public class TryIncrement
{
  static PrintWriter output = new PrintWriter(System.out,true);

  public static void main(String[] args) throws Exception
  {
    int count = 1;
    while(count < 10)
      output.println(" The number after " + count);   /* Now we will
      count = count + 1;                               add 1 to count */
      output.println(" is " + count);
  }
}
```

24. Is there any way a single programmer (for example, a student working alone on a programming assignment) can benefit from some of the ideas behind the inspection process?
25. When is it appropriate to start planning a program's testing?
    a. During design or even earlier
    b. While coding
    c. As soon as the coding is complete
26. Describe the contents of a typical test plan.
27. Devise a test plan to test the `increment` method of the `IncDate` class.
28. A programmer has created a module `sameSign` that accepts two `int` parameters and returns `true` if they are both the same sign, that is, if they are both positive, both negative, or both zero. Otherwise, it returns `false`. Identify a reasonable set of test cases for this module.
29. Explain the advantages and disadvantages of the following debugging techniques:
    a. Inserting output statements that may be turned off by commenting them out
    b. Using a Boolean flag to turn debugging output statements on or off
    c. Using a system debugger
30. Describe a realistic goal-oriented approach to data-coverage testing of the method specified below:

public boolean FindElement(list, targetItem)

Effect:	Searches list for targetItem.
Preconditions:	Elements of list are in no particular order; list may be empty.
Postcondition:	Returns true if targetItem is in list; otherwise, returns false.

31. A program is to read in a numeric score (0 to 100) and display an appropriate letter grade (A, B, C, D, or F).
    a. What is the functional domain of this program?
    b. Is exhaustive data coverage possible for this program?
    c. Devise a test plan for this program.
32. Explain how paths and branches relate to code coverage in testing. Can we attempt 100% path coverage?
33. Explain the phrase "life-cycle verification."
34. Create a `Date` class and an `IncDate` class as described in this chapter (or copy them from the web site). In the `IncDate` class you must create the code for the `increment` method, since that was left undefined in the chapter. Remember to follow the rules of the Gregorian calendar: A year is a leap year if either (i) it is divisible by 4 but not by 100 or (ii) it is divisible by 400. Include the preconditions and postconditions for `increment`. Use the `TDIncDate` program to test your program.
35. You should experiment with the frame output of the `TDIncDate` program. Follow the directions and record the results:
    a. Create a test input file called `MyTest.dat`.
    b. Run the program using `MyTest.dat` as the test input file, and `MyTest.out` as the output file.
    c. Change the `TestDriverFrame.java` class so that it sets the frame size to 500 × 300, and run the program again.
    d. Change the grid layout statement from a grid of 2,1 to a grid of 1,2, and run the program again.
    e. Experiment with other layout managers; use the available resources for information about them.
36. Enhance the `TDIncDate` program to include automatic test-case verification. For each of the commands that can be listed in the test-input file, you need to identify a test-result value, to be used to verify that the command was executed properly. For example, the constructor command `IncDate` can be verified by comparing the resultant value of the `IncDate` object to the date represented by the parameters of the command; the observer command `monthIs` can be verified by checking the value returned by the `monthIs` method to the expected month. The values needed to verify each command should follow the command and its parameters in the test input file. For example, a test input file could look like this:

```
IncDate Test Data B
IncDate
10
5
2002
10/5/2002
monthIs
10
quit
```

The test driver should read a command, read the command's parameters if necessary, execute the command by invoking the appropriate method, and then validate that the command completed successfully by comparing the results of the command to the test result value from the input file. The results of the test (pass or fail) should be written to the output file, and a count of the number of test cases passed and failed should be written to the screen.

37. Create a new program that uses the same basic architecture as the test driver program modeled in Figure 1.14, and that uses the same set of Java I/O statements as `TDIncDate(readLine`, `setLayout`, and so on). This is an open problem; your program can do whatever you like. For example, the input file could contain a list of student names plus three test grades for each student:

```
Smith
100
90
80
Jones
95
95
95
```

And the corresponding output file could contain the student's names and averages:

```
Smith
90
Jones
95
```

Finally, the output frame could contain summary information: for example, the number of students, the total average, the highest average, and so on. Remember to design your program so that the user can indicate the input and output file names through command-line parameters.

Chapter 2

Data Design and Implementation

Goals

Measurable goals for this chapter include that you should be able to

- describe the benefits of using an abstract data type (ADT)
- explain the difference between a primitive type and a composite type
- describe an ADT from three perspectives: logical level, application level, and implementation level
- explain how a specification can be used to document the design of an ADT
- describe, at the logical level, the component selector, and describe appropriate applications for the Java built-in types: class and array
- create code examples that demonstrate the ramifications of using references
- describe several hierarchical types, including aggregate objects and multidimensional arrays
- use packages to organize Java compilation units
- use the Java Library classes `String` and `ArrayList`
- identify the scope of a Java variable in a program
- explain the difference between a deep copy and a shallow copy of an object
- identify, define, and use Java exceptions when creating an ADT
- list the steps to follow when creating ADTs with the Java *class* construct

This chapter centers on data and the language structures used to organize data. When problem solving, the way you view the data of your problem domain and how you structure the data that your programs manipulate greatly influence your success. Here you learn how to deal with the complexity of your data using abstraction and how to use the Java language mechanisms that support data abstraction.

In this chapter, we also cover the various data types supported by Java: the primitive types (`int`, `float`, and so on), classes, interfaces, and the array. The Java class mechanism is used to create data types beyond those directly provided by the language. We review some of the class-based types that are provided in the Java Class Library and show you how to create your own class-based types. We use the Java class mechanism to encapsulate the data structures you are studying, as ADTs, throughout the textbook.

2.1 Different Views of Data

Data Types

When we talk about the function of a program, we usually use words like *add*, *read*, *multiply*, *write*, *do*, and so on. The function of a program describes what it does in terms of the verbs in the programming language. The data are the nouns of the programming world: the objects that are manipulated, the information that is processed by a computer program.

Data The representation of information in a manner suitable for communication or analysis by humans or machines

Data type A category of data characterized by the supported elements of the category and the supported operations on those elements

Atomic or primitive type A data type whose elements are single, nondecomposable data items

Humans have evolved many ways of encoding information for analysis and communication, for example letters, words, and numbers. In the context of a programming language, the term *data* refers to the representation of such information, from the problem domain, by the data types available in the language.

A data type can be used to characterize and manipulate a certain variety of data. It is formally defined by describing:

1. the collection of elements that it can represent.
2. the operations that may be performed on those elements.

Most programming languages provide simple data types for representing basic information—types like integers, real numbers, and characters. For example, an integer might represent a person's age; a real number might represent the amount of money in a bank account. An integer data type in a language would be formally defined by listing the range of numbers it can represent and the operations it supports, usually the standard arithmetic operations.

The simple types are also called atomic types or primitive types, because they cannot be broken into parts. Languages usually provide ways for a programmer to combine primitive types into more complex structures, which can capture relationships among the individual data items. For example, a programmer can combine two primitive inte-

ger values to represent a point in the *x*-*y* plane or create a list of real numbers to represent the scores of a class of students on an assignment. A data type composed of multiple elements is called a composite type.

Composite type A data type whose elements are composed of multiple data items

Data abstraction The separation of a data type's logical properties from its implementation

Just as primitive types are partially defined by describing their domain of values, composite types are partially defined by the relationship among their constituent values.

Composite data types come in two forms: unstructured and structured. An *unstructured* composite type is a collection of components that are not organized with respect to one another. A *structured* composite type is an organized collection of components in which the organization determines the means of accessing individual data components or subsets of the collection. In addition to describing their domain of values, primitive types are defined by describing permitted operations. With composite types, the main operation of interest is accessing the elements that make up the collection.

The mechanisms for building composite types in the Java language are called reference types. (We see why in the next section.) They include arrays and classes, which you are probably familiar with, and interfaces. We review all of these mechanisms in the next section.

In a sense, any data processed by a computer, whether it is primitive or composite, is just a collection of bits that can be turned on or off. The computer itself needs to have data in this form. Human beings, however, tend to think of information in terms of somewhat larger units like numbers and lists, and thus we want at least the human-readable portions of our programs to refer to data in a way that makes sense to us. To separate the computer's view of data from our own, we use data abstraction to create another view.

Data Abstraction

Many people feel more comfortable with things that they perceive as real than with things that they think of as abstract. Thus, *data abstraction* may seem more forbidding than a more concrete entity like *integer*. Let's take a closer look, however, at that very concrete—and very abstract—integer you've been using since you wrote your earliest programs. Just what is an integer? Integers are physically represented in different ways on different computers. In the memory of one machine, an integer may be a binary-coded decimal. In a second machine, it may be a sign-and-magnitude binary. And in a third one, it may be represented in two's-complement binary notation. Although you may not be familiar with these terms, that hasn't stopped you from using integers. (You can learn about these terms in an assembly language or computer organization course, so we do not explain them here.) Figure 2.1 shows some different representations of an integer.

The way that integers are physically represented determines how the computer manipulates them. As a Java programmer, however, you don't usually get involved at this level; you simply use integers. All you need to know is how to declare an `int` type variable and what operations are allowed on integers: assignment, addition, subtraction, multiplication, division, and modulo arithmetic.

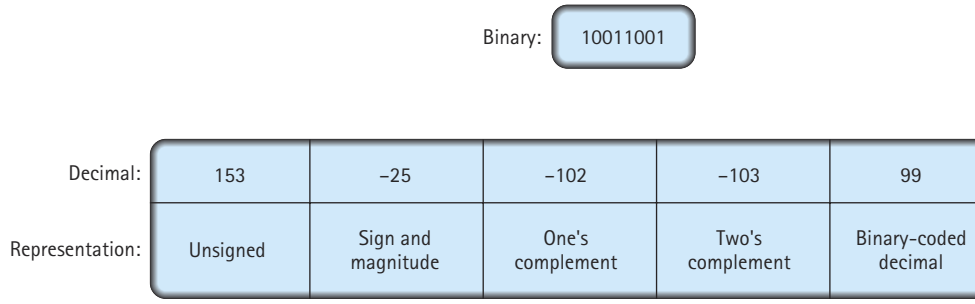


Figure 2.1 *The decimal equivalents of an 8-bit binary number*

Consider the statement

```
distance = rate * time;
```

It's easy to understand the concept behind this statement. The concept of multiplication doesn't depend on whether the operands are, say, integers or real numbers, despite the fact that integer multiplication and floating-point multiplication may be implemented in very different ways on the same computer. Computers would not be very popular if every time we wanted to multiply two numbers we had to get down to the machine-representation level. But we don't have to: Java has provided the `int` data type for us, hiding all the implementation details and giving us just the information we need to create and manipulate data of this type.

We say that Java has encapsulated integers for us. Think of the capsules surrounding the medicine you get from the pharmacist when you're sick. You don't have to know anything about the chemical composition of the medicine inside to recognize the big blue-and-white capsule as your antibiotic or the little yellow capsule as your decongestant. Data encapsulation means that the physical representation of a program's data is hidden by the language. The programmer using the data doesn't see the underlying implementation, but deals with the data only in terms of its logical picture—its abstraction.

Data encapsulation The separation of the representation of data from the applications that use the data at a logical level; a programming language feature that enforces information hiding

But if the data are encapsulated, how can the programmer get to them? Operations must be provided to allow the programmer to create, access, and change the data. Let's look at the operations Java provides for the encapsulated data type `int`. First of all, you can create variables of type `int` using declarations in your program. Then you can assign values to these integer variables by using the assignment operator and perform arithmetic operations on them using `+`, `-`, `*`, `/`, and `%`. Figure 2.2 shows how Java has encapsulated the type `int` in a nice neat black box.

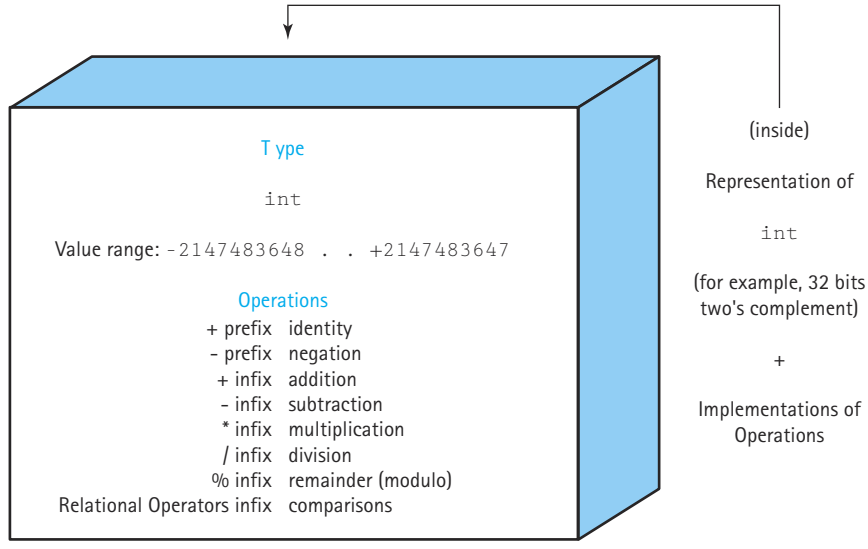


Figure 2.2 *A black box representing an integer*

The point of this discussion is that you have been dealing with a logical data abstraction of integer since the very beginning. The advantages of doing so are clear: you can think of the data and the operations in a logical sense and can consider their use without having to worry about implementation details. The lower levels are still there—they're just hidden from you.

Remember that the goal in design is to reduce complexity through abstraction. We extend this goal with another: to protect our data abstraction through encapsulation. We refer to the set of all possible values (the domain) of an encapsulated data "object," plus the specifications of the operations that are provided to create and manipulate the data, as an abstract data type (ADT for short).

Abstract data type (ADT) A data type whose properties (domain and operations) are specified independently of any particular implementation

In effect, all the Java built-in types are ADTs. A Java programmer can declare variables of those types without understanding the underlying implementation. The programmer can initialize, modify, and access the information held by the variables using the provided operations.

In addition to the built-in ADTs, Java programmers can use the Java *class* mechanism to build their own ADTs. For example, the `Date` class defined in Chapter 1 can be viewed as an ADT. Yes, it is true that the programmers who created it need to know about its underlying implementation; for example, they need to know that a `Date` is composed of three `int` instance variables, and they need to know the names of the instance variables. The application programmers who use the `Date` class, however, do not need this information. They only need to know how to create a `Date` object and how to invoke the exported methods to use the object.

Data Structures

A single integer can be very useful if we need a counter, a sum, or an index in a program. But generally, we must also deal with data that have many parts and complex interrelationships among those parts. We use a language's composite type mechanisms to build structures, called data structures, which mirror those interrelationships. Note that the data elements that make up a data structure can be any combination of primitive types, unstructured composite types, and structured composite types.

Data structure A collection of data elements whose logical organization reflects a relationship among the elements. A data structure is characterized by accessing operations that are used to store and retrieve the individual data elements.

When designing our data structures we must consider how the data is used because our decisions about what structure to impose greatly affect how efficient it is to use the data. Computer scientists have developed classic data, such as lists, stacks, queues, trees, and graphs, through the years. They form the major area of focus for this textbook.

In languages like Java, that provide an encapsulation mechanism, it is best to design our data structures as ADTs. We can then hide the detail of how we implement the data structure inside a class that exports methods for using the structure. For example, in Chapter 3 we develop a list data structure as an ADT using the Java *class* and *interface* constructs.

As we saw in Chapter 1, the basic operations that are performed on encapsulated data can be classified into categories. We have already seen three of these: *constructor*, *transformer,* and *observer*. As we design operations for data structures, a fourth category becomes important: *iterator*. Let's take a closer look at what each category does.

- A constructor is an operation that creates a new instance (object) of the data type. A constructor that uses the contents of an existing object to create a new object is called a *copy constructor*.
- Transformers (sometimes called *mutators*) are operations that change the state of one or more of the data values, such as inserting an item into an object, deleting an item from an object, or making an object empty.
- An observer is an operation that allows us to observe the state of one or more of the data values without changing them. Observers come in several forms: *predicates* that ask if a certain property is true, *accessor* or *selector* methods that return a value based on the contents of the object, and *summary* methods that return information about the object as a whole. A Boolean method that returns `true` if an object is empty and `false` if it contains any components is an example of a predicate. A method that returns a copy of the last item put into a structure is an example of an accessor method. A method that returns the number of items in a structure is a summary method.
- An iterator is an operation that allows us to process all the components in a data structure sequentially. Operations that return successive list items are iterators.

Data structures have a few features worth noting. First, they can be "decomposed" into their component elements. Second, the organization of the elements is a feature of

the structure that affects how each element is accessed. Third, both the arrangement of the elements and the way they are accessed can be encapsulated.

Note that although we design our data structures as ADTs, data structures and ADTs are not equivalent. We could implement a data structure without using any data encapsulation or information hiding whatsoever (but we won't!). Also, the fact that a construct is defined as an ADT does not make it a data structure. For example, the `Date` class defined in Chapter 1 implements a Date ADT, but that is not considered to be a data structure in the classical sense. There is no structural relationship among its components.

Data Levels

An ADT specifies the logical properties of a data type. Its implementation provides a specific representation such as a set of primitive variables, an array, or even another ADT. A third view of a data type is how it is used in a program to solve a particular problem; that is, its application. If we were writing a program to keep track of student grades, we would need a list of students and a way to record the grades for each student. We might take a by-hand grade book and model it in our program. The operations on the grade book might include adding a name, adding a grade, averaging a student's grades, and so forth. Once we have written a specification for our grade-book data type, we must choose an appropriate data structure to use to implement it and design the algorithms to implement the operations on the structure.

In modeling data in a program, we wear many hats. We must determine the abstract properties of the data, choose the representation of the data, and develop the operations that encapsulate this arrangement. During this process, we consider data from three different perspectives, or levels:

1. *Logical (or abstract) level:* An abstract view of the data values (the domain) and the set of operations to manipulate them. At this level, we define the ADT.
2. *Application (or user) level:* A way of modeling real-life data in a specific context; also called the problem domain. Here the application programmer uses the ADT to solve a problem.
3. *Implementation level:* A specific representation of the structure to hold the data items, and the coding of the operations in a programming language. This is how we actually represent and manipulate the data in memory: the underlying structure and the algorithms for the operations that manipulate the items on the structure. For the built-in types, this level is hidden from the programmer.

An Analogy

Let's look at a real-life example: a library. A library can be decomposed into its component elements: books. The collection of individual books can be arranged in a number of ways, as shown in Figure 2.3. Obviously, the way the books are physically arranged on the shelves determines how one would go about looking for a specific volume. The particular library we're concerned with doesn't let its patrons get their own books, however; if you want a book, you must give your request to the librarian, who gets the book for you.

The library "data structure" is composed of elements (books) with a particular interrelationship; for instance, they might be ordered based on the Dewey decimal system.

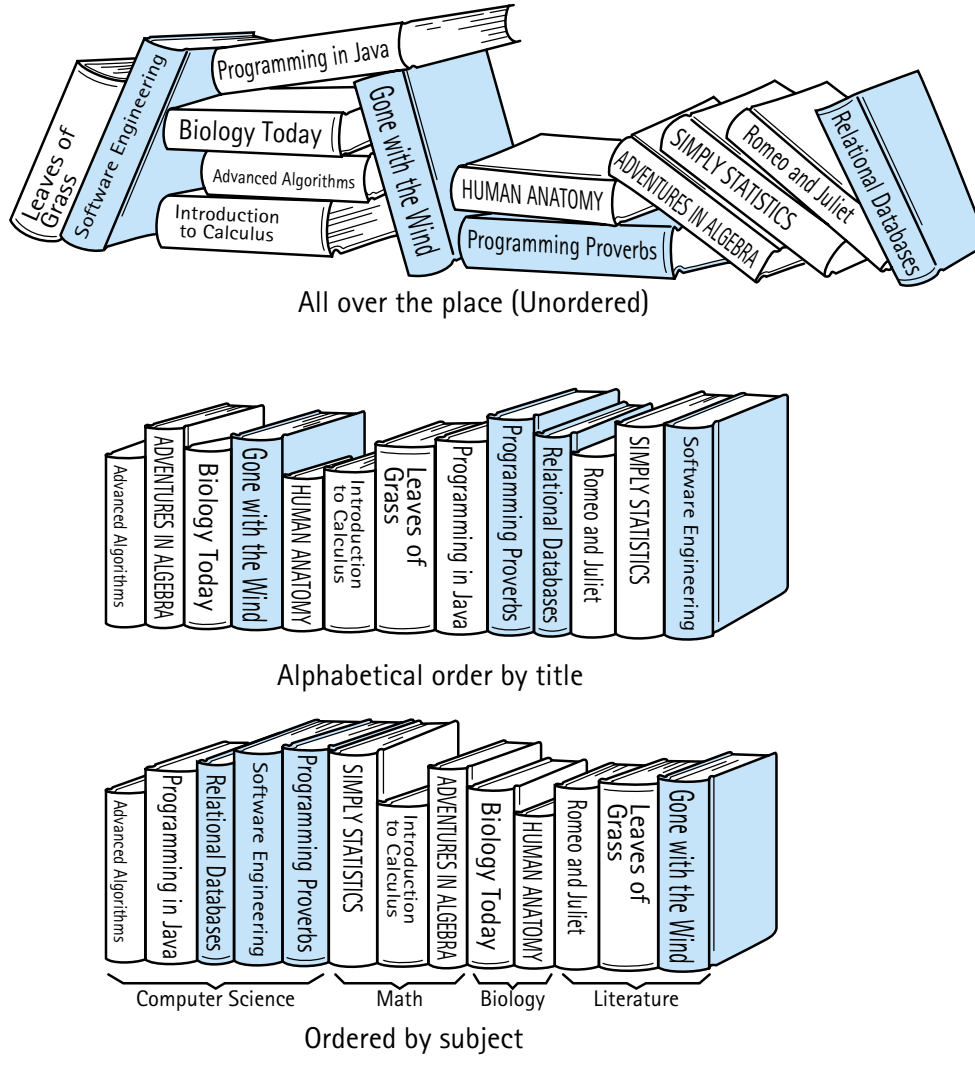


Figure 2.3 *A collection of books ordered in different ways*

Accessing a particular book requires knowledge of the arrangement of the books. The library user doesn't have to know about the structure, though, because it has been encapsulated: Users access books only through the librarian. The physical structure and abstract picture of the books in the library are not the same. The online catalog provides logical views of the library—ordered by subject, author, or title—that are different from its underlying representation.

We use this same approach to data structures in our programs. A data structure is defined by (1) the logical arrangement of data elements, combined with (2) the set of operations we need to access the elements. Let's see what our different viewpoints mean

in terms of our library analogy. At the application level, there are entities like the Library of Congress, the Dimsdale Collection of Rare Books, the Austin City Library, and the North Amherst branch library.

At the logical level, we deal with the "what" questions. What is a library? What services (operations) can a library perform? The library may be seen abstractly as "a collection of books" for which the following operations are specified:

- Check out a book.
- Check in a book.
- Reserve a book that is currently checked out.
- Pay a fine for an overdue book.
- Pay for a lost book.

How the books are organized on the shelves is not important at the logical level, because the patrons don't actually have direct access to the books. The abstract viewer of library services is not concerned with how the librarian actually organizes the books in the library. The library user only needs to know the correct way to invoke the desired operation. For instance, here is the user's view of the operation to check in a book: Present the book at the check-in window of the library from which the book was checked out, and receive a fine slip if the book is overdue.

At the implementation level, we deal with the answers to the "how" questions. How are the books cataloged? How are they organized on the shelf? How does the librarian process a book when it is checked in? For instance, the implementation information includes the fact that the books are cataloged according to the Dewey decimal system and arranged in four levels of stacks, with 14 rows of shelves on each level. The librarian needs such knowledge to be able to locate a book. This information also includes the details of what happens when each of the operations takes place. For example, when a book is checked back in, the librarian may use the following algorithm to implement the check-in operation:

CheckInBook

```
Examine due date to see whether the book is late.
if book is late
    Calculate fine.
    Issue fine slip.
Update library records to show that the book has been returned.
Check reserve list to see if someone is waiting for the book.
if book is on reserve list
    Put the book on the reserve shelf.
else
    Replace the book on the proper shelf, according to the library's shelf arrangement scheme.
```

All this, of course, is invisible to the library user. The goal of our design approach is to hide the implementation level from the user.

Picture a wall separating the application level from the implementation level, as shown in Figure 2.4. Imagine yourself on one side and another programmer on the other side. How do the two of you, with your separate views of the data, communicate across this wall? Similarly, how do the library user's view and the librarian's view of the library come together? The library user and the librarian communicate through the data abstraction. The abstract view provides the specification of the accessing operations without telling how the operations work. It tells *what* but not *how*. For instance, the abstract view of checking in a book can be summarized in the following specification:

float CheckIn (book)

Effect:	Accesses book and checks it into this library. Returns a fine amount (0 if there is no fine).
Preconditions:	Book was checked out of this library; book is presented at the check-in desk.
Postconditions:	return value = (amount of fine due); contents of this library is the original contents + book
Exception:	This library is not open

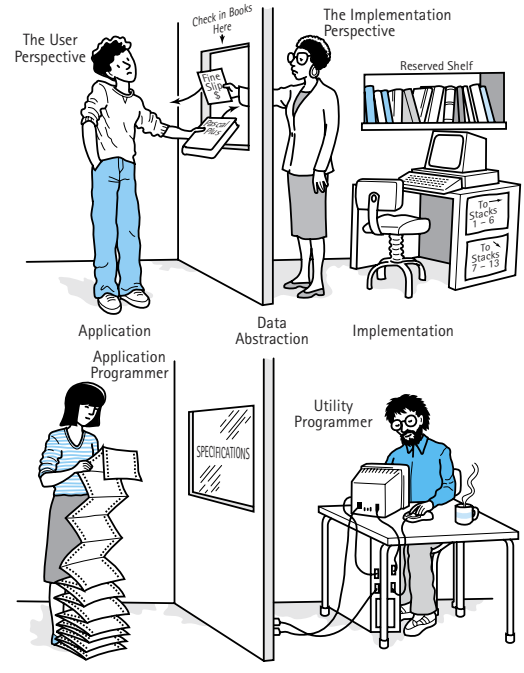


Figure 2.4 *Communication between the application level and implementation level*

The only communication from the user into the implementation level is in terms of input specifications and allowable assumptions—the preconditions of the accessing routines. The only output from the implementation level back to the user is the transformed data structure described by the output specifications, or postconditions, of the routines, or the possibility of an exception being raised. Remember that exceptions are extraordinary situations that disrupt the normal processing of the operation. The abstract view hides the underlying structure but provides functionality through the specified accessing operations.

Although in our example there is a clean separation, provided by the library wall, between the use of the library and the inside organization of the library, there is one way that the organization can affect the users—efficiency. For example, how long does a user have to wait to check out a book? If the library shelves are kept in an organized fashion, as described above, then it should be relatively easy for a librarian to retrieve a book for a customer and the waiting time should be reasonable. On the other hand, if the books are just kept in unordered piles, scattered around the building, shoved into corners and piled on staircases, the wait time for checking out a book could be very long. But in such a library it sure would be easy for the librarian to handle checking in a book—just throw it on the closest pile!

The decisions we make about the way data are structured affect how efficiently we can implement the various operations on that data. One structure leads to efficient implementation of some operations, while another structure leads to efficient implementation of other operations. Efficiency of operations can be important to the users of the data. As we look at data structures throughout this textbook we discuss the benefits and drawbacks of various design structure decisions. We often study alternative organizations, with differing efficiency ramifications.

When you write a program as a class assignment, you often deal with data at each of our three levels. In a job situation, however, you may not. Sometimes you may program an application that uses a data type that has been implemented by another programmer. Other times you may develop "utilities" that are called by other programs. In this book we ask you to move back and forth between these levels.

2.2 Java's Built-In Types

Java's classification of built-in data types is shown in Figure 2.5. As you can see, there are eight primitive types and three composite types; of the composite types, two are unstructured and one is structured. You are probably somewhat familiar with several of the primitive types and the composite types `class` and `array`.

In this section, we review all of the built-in types. We discuss them from the point of view of two of the levels defined in the previous section: the logical (or abstract) level and the application level. We do not look at the implementation level for the built-in types, since the Java environment hides it and we, as programmers, do not need to understand this level in order to use the built-in types. (Note, however, that when we begin to build our own types and structures, the implementation view becomes one of our major concerns.) For the built-in types we can interpret the remaining two levels as follows:

- The logical or abstract level involves understanding the domain of the data type and the operations that can be performed on data of that type. For the composite

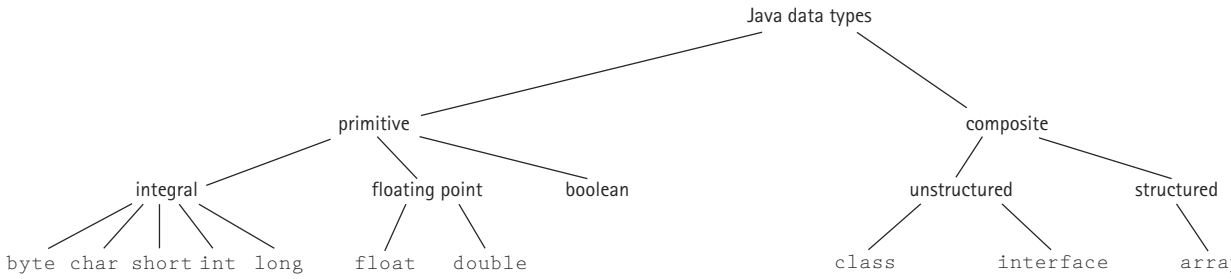


Figure 2.5 *Java data types*

types, the main operation of concern is how to access the various components of the type.

- The application level—in other words, the view of how we use the data types—includes the rules for declaring and using variables of the type, in addition to considerations of what the type can be used to model.

Primitive Data Types

Java's primitive types are `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, and `short`. These primitive types share similar properties. We first look closely at the `int` type from our two points of view, and then we give a summary review of all the others. We understand that you are already familiar with the `int` type; we are using this opportunity to show you how we apply our two levels to the built-in types.

Logical Level

In Java, variables of type `int` can hold an integer value between −2147483648 and 2147483647. Java provides the standard prefix operations of unary plus (+) and unary minus (-). Also, of course, the infix operations of addition (+), subtraction (-), multiplication (*), division (/), and modulus (%). We are sure you are familiar with all of these operations; remember that integer division results in an integer, with no fractional part.

Application Level

We declare variables of type `int` by using the keyword `int`, followed by the name of the variable, followed by a semicolon. For example

```
int numStudents;
```

You can declare more than one variable of type `int`, by separating the variable names with commas, but we prefer one variable per declaration statement. You can also provide an initial value for an `int` variable by following the name of the variable with an "= value" expression. For example

```
int numStudents = 50;
```

If you do not initialize an `int` variable, the system initializes it to the value 0. However, many compilers refuse to generate Java byte code if they determine that you could be using an uninitialized variable, so it is always a good idea to ensure that your variables are assigned values before they are used in your programs.

Variables of type `int` are handled within a program "by value." This means the variable name represents the location in memory of the value of the variable. This information may seem to belong in a subsection on implementation. However, it does directly affect how we use the variables in our programs, which is the concern of the application level. We treat this topic more completely when we reach Java's composite types, which are not handled by value.

For completeness sake, we should mention what an `int` variable can be used to model: Essentially anything that can be characterized by an integer value in the range stated above. Programs that can be modeled with an integer between negative two billion and positive two billion include the number of students in a class, test grades, city populations, and so forth.

We could repeat the analysis we made above of the `int` type for each of the primitive data types, but the discussion would quickly become redundant. Note that `byte`, `short`, and `long` types are also used to hold integer values, `char` is used to store Unicode characters, `float` and `double` are used to store "real" numbers, and the `boolean` type represents either `true` or `false`. Appendix C contains a table showing, for each primitive type, the kind of value stored by the type, the default value, the number of bits used to implement the type, and the possible range of values.

Let's move on to the composite types.

The Class Type

Primitive data types are the building blocks for composite types. A composite type gathers together a set of component values, sometimes imposing a specific arrangement on them (see Figure 2.6). If the composite type is a built-in type such as an array, the accessing mechanism is provided in the syntax of the language. If the composite type is

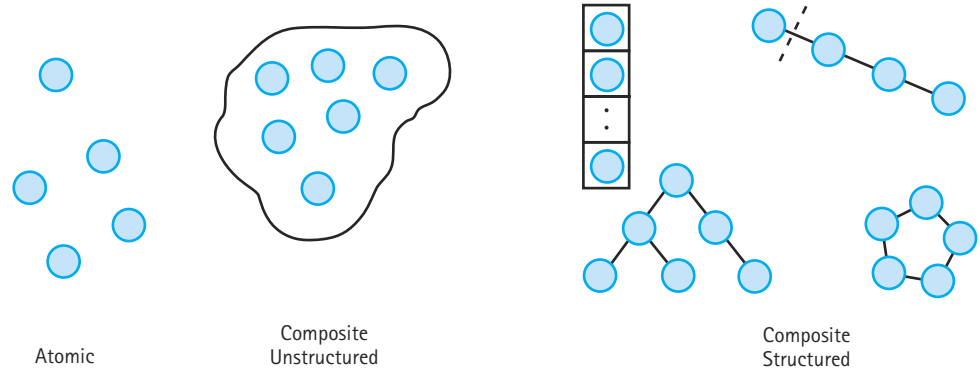


Figure 2.6 *Atomic (simple) and composite data types*

a user-defined type, such as the `Date` class defined in Chapter 1, the accessing mechanism is built into the methods provided with the class.

You are already familiar with the Java *class* construct from your previous courses and from the review in Chapter 1. The class can be a mechanism for creating composite data types. A specific class has a name and is composed of named data fields (class and instance variables—sometimes called *attributes*) and methods. The data elements and methods are also known as *members* of the class. The members of a class can be accessed individually by name. A class is unstructured because the meaning is not dependent on the ordering of the members within the source code. That is, the order in which the members of the class are listed can be changed without changing the function of the class.

In object-oriented programming, classes are usually defined to hold and hide data and to provide operations on that data. In that case, we say that the programmer has used the *class* construct to build his or her own ADT—and that is the focus of this textbook. However, in this section on built-in types, we use the class strictly to hold data. We do not hide the data and we do not define any methods for our classes. The class variables are public, not private. We use a class strictly to provide unstructured composite data collections. This type of construct has classically been called a *record*. The record is not available in all programming languages. FORTRAN, for instance, historically has not supported records; newer versions may. However, COBOL, a business-oriented language, uses records extensively. C and C++ programmers are able to implement records. Java classes provide the Java programmer with a record mechanism.

Many textbooks that use Java do not present this use of the Java *class* construct, since it is not considered a pure object-oriented construct. We agree that when practicing object-oriented design you should not use classes in the manner presented in this section. However, we present the approach for several reasons:

1. Other languages support the record mechanism, and you may find yourself working with those languages at some time.
2. Using this approach allows us to address the declaration, creation, and use of objects without the added complexity of dealing with class methods.
3. Later, when we discuss using classes to hide data, we can compare the information-hiding approach to the approach described here. The benefits of information hiding might not be as obvious if you hadn't seen any other approach.

In the following discussion, to differentiate the simple use of the *class* construct used here, from its later use to create ADTs, we use the generic term *record* in place of *class*.

Logical Level

A record is a composite data type made up of a finite collection of not necessarily homogeneous elements called fields. Accessing is done directly through a set of named field selectors.

We illustrate the syntax and semantics of the component selector within the context of the following program:

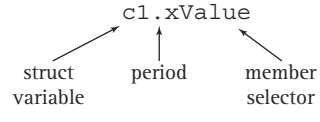
```
public class TestCircle
{
  static class Circle
  {
    int xValue;       // Horizontal position of center
    int yValue;       // Vertical position of center
    float radius;
    boolean solid;    // True means circle filled
  }

  public static void main(String[] args)

  {
    Circle c1 = new Circle();
    c1.xValue = 5;
    c1.yValue = 3;
    c1.radius = 3.5f;
    c1.solid = true;

    System.out.println("c1:    " + c1);
    System.out.println("c1 x: " + c1.xValue);
  }
}
```

The above program declares a record structure called `Circle`. The main method instantiates and initializes the fields of the `Circle` record `c1`, and then prints the record and the `xValue` field of the record to the output. The output looks like this:

```
c1:   TestCircle$Circle[at]111f71
c1 x: 5
```

The `Circle` record variable (the circle object) `c1` is made up of four components (or fields, or instance variables). The first two, `xValue` and `yValue`, are of type `int`. The third, `radius`, is a `float` number. The fourth, `solid`, is a `boolean`. The names of the components make up the set of member selectors.

The syntax of the component selector is the record variable name, followed by a period, followed by the member selector for the component you are interested in:



If this expression is on the left-hand side of an assignment statement, a value is being stored in that member of the record; for example:

```
c1.xValue = 5;
```

If it is used somewhere else, a value is being extracted from that place; for example:

```
output.println("c1 x: " + c1.xValue);
```

Application Level

Records are useful for modeling objects that have a number of characteristics. Records allow us to associate various types of data with each other in the form of a single item. We can refer to the composite item by a single name. We also can refer to the different members of the item by name. You probably have seen many examples of records used in this way to represent items.

We declare and instantiate a record the same way we declare and instantiate any Java object; we use the *new* command:

```
Circle c1 = new Circle();
```

Notice that we did not supply a constructor method in our definition of the `Circle` class in the above program. When using the class as a record mechanism it is not necessary to provide a constructor, since the record components are not hidden and can be initialized directly from the application. Of course, you can provide your own constructor if you like, and that may simplify the use of the record. If no constructor is defined, Java provides a default constructor that initializes the constituent parts of the record to their default values.

In the previous section we discussed how primitive types such as `int`s are handled "by value." This is in contrast to how all nonprimitive types, including records or any objects, are handled. The variable of a primitive type holds the value of the variable, whereas a variable of a nonprimitive type holds a *reference* to the value of the variable. That is, the variable holds the address where the system can find the value of the variable. We say that the nonprimitive types are handled "by reference." This is why, in Java, composite types are known officially as reference types. *Understanding the ramifications of handling variables by reference is very important, whether we are dealing with records, other objects, or arrays.*

The differences between the ways "by value" and "by reference" variables are handled is seen most dramatically in the result of a simple assignment statement. Figure 2.7 shows the result of the assignment of one `int` variable to another `int` variable, and the result of the assignment of one `Circle` object to another `Circle` object. Actual circles represent the `Circle` objects in the figure.

When we assign a variable of a primitive type to another variable of the same type, the latter becomes a copy of the former. But, as you can see from the figure, this is not the case with reference types. When we assign object `c2` to object `c1`, `c1` does *not* become a copy of `c2`. Instead, the reference associated with `c1` becomes a copy of the reference associated with `c2`. This means that both `c1` and `c2` now reference the same object. The feature section below looks at the ramifications of using references from four perspectives: aliases, garbage, comparison, and use as parameters.

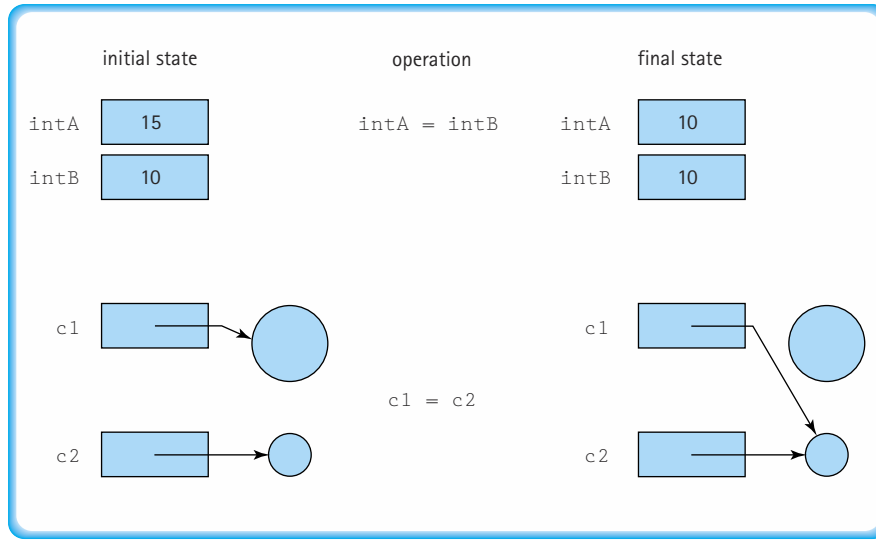


Figure 2.7 *Results of assignment statements*

Java includes a reserved word `null` that indicates an absence of reference. If a reference variable is declared without being assigned an instantiated object, it is automatically initialized to the value `null`. You can also assign `null` to a variable, for example:

```
c1 = null;
```

And you can use `null` in a comparison:

```
if (c1 == null)
  output.println("The Circle is not instantiated");
```

Ramifications of Using References

Aliases

The assignment of one object to another object, as shown in Figure 2.7, results in both object variables referring to the same object. Thus, we have two names for the same object. In this case we say that we have an "alias" of the object. Good programmers avoid aliases because they make programs hard to understand. An object's state can change, even though it appears that the program did not access the object, when the object is accessed through the alias. For

example, consider the `IncDate` class that was defined in Chapter 1. If `date1` and `date2` are aliases for the same `IncDate` object, then the code

```
output.println(date1);
date2.increment();
output.println(date1);
```

would print out two different dates, even though at first glance it would appear that it should print out the same date twice. This type of behavior can be very confusing for a maintenance programmer and lead to hours of frustrating testing and debugging.

Garbage

It would be fair to ask in the situation depicted in the lower half of Figure 2.7, what happens to the space being used by the larger circle? After the assignment statement, the program has lost its reference to the large circle, and so it can no longer be accessed. Memory space like this, that has been allocated to a program but that can no longer be accessed by a program, is called garbage. There are other ways that garbage can be created in a Java program. For example, the following code would create 100 objects of class `Circle`; but only one of them can be accessed through `c1` after the loop is finished executing:

```
Circle c1;
for (n = 1; n <= 100; n++)
{
  Circle c1 = new Circle();
  // code to initialize and use c1 goes here
}
```

The other 99 objects cannot be reached by the program. They are garbage.

When an object is unreachable, the Java run time system marks it as garbage., The system regularly performs an operation known as garbage collection, in which it finds unreachable objects and deallocates their storage space, making it once again available in the free pool for the creation of new objects.

This approach, of creating and destroying objects at different points in the application by allocating and deallocating space in the free pool is called dynamic memory management. Without it, the computer would be much more likely to run out of storage space for data.

Garbage The set of currently unreachable objects

Garbage collection The process of finding all unreachable objects and deallocating their storage space

Deallocate To return the storage space for an object to the pool of free memory so that it can be reallocated to new objects

Dynamic memory management The allocation and deallocation of storage space as needed while an application is executing

Comparing Objects

The fact that nonprimitive types are handled by reference impacts the results returned by the == comparison operator. Two variables of a nonprimitive type are considered identical, in terms of

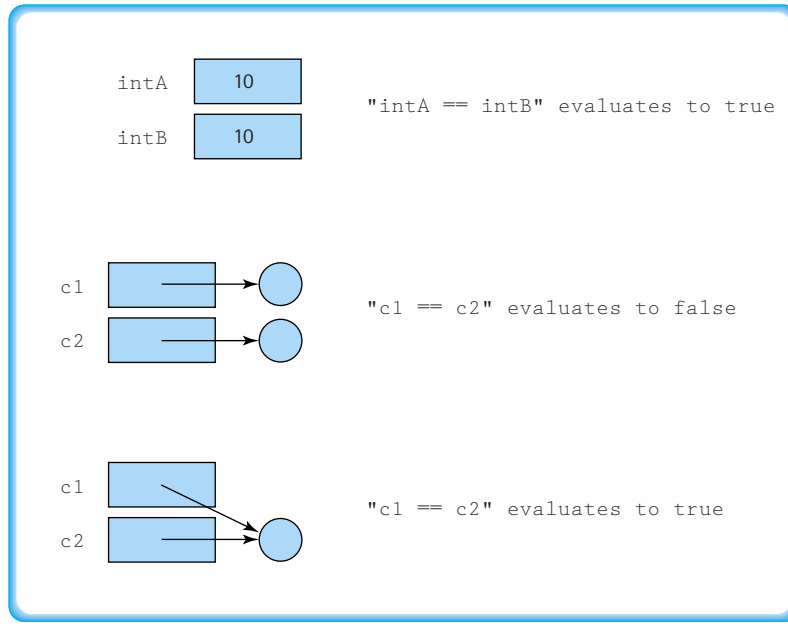


Figure 2.8 *Comparing primitive and nonprimitive variables*

the == operator, only if they are aliases for one another. This makes sense when you consider that the system compares the contents of the two variables. That is, it compares the two references that those variables contain. So even if two variables of type `Circle` have the same `xValue` values, the same `yValue` values, the same `radius` values, and the same `solid` values, they are not considered equal. Figure 2.8 shows the results of using the comparison operator in various situations.

Parameter Passing

When methods are invoked, they are often passed information (arguments) through their parameters. Some programming languages allow the programmer to control whether arguments are passed by value (a copy of the argument's value is used) or by reference (a copy of the argument's memory location is used). Java does not allow such control. Whenever a variable is passed as an argument, the value stored in that variable is copied into the method's corresponding parameter. All Java arguments are passed by value. Therefore, if the variable is of a primitive type, the actual value (`int`, `double`, and so on) is passed to the method; and if it is a reference type, then the reference that it contains is passed to the method.

Notice that passing a reference variable as an argument causes the receiving method to receive an alias of the object that is referenced by the variable. If it uses the alias to make changes to the object, then when the method returns, an access via the variable finds the object in its modified state.

We return many times to these subtle, but important, considerations.

Interfaces

The word *interface* means a common boundary shared by two interacting systems. We use the term in many ways in computer science. For example, the user interface of a program is the part of the program that interacts with the user, and the interface of an object's method is its set of parameters and the return value it provides.

In Java, the word interface has a very specific meaning. In fact, `interface` is a Java keyword. We look briefly at interfaces in this subsection. Throughout the textbook we find places to use the Java *interface* mechanism, at which times we expand our coverage of the topic.

Logical Level

A Java interface looks very similar to a Java class. It can include data, that is, variable declarations, and methods. However, all variables declared in an interface must be `final`, `static` variables; in other words, they must be constants. And only the interface descriptions of methods are included; no method bodies or implementations are allowed. Perhaps this is why the language designers decided to call this construct an interface. Methods that are declared without bodies are called abstract methods.

Abstract method A method declared in a class or an interface without a method body

Here is an example of an interface, with one constant, `Pi`, and three abstract methods, `perimeter`, `area`, and `weight`:

```
public interface FigureGeometry
{
  public static final float Pi = 3.14;

  public abstract float perimeter();
  // Returns perimeter of current object

  public abstract float area();
  // Returns area of current object

  public abstract int weight(int scale);
  // Returns weight of current object
}
```

Java provides the keyword `abstract` that we must use when declaring an abstract method in a class. But we do not need to use it when defining the methods in an interface. Its use is redundant, since all methods of an interface must be abstract. We could have omitted it from the above code segment, but chose to show how it may optionally be used, as added documentation, to remind us that the methods are abstract.

At the logical level we look at the domain of values of a data type and the available operations to manipulate them. The domain of values for an interface is made up of classes! Interfaces are used by being "implemented" by classes. For example, a program-

mer has a `Circle` class implement the `FigureGeometry` interface by using the following line to begin the `Circle` class:

```
public class Circle implements FigureGeometry
```

When a class implements an interface, it receives access to all of the constants defined in the interface. It must provide an implementation, that is, a body, for all the abstract methods declared in the interface. So, the `Circle` class and any other class that implements the `FigureGeometry` interface, would be required to repeat the declarations of the three methods and also provide code for their bodies. Classes that implement an interface are not constrained to only implementing the abstract methods; they can also add data fields and methods of their own.

There are some other issues with interfaces (relationship to abstract classes, use of subinterfaces) that we address, when needed, later in the text.

Application Level

Interfaces are a versatile and powerful programming construct. They can be used in the following ways.

As a contract If we have an abstract view of a class that can have several different implementations, we can capture our abstract view in an interface. Then we can have separate classes implement the interface, with each class providing one of the alternate implementations. This way we are sure that all of the classes provide the same abstraction; we should be able to use them interchangeably in our application programs.

To share constants If there is a set of constant values that we want to use in several different classes, we can define the constants in an interface and have each of the classes implement the interface. Implementing the interface provides access rights to the constants.

To replace multiple inheritance Some languages allow classes to inherit from more than one superclass. This is called *multiple inheritance*. Java does not support multiple inheritance because it can lead to obtuse programs and would greatly complicate the underlying Java environment. However, there are many situations for which we would like to relate the definition of a new class to more than one previously defined class. In these cases, in Java, we use interfaces. A class can extend one superclass, but it can implement many interfaces. So for example, we might see a declaration such as:

```
public class Circle extends Figure implements FigureGeometry, Comparable
```

`Circle` inherits methods and data from the `Figure` class, and must implement any abstract classes defined in the `FigureGeometry` and `Comparable` interfaces. The prime benefit of this is that objects of type `Circle` can be used as if they were objects of type `Figure`, `FigureGeometry`, or `Comparable`.

To provide a generic type mechanism We can design and build ADTs to help us organize data of a specific type. For example, in Chapter 3 we implement an ADT that provides a list of strings. This ADT, and any ADT, would be more reusable if we did not limit it to a specific contained type, in this case, strings. It would be better to have an ADT that lets us manipulate lists of anything. Then, at our discretion, we could use it for lists of letters or lists of integers or whatever. We call such ADTs *generic structures*. In the latter part of Chapter 3 you learn how to use the Java interface construct to provide generic structures.

Arrays

Classes provide programmers a way to collect into one construct several different attributes of an entity and refer to those attributes by name. Many problems, however, have so many components that it is difficult to process them if each one must have a unique name. An array—the last of Java's built-in types—is the data type that allows us to solve problems of this kind. We are sure that you have studied and used arrays in your previous work. Here we revisit arrays, using the terminology and views established in this chapter.

In general terminology, an array differs from a class in three fundamental ways:

1. An array is a homogeneous structure (all components in the structure are of the same data type), whereas classes are heterogeneous structures (their components may be of different types).
2. A component of an array is accessed by its position in the structure, whereas a component of a class is accessed by an identifier (the name).
3. Because array components are accessed by position, an array is a structured composite type.

Logical Level

A one-dimensional array is a structured composite data type made up of a finite, fixed-size collection of ordered homogeneous elements to which there is direct access. *Finite* indicates that there is a last element. *Fixed size* means that the size of the array must be known at compile time, but it doesn't mean that all of the slots in the array must contain meaningful values. *Ordered* means that there is a first element, a second element, and so on. (It is the relative position of the elements that is ordered, not necessarily the values stored there.) Because the elements in an array must all be of the same type, they are physically *homogeneous*; that is, they are all of the same data type. In general, it is desirable for the array elements to be logically homogeneous as well—that is, for all of the elements to have the same purpose. (If we kept a list of numbers in an array of integers, with the length of the list—an integer—kept in the first array slot, the array elements would be physically, but not logically, homogeneous.)

The component selection mechanism of an array is *direct access*, which means we can access any element directly, without first accessing the preceding elements. The desired element is specified using an index, which gives its relative position in the collection.

The semantics (meaning) of the component selector is "Locate the element associated with the index expression in the collection of elements identified by the array

name." Suppose, for example, we are using an array of integers, called `numbers`, with 10 elements. The component selector can be used in two ways:

1. It can be used to specify a place into which a value is to be copied, such as
   `numbers[2] = 5;`
2. It can be used to specify a place from which a value is to be retrieved, such as
   `value = numbers[4];`

If the component selector is used on the left-hand side of the assignment statement, it is being used as a transformer: the storage structure is changing. If the component selector is used on the right-hand side of the assignment statement, it is being used as an observer: It returns the value stored in a place in the array without changing it. Declaring an array and accessing individual array elements are operations predefined in nearly all high-level programming languages.

In addition to component selection, there is one other "operation" available for our arrays. In Java, each array that is instantiated has a public instance variable, called `length`, associated with it that contains the number of components in the array. You access the variable using the same syntax you use to invoked object methods: You use the name of the object followed by a period, followed by the name of the instance variable. For the `numbers` example, the expression:

```
numbers.length
```

would have the value 10.

Application Level

A one-dimensional array is the natural structure for the storage of lists of like data elements. Some examples are grocery lists, price lists, lists of phone numbers, and lists of student records. You have probably used one-dimensional arrays in similar ways in some of your programs.

The declaration of a one-dimensional array is similar to the declaration of a simple variable (a variable of a primitive data type), with one exception. You must indicate that it is an array by putting square brackets next to the type:

```
int[] numbers;
```

Alternately, the brackets can go next to the name of the array:

```
int numbers[];
```

We prefer the former approach to declaring arrays, since it is more consistent with the way we declare other variables in Java.

Arrays are handled by reference, just like classes. This means they need to be treated carefully, just like classes, in terms of aliases, comparison, and their use as

parameters. And like classes, in addition to being declared, an array must be instantiated. At instantiation you specify how large the array is to be:

```
numbers = new int[10];
```

As with objects, you can both declare and instantiate arrays with a single command:

```
int[] numbers = new int[10];
```

A few more questions you may have about arrays:

- What are the initial values in an array instantiated by using `new`? If the array components are primitive types, they are set to their default value. If the array components are reference types, the components are set to `null`.
- Can you provide initial values for an array? An alternate way to create an array is with an initializer list. For example, the following line of code declares, instantiates, and initializes the array `numbers`:
  ```
  int numbers[] = {5, 32, -23, 57, 1, 0, 27, 13, 32, 32};
  ```
- What happens if we try to execute the statement
  ```
  numbers[n] = value;
  ```
  when `n` is less than 0 or when `n` is greater than 9? The result is that a memory location outside the array would be accessed, which causes an error. This error is called an *out-of-bounds error*. Some languages, C++ for instance, do not check for this error, but Java does. If your program attempts to use an index that is not within the bounds of the array, an `ArrayIndexOutOfBoundsException` is thrown. Rather than trying to catch this error, you should write your code to prevent it. Exceptions are covered in more detail later in this chapter.

Type Hierarchies

In all of our examples of composite types, notably with records and arrays, we have used composite types whose components have been primitive types. We looked at a record, `Circle`, that had four primitive type fields, and an array, `numbers`, of the primitive `int` type. We used this approach to simplify the discussion; it allowed us to concentrate on the structuring mechanism without introducing unnecessary complications. In practice, however, the components of these types can be any Java type or class: built-in primitive types like we have used so far, built-in nonprimitive types, or even user-defined types.

In this subsection we introduce several ways of combining our built-in types and classes into versatile hierarchies.

Aggregate Objects

The instance variables of our objects can themselves be references to objects. In fact, this is a very common approach to the organization of objects in our world. For example, a

page object might be part of a book object that is part of a shelf that is part of a library, and so on.

Consider the example from the section entitled The Class Type, of a class modeling a circle that includes variables for horizontal and vertical positions. Instead of these two instance variables, we could have defined a `Point` class to model a point in two-dimensional space, as follows:

```
public class Point
{
  public int xValue;
  public int yValue;
}
```

Then, we could define a new circle class as:

```
public class NewCircle
{
  public Point location;
  public float radius;
  public boolean solid;
}
```

Aggregate object An object whose class definition includes variables that are themselves references to classes.

An object of class `NewCircle` has three instance variables, one of which is an object of class `Point`, which in turn has two instance variables. An object, like `NewCircle`, made up of other objects is called an aggregate object. We call the relationship between the classes `NewCircle` and `Point` a "has a" relationship, as in "a `NewCircle` object *has a* `Point` object" as an instance variable. The *has a* relationship is depicted in UML with a diamond on the composite end of a link between the two classes, as shown in Figure 2.9.

When we instantiate and initialize an object of type `NewCircle`, we must remember to also instantiate the composite `Point` object. For example, to create a solid circle at position <5, 7> with a radius of 2.5, we would have to code:

```
NewCircle myNewCircle = new NewCircle();
myNewCircle.location = new Point();
myNewCircle.location.xValue = 5;
myNewCircle.location.yValue = 3;
myNewCircle.radius = 2.5f;
myNewCircle.solid = true;
```

Although this is a syntactically correct approach to structuring data, the use of composite objects in this fashion quickly becomes tedious for the application programmer. It is

Figure 2.9 *UML diagram showing has a relationship*

much easier if we define methods, such as a constructor method, to access and manipulate our objects. That is the approach we take below in the section on user-defined types, when we move from using classes as records to using classes to create true ADTs.

Arrays of Objects

Although arrays with atomic components are very common, many applications require a collection of composite objects. For example, a business may need a list of parts records or a teacher may need a list of students in a class. Arrays are ideal for these applications. We simply define an array whose components are objects.

Let's define an array of `NewCircle` objects. Declaring and creating the array of objects is exactly like declaring and creating an array in which the components are atomic types:

```
NewCircle[] allCircles = new NewCircle[10];
```

`allCircles` is an array that can hold ten references to `NewCircle` objects. What are the locations and radii of the circles? We don't know yet. The array of circles has been instantiated, but the `NewCircle` objects themselves have not. Another way of saying this is that `allCircles` is an array of references to `NewCircle` objects, which are set to `null` when the array is instantiated. The objects must be instantiated separately. The following code segment initializes the first and second circles. It assumes that a `NewCircle` object `myNewCircle` has been instantiated and initialized as described in the preceding section, Aggregate Objects.

```
NewCircle[] allCircles = new NewCircle[10];
allCircles[0] = new NewCircle();
allCircles[0] = myNewCircle;
allCircles[1] = new NewCircle();
allCircles[1].location = new Point();
allCircles[1].location.xValue = 6;
allCircles[1].location.yValue = 6;
allCircles[1].radius = 1.3f;
allCircles[1].solid = false;
```

Normally an array like this would be initialized using a *for* loop and a constructor method, but we used the above approach so that we could demonstrate several of the subtleties of the construct. Figure 2.10 shows what the array looks like with values in it.

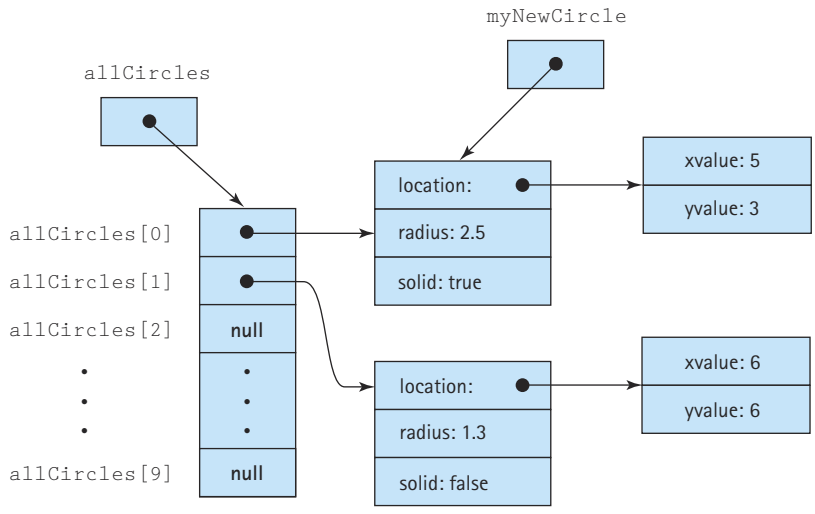


Figure 2.10 *The* `allCircles` *array*

Study the code above and Figure 2.10. In particular, notice how we must instantiate each element in the array with the new command. Also, notice that `myNewCircle` and `allCircles[0]` are aliases.

Keep in mind that an array name with no brackets is the array object. An array name with brackets is a component. The component can be manipulated just like any other variable of that type. The following table demonstrates these relationships:

Expression	Class/Type
`allCircles`	An array
`allCircles[1]`	A `NewCircle`
`allCircles[1].location`	A `Point`
`allCircles[1].location.xValue`	An `int`

Two-Dimensional Arrays

A one-dimensional array is used to represent items in a list or a sequence of values. A two-dimensional array is used to represent items in a table with rows and columns, provided each item in the table is of the same type or class. A component in a two-dimensional array is accessed by specifying the row and column indexes of the item in the array. This is a familiar task. For example, if you want to find a street on a map, you look up the street name on the back of the map to find the coordinates of the street, usually a number and a letter. The number specifies a row, and the letter specifies a column. You find the street where the row and column meet.

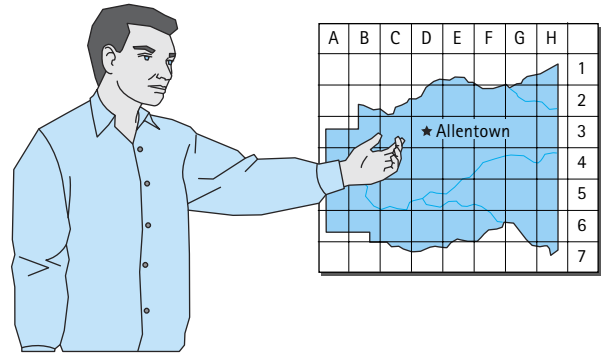


Figure 2.11 shows a two-dimensional array with 100 rows and 9 columns. The rows are accessed by an integer ranging from 0 through 99; the columns are accessed by an integer ranging from 0 through 8. Each component is accessed by a row–column pair—for example, `[0][5]`.

A two-dimensional array variable is declared in exactly the same way as a one-dimensional array variable, except that there are two pairs of brackets. A two-dimensional array object is instantiated in exactly the same way, except that sizes must be specified for two dimensions.

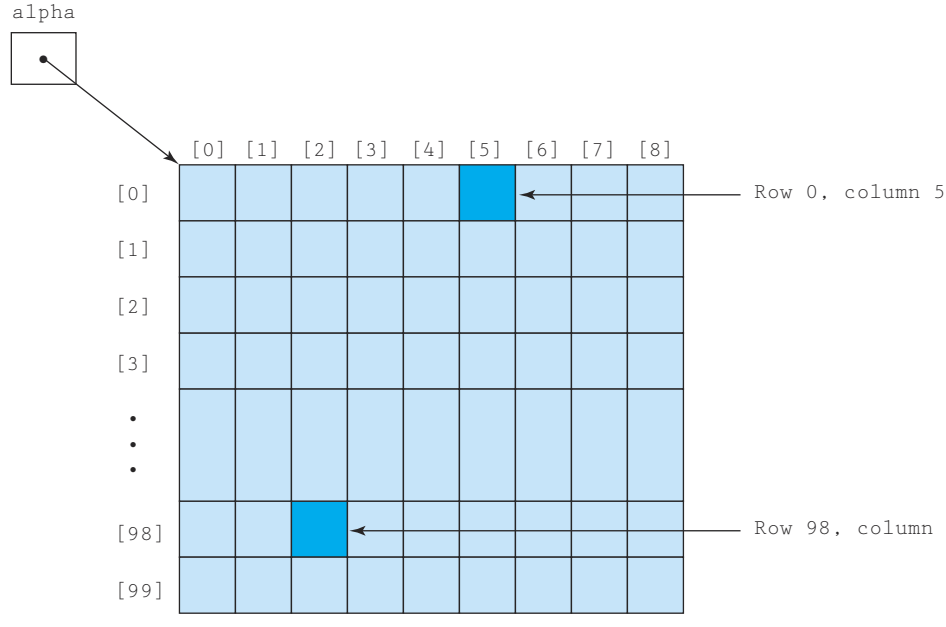


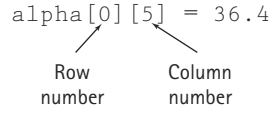
Figure 2.11 *alpha array*

The following code fragment would create the array shown in Figure 2.11, where the data in the table are floating-point numbers.

```
double[][] alpha;
alpha = new double[100][9];
```

The first dimension specifies the number of rows, and the second dimension specifies the number of columns.

To access an individual component of the `alpha` array, two expressions (one for each dimension) are used to specify its position. We place each expression in its own pair of brackets next to the name of the array:



Note that `alpha.length` would give the number of rows in the array. To obtain the number of columns in a row of an array, we access the `length` attribute for the specific row. For example, the statement

```
rowLength = alpha[30].length;
```

stores the length of row 30 of the array `alpha`, which is 9, into the `int` variable `rowLength`.

The moral here is that in Java each row of a two-dimensional array is itself a one-dimensional array. Many programming languages directly support two-dimensional arrays; Java doesn't. In Java, a two-dimensional array is an array of references to array objects. Because of the way that Java handles two-dimensional arrays, the drawing in Figure 2.11 is not quite accurate. Figure 2.12 shows how Java actually implements the array `alpha`. From the Java programmer's perspective, however, the two views are synonymous in the majority of applications.

Multilevel Hierarchies

We have just looked at various ways of combining Java's built-in type mechanisms to create composite objects, arrays of objects, and two-dimensional arrays. We do not have to stop there. We can continue along these lines to create whatever sort of structure best matches our data. Classes can have arrays as variables, aggregate objects can be made from other aggregate objects, and we can create arrays of three, four, or more dimensions.

Consider, for example, how a programmer might structure data that represents students for a professor's grading program. This professor grades each test with both a numerical grade and a letter grade. Therefore, the programmer decides to represent a test as a record, called `test`, with two fields: `score` of type `int` and `grade` of type `char`. Each student takes a sequence of tests—these are represented by an array of `test` called `marks`. A student also has a name and an attendance record. So a student

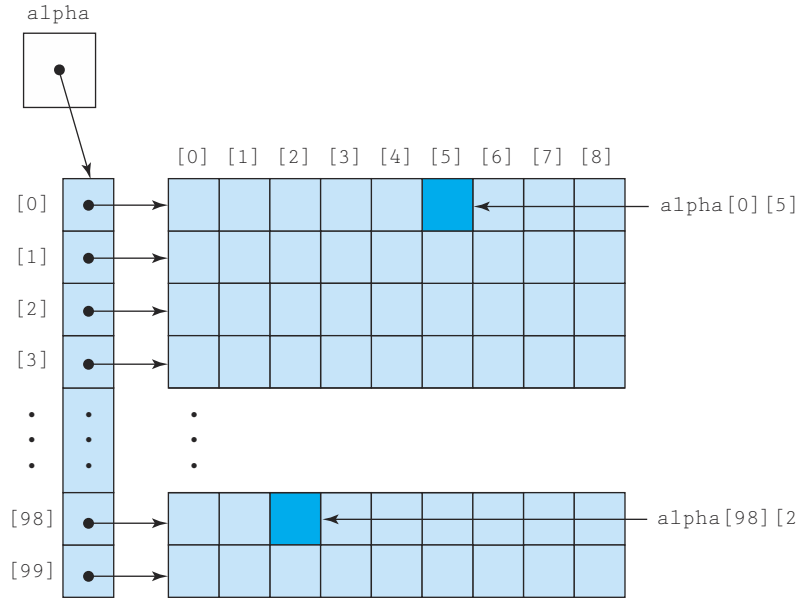


Figure 2.12 *Java implementation of the* `alpha` *array*

could be represented by a record with three fields: `name` of type `String`, `marks` of type array of `test`, and `attendance` of type `int`. Since the professor has many students in a course, the programmer creates another array, called `course`, that is an array of `student`. Wow! We have an array of records of three fields, one of which is itself an array of records of two fields. See Figure 2.13 for a logical view of this multilevel structure.

The idea is to use the built-in typing mechanisms to model the real world structure of the data. This makes it easier for us to organize our processing of the data.

In the next section we look at how we can extend Java's built-in types by encapsulating composite types with programmer-defined methods, to simplify their access and manipulation. When we do this we are creating our own ADTs.

2.3 Class-Based Types

The class construct sits at the center of the Java programming world. In the previous section, you learned how the Java class could be used to structure data into records. As we stated then, that is not a proper use of the class construct when practicing object-oriented design. In this section, you learn how to use classes to implement ADTs. This is the correct way to use the class construct.

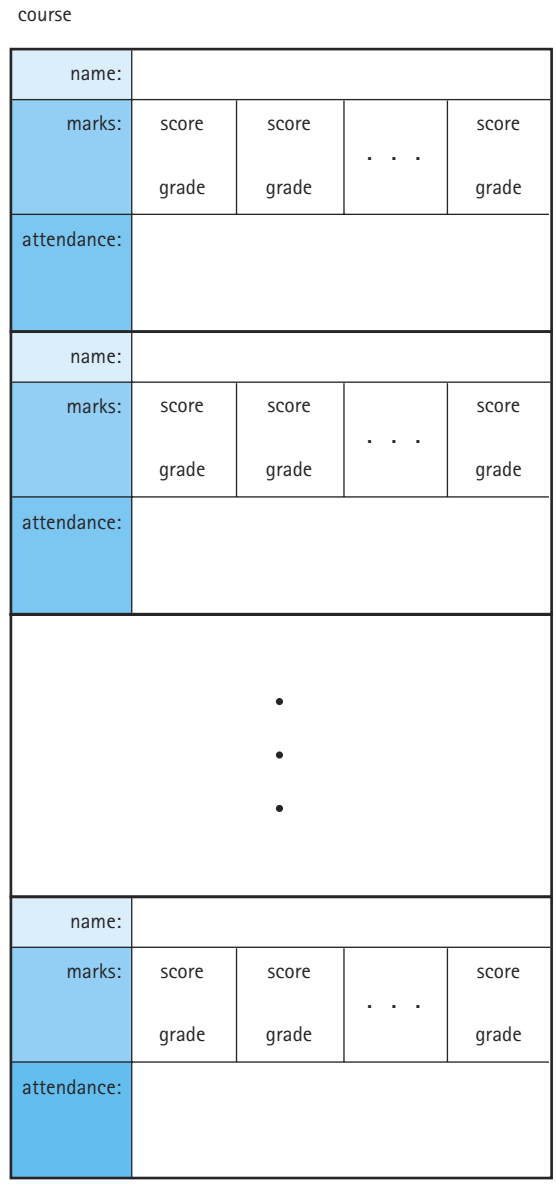


Figure 2.13 *Logical view of array of student records*

Meaning of "Type"

The Java language specification reserves the word *type* to mean those abstract data types (ADTs) that are built into the language, such as `int`, `double`, `char`, `array`, and `class`. Every ADT that we design and implement as a class in Java is considered by the language to be a member of the same type, which is the Java `class` type.

More generally, the word *type* is often used to refer to an ADT and its implementation in whatever programming language is being used. Thus, there is a potential for some minor confusion with respect to Java's use of the term and the use of the term in general. Wherever we use the word *type*, and the context of the usage does not clarify the meaning, we modify the term to provide clarification. Thus, we may use "Java type," "built-in type," or "primitive type" to indicate that we are using the term in the strict Java sense. Elsewhere, we use the term in its more general sense, for example, to refer to the implementation of a programmer defined ADT. Thus, we may refer to the `Date` type or the `Circle` type.

Using Classes in Our Programs

Once a programmer has defined a class, objects of the class type can be declared, instantiated, and used in many other classes. For purposes of this discussion, we call the class being used the *tool class*, and the class using it the *client class*. The client class could be an *application*, that is, a class with a `main` method that would be executed when we invoke the Java interpreter. For the client class to use the tool class, the definition of the tool class must be visible to the Java compiler/interpreter, when the client class is compiled or interpreted. There are several ways you can ensure this:

Inner class A class defined as a member of another class

Package A set of related classes, grouped together to provide efficient access and use

1. Insert the tool class code directly into the client class file. In this case, we call the tool class an inner class. There are some situations, especially with respect to dynamic event handling, where inner classes provide an elegant solution to difficult problems. Usually, however, their use is too restrictive.
2. Computer systems that support Java have a well-defined set of subdirectories to search when a Java class is needed. Usually an environment variable called `ClassPath` defines this set of subdirectories. Place the tool class file in one of these subdirectories.
3. The Java package construct is used by programmers to collect into a single unit a group of related classes. Put the tool class in a package and import the package into the client that uses it. Note that the compiler/interpreter must be able to find the package, so it must be located in an appropriate subdirectory on the `ClassPath`. The feature section below describes the details of using Java packages.

Java Packages

Java lets us group related classes together into a unit called a package. Packages provide several advantages:

- They let us organize our files.
- They can be compiled separately and imported into our programs.
- They make it easier for programs to use common class files.
- They help us avoid naming conflicts (two classes can have the same name if they are in different packages).

Package Syntax

The syntax for a package is extremely simple. All we have to do is to specify the package name at the start of the file containing the class. The first noncomment nonblank line of the file must contain the keyword `package` followed by an identifier and a semicolon. By convention, Java programmers start a package identifier with a lowercase letter to distinguish package names from class names:

```
package someName;
```

After this we can write import declarations, to make the contents of other packages available to the classes inside the package we are defining, and then one or more declarations of classes. Java calls this file a *compilation unit*. The classes defined in the file are members of the package. Note that the imported classes are not members of the package.

Although we can declare multiple classes in a compilation unit, only one of them can be declared public. The others are hidden from the world outside the package. (We investigate visibility topics later in this section.) If a compilation unit can hold at most one public class, how do we create packages with multiple public classes? We have to use multiple compilation units, as we describe next.

Packages with Multiple Compilation Units

Each Java compilation unit is stored in its own file. The Java system identifies the file using a combination of the package name and the name of the public class in the compilation unit. Java restricts us to having a single public class in a file so that it can use file names to locate all public classes. Thus, a package with multiple public classes must be implemented with multiple compilation units, each in a separate file.

Using multiple compilation units has the further advantage that it provides us with more flexibility in developing the classes of a package. Team programming projects would be very cumbersome if Java made multiple programmers share a single package file.

We split a package among multiple files simply by placing its members into separate compilation units with the same package name. For example, we can create one file containing the following code (the ... between the braces represents the code for each class):

```
package someName;
public class One{ ...  }
class Two{ ...  }
```

and a second file containing:

```
package someName;
class Three{ ...  }
public class Four{ ...  }
```

with the result that the package `someName` contains four classes. Two of the classes, `One` and `Four` are public, and so are available to be imported by application code. The two file names must match the two public class names; thus the files must be named `One.java` and `Four.java`.

Many programmers simply place every class in its own compilation unit. Others gather the nonpublic classes into one unit, separate from the public classes. How you organize your packages is up to you, but you should be consistent to make it easy to find a specific member of a package among all of its files.

How does the Java compiler manage to find these pieces and put them together? The answer is that it requires that all compilation unit files for a package be kept in a single directory or folder that matches the name of the package. For our preceding example, a Java system would store the source code in files called `One.java` and `Four.java`, both in a directory called `someName`.

The import Statement

In order to access the contents of a package from within a program, you must import it into your program. You can use either of the following forms of import statements:

```
import packagename.*;
import packagename.Classname;
```

An import declaration begins with the keyword `import`, the name of a package and a dot (period). Following the period you can either write the name of a class in the package, or an asterisk (`*`). The declaration ends with a semicolon. If you know that you want to use exactly one class in a particular package, then you can simply give its name in the import declaration. More often, however, you want to use more than one of the classes in a package, and the asterisk is a shorthand notation to the compiler that says, "Import whatever classes from this package that this program uses."

Packages and Subdirectories

Many computer platforms use a hierarchical file system. The Java package rules are defined to work seamlessly with such systems. Java package names may also be hierarchical; they may contain periods separating different parts of the name, for example, `ch03.stringLists`. In such a case, the package files must be placed underneath a set of subdirectories that match the separate parts of the package name. Following the same example, the package files should be placed in a directory named `stringLists` that is a subdirectory of a directory named `ch03`. You can import the entire package into your program with the following statement:

```
import ch03.stringLists.*;
```

As long as the directory that contains the `ch03` directory is on the `ClassPath` of your system, the compiler will be able to find the package you requested. The compiler automatically looks in all the directories listed in the `ClassPath`. In this case it will actually look in the `ClassPath` directories for a subdirectory named `ch03` that contains a subdirectory named `stringLists`, and upon finding such a subdirectory, it will import all of the members of the `ch03.stringLists` package that it finds there.

Many of the files created to support this textbook are organized into packages. They are organized exactly as described above and can be found on our web site. All the files are found in a directory named `bookFiles`. It contains a separate subdirectory for each chapter of the book: `ch01`, `ch02`, ..., `ch10`. Where packages are used, you will find the corresponding subdirectories underneath the chapter subdirectories. For example, the `ch03` subdirectory does indeed contain a subdirectory named `stringLists` that contains four files that define Java classes related to a string list ADT. Each of the class files begins with the statement

```
package ch03.stringLists;
```

Thus, they are all in the `ch03.stringLists` package. If you write a program that needs to use these files you simply need to import the package into your program and make sure the parent directory of the `ch03` directory, i.e., the `bookFiles` directory, is included in your computer's `ClassPath`.

We suggest that you copy the entire `bookFiles` directory to your computer's hard drive, ensuring easy access to all the book's files and maintaining the crucial subdirectory structure required by the packages. Also, make sure you extend your computer's `ClassPath` to include your new `bookFiles` dirctory. See the Preface for more information.

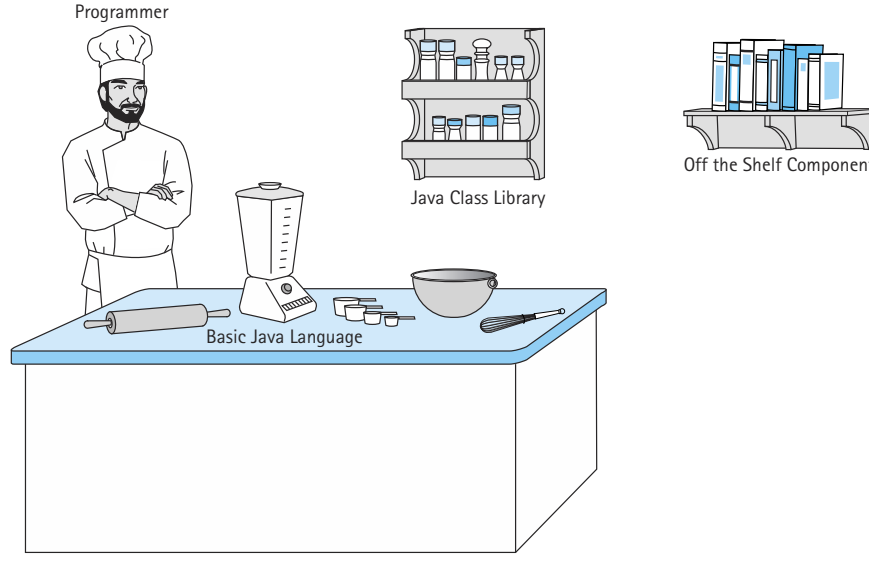
Sources for Classes

Java programs are built using a combination of the basic language and pre-existing classes. In effect, the pre-existing classes act as extensions to the basic language; this extended Java language is large, complex, robust, powerful and ever changing. Java programmers should never stop learning about the nuances of the extended language—an exciting prospect for those who like an intellectual challenge.

When designing a Java-based system to solve a problem, we first determine what classes are needed. Next we determine if any of these classes already exist; and if not, we try to discover classes that do exist that can be used to build the needed classes. Additionally, we often create our own classes, "helper" classes that are used to build the needed classes.

Where do the classes come from? There are three sources:

1. The Java Class Library—The Java language is bundled with a class library that includes hundreds of useful classes. We look at the library in a subsection below.
2. Build your own—Suppose you determine that a certain class would be useful to aid in solving your programming problem, but the class does not exist. Therefore, you create the needed class, possibly using pre-existing classes in the process. The new class becomes part of the extended language, and can be used on future projects. We look at how to build our own classes in a later section, and throughout the rest of the textbook.
3. Off the shelf—Software components, such as classes or packages of classes, which are obtained from third party sources, are called off-the-shelf components. When they are bought, we call them "commercial off-the-shelf" components, or COTS. Java components can be bought from software shops, or even found free on the web. When you obtain software, or anything else, from the web for your own use, you should make sure you are not violating a copyright. You also need to use care in determining that free components work correctly and do not contain viruses or other code that could cause problems.



As our study of data structures, abstract data types, and Java continues, we sometimes investigate how to build a class that mirrors the functionality of a pre-existing class, for example a class in the Java Class Library. There are two reasons we may do

this: convenience and computer science content. It may be that the class provides a good, convenient example of a language construct or programming approach—for example, our use of the `Date` example throughout the first two chapters—even though the library provides ways of creating and using Date objects. Alternately, it may be that the study of the class is crucial to the content of this textbook—classic data structures. For example, in Chapter 4 we study how to implement a Stack ADT, even though a Stack ADT is provided in the library. Understanding the possible implementations of stacks, and the ramifications of implementation choices, is considered crucial for serious students of computing.

There are other reasons that a programmer might want to create his or her own class that mimic the functionality of a library class: simplicity and control. The Java developers designed library classes to provide robust functionality. Robustness is an important quality for library classes. Sometimes, however, the robustness of a class equates to complexity or inefficiency. Additionally, you must remember that the Java Class Library is not a static construct. The changes to the library are usually in the form of enhancements, but there have also been cases where features of the library have been deprecated. A deprecated feature is one that may not be supported in future versions of the library. Deprecation acts as a warning to programmers—use this construct at your own risk; it works now, but might not work later!

Deprecated A Java construct is deprecated when the Java developers have decided that the construct might not be supported in future releases of the language; use of deprecated features is discouraged.

Consider the history of dates in Java. In the original public release of Java, JDK 1.0 in 1995, the library included a `Date` class that allowed a programmer to represent dates and times. This class could be used to specify and manipulate a date/time in two forms: the number of milliseconds between the date/time and January 1, 1970, midnight, or by using discrete attributes of the date/time, such as month, day, year, hour. As you can imagine, for most purposes the latter form was easier to use. Nevertheless, the latter form of use was deprecated with the release of JDK 1.1 in 1996 because it did not support Java's goal of internationalization. Although many countries use the Gregorian calendar that the `Date` class is based on, there are other calendars in use around the world, for example the Chinese calendar.

The `Calendar` class was introduced in Java 1.1 to support all kinds of calendars. It provided features to replace the deprecated functionality of `Date`. The `Calendar` class is well designed and very useful; but it is not trivial to use. The `Calendar` class cannot be directly instantiated; programmers must use its `getInstance` method to obtain a local instance of a calendar, and instantiate this local instance as a subclass of `Calendar`. To use the "standard" solar calendar, with years numbered from the birth of Christ, a programmer would use the `GregorianCalendar` subclass of `Calendar`. The `GregorianCalendar` class exports 28 methods and defines 42 constants for use with the methods of the class.

Considering all of this, it is no wonder that programmers who need a simple date class—perhaps one that allows a month, day, and year to be passed to the constructor, provides three simple observer methods, and provides methods to increment a date and compare two dates—might decide to implement their own class.

The Java Class Library

Programming with an object-oriented language depends heavily on the use of classes from the language's standard library. The Java standard class library includes over 70 packages and subpackages, with hundreds of classes and interfaces, and thousands of exported methods and constants. It is not our goal in this textbook to teach the standard library. However, we do encourage the reader to begin to learn about the library, and to continue studying the library.

Sun Microsystems, Inc., the developers of Java, maintains a public web site[1] where they have provided extensive documentation about the class libraries. The list below briefly describes some of the prominent packages and subpackages found on the Sun site. Visit their site for more information. In this subsection, we review some of the most important classes, especially with respect to the goals of this textbook. Throughout the text, as we reach places where we need to use library constructs in a new way, we expand on this coverage.

Some Important Library Packages

`java.awt`	(Abstract Windowing Toolkit) Contains tools for creating user interfaces, graphics, and images.
`java.awt.event`	Provides interfaces and classes for handling the different types of events created by AWT components.
`java.io`	System input and output through data streams, serialization, and the file system.
`java.lang`	Provides basic classes for use in creating Java programs.
`java.math`	Provides classes for performing mathematical operations.
`java.text`	Provides classes and interfaces for handling text, dates, numbers, and messages.
`java.util`	Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).
`java.util.jar`	Provides classes for reading and writing the JAR (Java ARchive) file format, which is based on the standard ZIP file format.

Some Useful General Library Classes

As we are studying data structures with the Java language, we use various utility classes that are available in the Java library. In this subsection, we introduce some of these classes.

---

[1] http://java.sun.com/j2se/1.3/docs/api/index.html

The System class The `System` class is part of the `java.lang` package. All of the `System` class's methods and variables are class methods and variables. They are defined to be static—they are unique to the class, rather than to objects of the class. We simply use the `System` class methods and variables directly in our programs; we access them through the class name rather than through the name of an instantiated object. For example, in the `TestCircle` program listed above in Section 2.2, we used the `System` variable `out` as a destination for our output:

```
System.out.println("c1:    " + c1);
```

We can also use the `System` class to obtain current system properties, such as the amount of available memory.

The Random class The `Random` class is part of the `util` package. Programmers use it to generate a list of random numbers. Random numbers are useful when creating simulations, or models of real-world situations, with our programs. We use the `Random` class in Chapter 10 to generate lists of random numbers for sorting.

The DecimalFormat class The `DecimalFormat` class is part of the `java.text` package. To use it a programmer calls one of its constructors to define a format pattern. Then this instance can be used to format numbers for output. In Chapter 4 we use the `DecimalFormat` class to format numbers so that output columns line up nicely.

The Throwable and Exception Classes We introduced the concept of exceptions in Chapter 1. Recall that an exception is associated with an unusual, often unpredictable event, detectable by software or hardware, that requires special processing. One system unit raises or throws an exception, and another unit catches the exception and processes it. Processing an exception is also called handling the exception.

Throw an exception Interrupt the normal processing of a program to raise an exception that must be handled or rethrown by the surrounding block of code

Catch an exception Code that is executed to handle a thrown exception is said to catch the exception

When a part of a Java system determines that an exception has occurred, it "announces" the exception using the Java `throw` statement. This can occur within the Java interpreter, within a library method, or within our own code. (We discuss how to define, and how to determine when to throw our own exceptions, in the section below about building our own ADTs. For now, we look at predefined exceptions.) When an exception is thrown, it must either be caught and handled by the surrounding block of code, or thrown again to the next outer block of code. If an exception is thrown all the way out of a method, it propagates to the calling method. An exception that is continually thrown until it makes it all the way up the chain of calling methods and is thrown by the `main` method to the Java interpreter, is handled by the interpreter: An error message is printed along with some system information (a system stack trace) and the program exits.

All exceptions in Java are subclasses of the `java.lang.Throwable` class. Only objects or instances of this class (or subclasses of this class) are thrown within a Java system. The `Throwable` class provides several methods related to exceptions, notably the `getMessage` method that returns the error message associated with the `Throwable` object, and the `printStackTrace` method that prints a trace of the sequence of system calls that led to the `throw` statement.

The `Throwable` class has two standard subclasses: `java.lang.Error` and `java.lang.Exception`. The former is used for defining catastrophic exceptional situations that are best handled by simple program termination. We are concerned with the latter subclass, the `Exception` class, which is used for defining exceptional situations from which we may be able to recover. The `Exception` class extends the `Throwable` class with two methods, both constructors:

Method Name	Parameter Type	Returns	Operation Performed
`Exception`	(none)	`Exception`	Constructs an exception with no specified message.
`Exception`	`String`	`Exception`	Constructs an exception with the specified message.

Exceptions are defined by extending the `Exception` class. If you look at the Java library information you see dozens of predefined subclasses of the `Exception` class, each of which might also have many subclasses. The result is that there are hundreds of exceptions defined in the Java library.

Let's look at a few examples of throwing and handling predefined exceptions.

Review the `IncDate` test driver program, `IDTestDriver`, developed at the end of Chapter 1. Notice the heading of the program's `main` method:

```
public static void main(String[] args) throws IOException
```

As you can see, in the declaration of the `main` method we have told the system that this method can throw the predefined `IOException` exception. Where would `IOException` be raised in the program? This program uses the `readLine` method defined in the `BufferedReader` class. A quick look at the documentation of the `readLine` method shows that it throws an `IOException` "if an I/O error occurs." Since it is possible for that exception to be thrown by the `readLine` method, the surrounding code (the `main` method), must either catch and handle the exception, or throw the exception. In this case, we have decided to just throw the exception out to the interpreter, which would terminate the program. Note that this is a perfectly valid option; in fact, if there is not enough information to properly handle an exception at one level of a program, the best approach is to throw the exception out to the next level, where it may be handled more properly.

If we decided to handle the exception within the test driver program itself we would surround the section of the program where the exception can be raised with a *try-catch* statement. For example:

```
try
{
  month = Integer.parseInt(dataFile.readLine());
  day = Integer.parseInt(dataFile.readLine());
  year = Integer.parseInt(dataFile.readLine());
}
catch (IOException readExcp)
{
  outFile.println("There was trouble reading in month, day, year.");
  outFile.println("Exception: " + readExcp.getMessage());
  System.exit();
}
```

Now, if the `IOException` exception is raised by any of the `readLine` methods within the *try* block, it is handled by the code in the *catch* block. Notice the rather unusual syntax of the *catch* statement:

```
catch (ExceptionClassName varName)
```

If the exception class referenced in the *catch* statement is thrown by any of the statements in the *try* block, the *catch* statement defines a new object of that exception class, and that object becomes equated with the thrown exception. So in this example, the variable `readExcp` represents the exception that is caught. Because `readExp` is an instantiation of a subclass of `Throwable`, it has a `getMessage` method. In the *catch* block we can use `readExcp.getMessage()` to print the message associated with the exception.

In this example, we are handling the exception by printing our own brief error message, then printing the error message associated with the exception, and then terminating the program. Realistically, there is not much more we can do in this situation. Since this is essentially the same action the interpreter does for us anyway, it is probably better to just throw the exception. Besides, as we explained when we developed the test driver program, it is not important that the test driver be robust, since we are only using it to test another class; the test driver is not delivered to a customer.

There are some other nuances involved with handling predefined exceptions—for example, the use of Java's `finally` clause, and the option of handling and still rethrowing the exception. It could quickly become confusing if we tried to cover all of these topics at once, so we put off a discussion of other options until we reach an example that requires their use.

One last note about predefined exceptions. The `java.lang.RunTimeException` class is treated uniquely by the Java environment. Exceptions of this class are thrown during the normal operation of the Java Virtual Machine when a standard run-time program error occurs. Examples of run-time errors are division by zero and array-index-out-of-bounds situations. Since run-time exceptions can happen in virtually any method or segment of code, we are not required to explicitly handle these exceptions. If it were

required, our programs would become unreadable because of all the necessary *try*, *catch*, and *throw* statements. These exceptions are classified as unchecked exceptions.

Unchecked exception An exception of the `RunTimeException` class, it does not have to be explicitly handled by the method within which it might be raised.

Wrapper class A Java class that wraps a primitive type, letting it be manipulated as an object, and providing some useful utility methods related to the type.

Wrappers There are situations where a Java programmer wants to use a variable of class `Object` to reference many different kinds of objects. This is possible, since `Object` is a superclass of all other classes. This feature provides a powerful tool; however, it does suffer from one limitation—the variable of class `Object` cannot reference primitive type values, since the primitive types are not objects. To resolve this deficiency, the Java Class Library includes a wrapper class for each of the primitive types. To store a primitive value in the `Object` variable, the programmer first "wraps" it in the appropriate wrapper class. These classes are known as wrapper classes since they literally wrap a primitive valued variable in an object's structure, as shown in Figure 2.14. The following table lists the primitive types and the built-in class to which each corresponds.

Primitive Type	Object Type
`boolean`	`Boolean`
`byte`	`Byte`
`char`	`Character`
`double`	`Double`
`float`	`Float`
`int`	`Integer`
`long`	`Long`
`short`	`Short`

As you can see, the general rule is that the class name is the same as the name of the built-in type, except that its first letter is capitalized. The two cases that differ are that the class corresponding to `int` is called `Integer` and the class corresponding to `char` is `Character`.

The wrapper classes are a part of the `java.lang` package.

In addition to allowing us to treat a primitive type as an object, the wrapper classes provide many useful conversion and utility methods related to their associated primitive type. For example, we used the `Integer` wrapper class method `parseInt` in the `IDTestDriver` program in Chapter 1:

```
month = Integer.parseInt(dataFile.readLine());
day = Integer.parseInt(dataFile.readLine());
year = Integer.parseInt(dataFile.readLine());
```

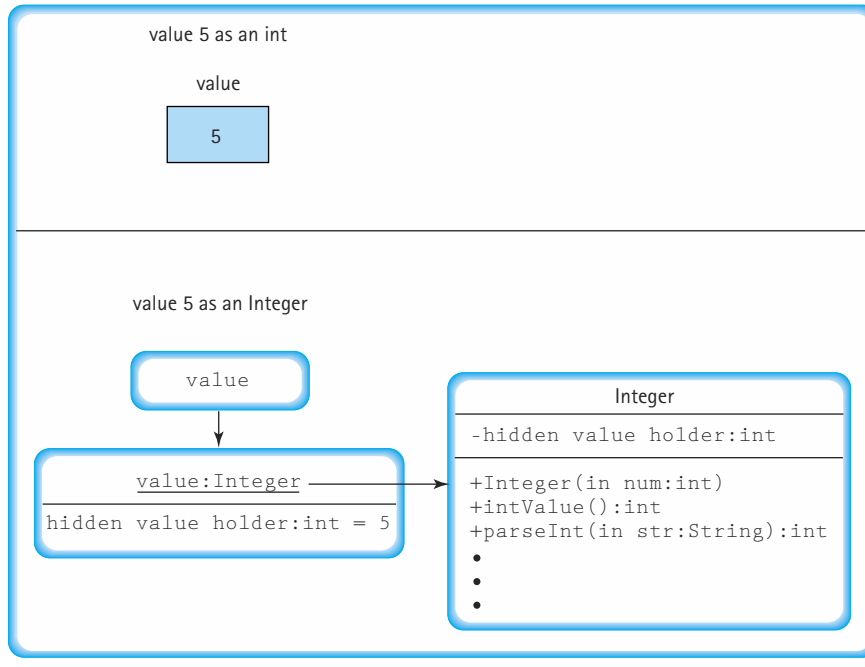


Figure 2.14 *The integer value 5 as an* `int` *variable* `value`*, and an* `Integer` *object* `value`

The `parseInt` method accepts a string as a parameter and transforms it into the corresponding integer. For example if it is passed the string "27" it returns the `int` value 27. Since the `BufferedReader datafile` we defined in the `IDTestDriver` program returns all input in the form of strings, the `parseInt` method allows us to transform the input into a more useful form.

Some Class Library ADTs

In addition to the utility classes just described, the Java Class Libraries include some ADTs that are pertinent to your study. A class provides an ADT if its basic purpose is to allow the programmer to store data in an abstract structure, hiding the implementation of the structure from the programmer but allowing the programmer to access the data through various exported methods.

In some sense the wrapper classes described at the end of the previous section provide ADTs—but the main way we use those classes is to access their general utility class methods, such as the `parseInt` method of `Integer`. Such methods are not really acting on an object; `parseInt` accepts a string parameter and returns a primitive `int`. It is invoked through the `Integer` class and not through a specific object of the class.

In this section we look at the Java Class Library ADTs `String` and `ArrayList` from the logical-level and application-level viewpoints. For array lists we also take a peek at the implementation level, since it is instructive to do so.

Strings The Java `String` class is part of the `java.lang` package. Remember that this package provides classes that are fundamental to the design of the Java programming language. In fact, this package is automatically imported into every Java program.

Strings are a fundamental building block for many programs. We have already been using them extensively in this textbook, for input and output to our programs and to indicate file names within our test drivers. We assume you have some experience using strings in your previous programming. Nevertheless, we provide a brief review of the Java `String` class here.

Logical Level The first thing we want to remind you about strings is that they are immutable. If an object doesn't have any methods that can change its state, it is immutable. A string is an immutable object; we can only retrieve its contents. There is no way to change a string object. We can only assign a new reference to a `String` variable. In other words, a `String` variable references a `String` object. Once created, that object cannot be changed; however, we can change the `String` variable so that it references a different `String` object.

Immutable object An object whose state cannot be changed once it is created

The Java `String` class provides operations for joining strings, copying portions of strings, changing the case of letters in strings, converting numbers to strings, and converting strings to numbers. Their use is straightforward and we leave it to you to review them. Notice that, due to the immutability of strings, any operation that appears to change a string, for example the `toUpperCase` method, actually returns a new `String` object rather than changing the current string. For example, if the string `objectnameB` has an associated value "Adam", the statement

```
nameA = nameB.toUpperCase();
```

creates a new `String` object with value "ADAM", assigns its reference to the `nameA` string variable, but leaves the `nameB` string variable and object unchanged.

You cannot compare strings using the relational operators. Syntactically, Java lets you write the comparisons for equality (==) and inequality (!=) between values of class `String`, but the comparison that this represents is not what you typically want. Since `String` is a reference type, when you compare two strings this way, Java checks to see that they have the same address. It does not check to see whether they contain the same sequence of characters.

Rather than using the relational operators, we compare strings with a set of value-returning instance methods that Java supplies as part of the `String` class. Because they

are instance methods, the method name is written following a `String` object, separated by a dot. The string that the method name is appended to is one of the strings in the comparison, and the string in the parameter list is the other. The two most useful comparison methods are summarized in the following table.

Method Name	Parameter Type	Returns	Operation Performed
`equals`	`String`	`boolean`	Tests for equality of string contents.
`compareTo`	`String`	`int`	Returns 0 if equal, a positive integer if the string in the parameter comes before the string associated with the method, and a negative integer if the parameter comes after it.

For example, if `lastName` is a `String` variable, you can use

```
lastName.equals("Olson")
```

to test whether `lastName` equals "Olson."

Application Level Since the use of strings in programs is so prevalent, the Java language provides a few shortcuts for using the `String` class that differentiate it from all the other classes in the Java library. We saw one of these in Chapter 1 when we noted how a `toString` method is automatically applied to an object that is being used as a string. Let's look at two more special conventions for strings, string literals, and the concatenation operator.

String Literals Just as Java provides literals for all of its primitive types (for example `-154` is a literal of type `int` and `true` is a literal of type `boolean`), it provides a literal string mechanism. To indicate a literal string, you simply enclose the sequence of characters between double quotation marks. For example:

```
"this is a literal string"
```

A literal string actually represents an object of class `String`. Enclosing a sequence of characters in the double quotation marks is equivalent to declaring and instantiating a new `String` object. Therefore, the following two code sequences are equivalent:

```
String myString;
myString = new String ("The Cat in the Hat");
-------------------------------------------
String myString = "The Cat in the Hat";
```

The String Concatenation Operator The `String` class exports a method `concat` that allows a programmer to concatenate two strings together to form a third string. However, this operation is so prevalent in Java programs that the language designers provide us with a shortcut, the + string operation. The result of concatenating two strings is a new string containing the characters from both strings. For example, given the statements

```
String first = "The Cat in the Hat";
String second = "Comes Back";
String third = first + second;
output.println(third);
```

the string "The Cat in the HatComes Back" appears in the `output` stream. Notice that the system does not automatically insert blanks between two concatenated strings.

Concatenation works only with values of type `String`. However, if we try to concatenate a value of one of Java's built-in types to a string, Java automatically converts the value into an equivalent string and performs the concatenation. In fact, we can concatenate an object of any class to a string; the system looks for the object's `toString` operation to transform the object into a string before the concatenation.

Array Lists The `ArrayList` class is part of the `java.util` package. The functionality of the `ArrayList` class is similar to that of the array. In fact, the array is the underlying implementation structure used in the class. In contrast to an array, however, an array list can grow and shrink; its size is not fixed for its lifetime.

The `ArrayList` class was added to the library with the release of Java 1.2. It provides essentially the same functionality as the original library's `Vector` class, with which you may be familiar from a previous course. However, the `Vector` class supports concurrent programming; that is, it supports programs that have more than one active thread. A *thread* is a flow of control in a program. Advanced Java programs can have multiple control flows that execute simultaneously and interact with each other. The support that is necessary to enable concurrent programming requires extra processing whenever a `Vector` method is invoked, even when we aren't using multiple threads. The extra processing makes the `Vector` class a poor choice for use with single-threaded programs, such as the programs of this textbook. For single-threaded programs, you should use the `ArrayList` class instead of the `Vector` class.

Logical Level We approach the logical view of array lists by comparing and contrasting them with arrays. Like an array, an array list is a structured composite data type, made up of a collection of ordered elements. As with an array, we can access an

element of an array list directly by specifying an index. However, arrays and array lists differ in many ways:

1. Arrays can be declared to hold data of a specific type; array lists hold variables of type `Object`. Therefore, *every* array list can hold virtually any type of data, even a primitive type if it is contained within a wrapper object.
2. An array remains at a fixed capacity throughout its lifetime; the capacities of array lists grow and shrink, depending on need.
3. An array has a length; an array list has a size, indicating how many objects it is currently holding, and a capacity, indicating how many elements the underlying implementation could hold without having to be increased.

The following table describes some of the interesting `ArrayList` operations.

Method Name	Parameter Type	Returns	Operation Performed
`ArrayList`	(none)		Constructs an empty array list of capacity 10.
`ArrayList`	`int`		Constructs an empty array list of the capacity indicated by the parameter.
`add`	`int`, `Object`	`void`	Inserts the specified `Object` at the specified position; shifts all subsequent elements to the right one place.
`add`	`Object`	`void`	Inserts the specified `Object` at the end.
`ensureCapacity`	`int`	`void`	Increases the capacity of the array list to at least the specified capacity, if it is currently less than the specified capacity.
`get`	`int`	`Object`	Returns the element at the specified position.
`isEmpty`	(none)	`boolean`	Returns `true` if the array list is empty, `false` otherwise.
`remove`	`int`	`Object`	Removes the element at the specified position, shifts all subsequent elements to the left one place, and returns the removed element.
`size`	(none)	`int`	Returns current size.
`trimToSize`	(none)	`void`	Trims the capacity of the array list to its size.

Implementation Level It is not necessary to peek at the underlying implementation of array lists in order to use them in our programs. Nevertheless, it is an instructive exercise, and helps us understand when to choose an array list structure over an array and vice versa.

We can imagine a Java array list consisting of an array and two integer variables that hold the capacity (length) and size (number of current elements) of the array. The underlying array is always "left-justified;" in other words, any empty slots are at higher indices then the slots being used.

It is easy to see how observer methods, like `get`, `isEmpty`, and `size` are implemented; the appropriate information is simply calculated and returned. But what about operations that change the contents of the array list; for example, the `add` operation? These are more interesting.

First, we consider a "standard" `add` operation, one that does not require a change in the size of an array list. Suppose we have an array list `letters` that we are using to hold characters. Suppose its capacity is 8 and it has a current size of 6. (See the "Before" section of Figure 2.15, which represents this situation. In the figure we follow several simplifying conventions: we show characters inside the array locations rather than show each of them as separate objects; we label the underlying array with the name of the array list.)

Now suppose we perform the operation

```
letters.add(2, 'X');
```

To make room for the addition of the character `'X'` at index 2, the underlying implementation would first copy the elements at positions 5 down to 2 into locations 6 down to 3. That frees location 2 so that the `'X'` can be copied into it. Additionally, the `size` variable would need to be updated. (See the "Processing" section of Figure 2.15, which represents the activity taking place during the add operation, and the "After" section, which represents the state of the array list after the operation is completed.)

Notice that inserting one element into the array list requires many steps; depending on where the element is inserted, it could require shifting the entire contents of the underlying array (if inserted at index 0), or it could require no shifting whatsoever (if inserted at location size).

The processing becomes even more complicated if we try to add an element to an array list that is already at its capacity. In this case, the underlying implementation creates a new array to hold the array list information—an array that is larger than the current array list. It then copies the contents of the old array into the new array, leaving an empty slot for the additional element. Finally, it copies the new element into the appropriate location of the new array and updates the `capacity` and `size` variables. The new array is now the array list; the old array is garbage and is eventually reclaimed by the run-time garbage-collection process.

There is actually more that goes on behind the scenes than we have described here; however, we think we have covered enough for you to get the point. With an array, memory is reserved ahead of time to hold the array elements; with an array list, memory can be allocated "on the fly," as needed. Array lists can be a useful construct for saving space, but the space savings might be at the expense of extra processing time. Time/space tradeoffs are common in computer programming.

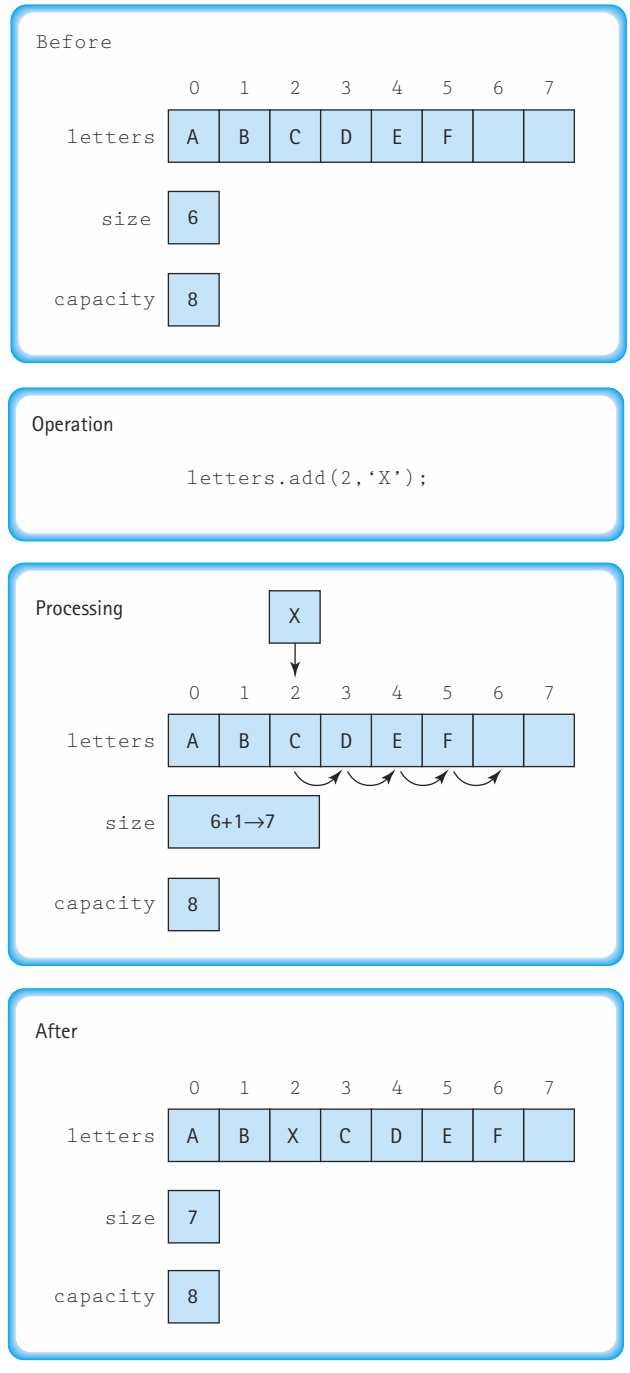


Figure 2.15 *Array list implementation*

Application Level Due to the similarities between arrays and array lists, we can use an array list in place of an array in virtually any application. However, the differences between arrays and array lists often mean that for a specific application, one or the other of these structures is the appropriate choice. It is impossible to list definite rules on when to choose one approach over the other, since there can be multiple factors to consider, and since each application has its own requirements. Nevertheless, we offer the following short set of guidelines:

Use an array when

1. space is not an issue.
2. execution time is an issue.
3. the amount of space required does not change much from one execution of the program to the next.
4. the actively used size of the array does not change much during execution.
5. the position of an element in the array has relevance to the application. (For example, the value in location *n* represents the profits for day *n* of a business period.)

Use an array list when

1. space is an issue.
2. execution time is not an issue.
3. the amount of space required changes drastically from one execution of the program to the next.
4. the actively used size of the array list changes a great deal during execution.
5. the position of an element in the array list has no relevance to the application.
6. most of the insertions and deletions to the array list take place at the size index. (Therefore, no extra overhead is incurred by these operations.)

We use an array list in Chapter 4 to implement a Stack ADT.

Building Our Own ADTs

In Chapter 1 we emphasized that the central task in the object-oriented design of software is the identification of classes. Once we identify the logical properties of the classes that we use to solve our problem, we must either find pre-existing versions of the classes or build them ourselves. Designing and building classes as ADTs allows us to take advantage of the benefits of abstraction and information hiding.

Remember that ADTs can be considered at three levels: The logical level specifies the interface and functionality, the implementation level is where the coding details take place, and the application level is where the ADT is used. Sometimes one programmer is involved in all three levels of an ADT—the same individual describes it, builds it, and uses it. At other times the design might come from one programmer, the implementation from another, and a third might be the one to use it. In the course of our discus-

sions, we typically assume that the designer and implementer are the same person; we call this person the programmer. We also assume that the same person, or perhaps other people, are the ones to use the ADT at the application level; in that role we call them the application programmers. Finally, there are the people who use the application programs; we call them the users.

To help us understand what makes a class an ADT, we return to two previous examples, `Circle` and `Date`. Figure 2.16 lists a version of each of these, side by side, so that you can easily compare their implementations. `Circle` is an example of a record structure. It is not an ADT since its instance variables are not hidden. `Date` is an ADT. Its instance variables are hidden and cannot be directly accessed from outside the class.

```
the Circle record
public class Circle
{
  public int xValue;
  public int yValue;
  public float radius;
  public boolean solid;
}
```

```
the Date ADT
public class Date
{
  protected int year;
  protected int month;
  protected int day;
  protected static final int MINYEAR = 1583;

  public Date(int newMonth, int newDay, int newYear)
  {
    month = newMonth;
    day = newDay;
    year = newYear;
  }

  public int yearIs()
  {
    return year;
  }

  public int monthIs()
  {
    return month;
  }

  public int dayIs()
  {
    return day;
  }
}
```

Figure 2.16 *`Circle` and `Date` implementations*

Access Modifiers

The difference in visibility of the `Circle` data and the `Date` data is due to the access modifiers used in the declaration of the data. Java allows a wide spectrum of access control, as summarized in the following table:

Modifier	Visibility
`public`	Within the class, subclasses in the same package, subclasses in other packages, everywhere
`protected`	Within the class, subclasses in the same package, subclasses in other packages
`package`	Within the class, subclasses in the same package
`private`	Within the class

The `public` access modifier used in `Circle` makes its data "publicly" available; any code that can "see" an object of the class can access and change its data. Additionally, any class derived from the `Circle` class inherits its public parts.

Public access sits at one end of the access spectrum, allowing open access to the data. At the other end of the spectrum is private access. When a programmer declares a class's variables and methods as `private`, they can be used only inside the class itself and they are not inherited by subclasses. We often use private access within our ADTs to hide their data. However, if we intend to extend our ADTs with subclasses, we may want to use the protected or package access instead.

The `protected` access modifier used in `Date` is similar to private access, only slightly less rigid. It "protects" its data from outside access, but allows it to be accessed from within its own class or from any class derived from its class. You may recall that in Chapter 1 we created a subclass of `Date` called `IncDate` that included a transformer method `increment`. The `increment` method required access to the instance variables of `Date`, since it would update the represented date to the next day. Therefore, the `Date` instance variables were assigned protected access. (An even better approach might have been to include `Date` and `IncDate` in the same package, perhaps a `Calendar` package, and use package access as described in the next paragraph.)

The remaining type of access is called *package access*. A variable or method of a class defaults to package access if none of the other three modifiers are used. Package access means that the variable or method is accessible to any other class in the same package; also the variable or method is inherited by any of its subclasses that are in the same package.

Note that the same rules for visibility and inheritance described above for instance variables apply equally well to the methods, constants, and inner classes of a class.

Exported Methods

If we hide the data of our ADTs, then how can other classes use the data? The answer is through publicly available methods of the class. By restricting access of the data of a

class to the methods of the class, we reap the benefits of abstraction and information hiding that were described in Chapter 1.

Consider once again the implementation of the Date ADT in Figure 2.16. The `year`, `month`, and `day` variables are all protected from outside access. This particular ADT provides one constructor method, `Date`, which accepts three integer parameters and initializes the variables of the `Date` object accordingly. The Date ADT also provides three observer methods: `yearIs`, `monthIs`, and `dayIs`. Using the constructor and observer methods, another class can create `Date` objects and "observe" the constituent data.

It is not hard to imagine creating some more interesting methods for the `Date` class. For example, as was suggested before, we could include a transformer method called `increment` that would change the value of the `Date` to the next day. We could also create a method that operates on more than one `Date` object—for example, a `difference` method that returns the number of days between two dates. The method could accept one date as a parameter and use the `Date` instance through which it is invoked as the other date. Its declaration might look something like this:

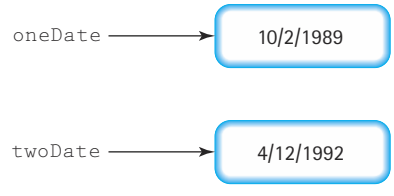
```
public int difference(Date inDate);
```

In that case, the following program segment would assign the value 5 to the variable `daysLeft`.

```
Date holiday = new Date (12, 25, 2002);
Date today = new Date (12, 20, 2002);
int daysLeft;

daysLeft = holiday.difference(today);
```

Copying Objects

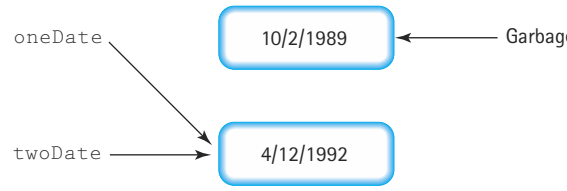
In the course of using an ADT, an application programmer might need to make a copy of the ADT object. Since ADTs are implemented as classes, they are handled by reference; if you simply use Java's assignment operator (=) to perform the copy, you end up with an alias of the copied object. For example, suppose `oneDate` and `twoDate` are both `Date` objects, representing the dates 10/2/1989 and 4/12/1992, respectively:



Then the statement

```
oneDate = twoDate;
```

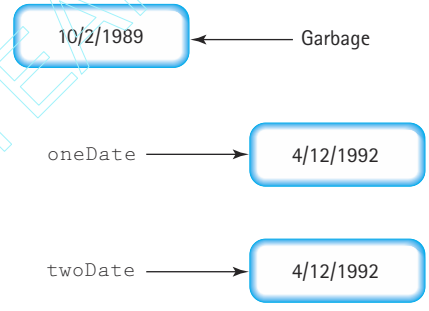
would create aliases, and garbage, as follows:



To create a true copy, and not just an alias, a programmer could use the `Date` constructor and observer methods as follows:

```
oneDate = new Date (twoDate.monthIs(), twoDate.dayIs(), twoDate.yearIs());
```

This approach would create a new `Date` object with the same variable values as the `twoDate` object. The result of the operation would look like this:



This approach eliminates the creation of an alias. Now, `oneDate` and `twoDate` are separate objects, and changes to one do not affect the other.

Since creating a copy of an ADT is a common operation, it is appropriate to include a special constructor for an ADT, called a *copy constructor*, which encapsulates the above operation. We pass the copy constructor an instance of the ADT and it creates a new instance of the ADT that is a copy of the argument. For the `Date` class the copy constructor would be:

```
public Date (Date inDate)
{
  year = inDate.year;
  month = inDate.month;
  day = inDate.day;
}
```

Notice that within the copy constructor the system has direct access to the instance variables of the `Date` parameter `inDate`, even though those variables were declared as protected. This works because this code resides inside the `Date` class and therefore has

access to the private and protected members. Using the copy constructor, we can now create a true copy as follows:

```
oneDate = new Date (twoDate);
```

Creating the copy constructor for the `Date` class was fairly straightforward. We simply had to copy the variables of the `Date` parameter to the fields of the new `Date` object. This approach works fine for a simple ADT like `Date`. However, we must be more careful when working with composite ADTs.

Previously in this chapter, in the section about Aggregate Objects, we listed the following definitions of the `Point` and `NewCircle` classes:

```
public class Point
{
  public int xValue;
  public int yValue;
}
```

```
public class NewCircle
{
  public Point location;
  public float radius;
  public boolean solid;
}
```

As you can see, an object of the class `NewCircle` is a composite object, since one of its instance variables is an object of the class `Point`. Consider the following code that implements a copy constructor for `NewCircle` in the same straightforward manner that was used for the `Date` class above:

```
public NewCircle (NewCircle inNewCircle)
// This code is incorrect
{
  location = inNewCircle.location;
  radius = inNewCircle.radius;
  solid = inNewCircle.solid;
}
```

At first glance this seems as if it would provide a reasonable copy of a `Circle` object. However, upon closer scrutiny, we see that there is a hidden alias that has been created. The line in the constructor that copies the location variable

```
location = inNewCircle.location;
```

is using the standard assignment statement on an object. Since all objects are handled by reference, what is actually copied is the reference to that object, rather than the contents of the object. We end up with two separate `Circle` objects that are both referencing the same `Point` object. The `NewCircle` copy constructor above is an example of a shallow copy. Shallow copying is rarely useful.

Shallow copy An operation that copies a source class instance to a destination class instance, simply copying all references so that the destination instance contains duplicate references to values that are also referred to by the source.

To rectify the problems created with a shallow copy, we need to create new instances of any nonprimitive variables of the object that we are copying. This approach results in a deep copy. The correct code for the copy constructor for `NewCircle` is:

```
public NewCircle (NewCircle inNewCircle)
{
  location = new Point;
  location.xValue = inNewCircle.location.xValue;
  location.yValue = inNewCircle.location.yValue;
  radius = inNewCircle.radius;
  solid = inNewCircle.solid;
}
```

Deep copy An operation that copies one class instance to another, using observer methods as necessary to eliminate nested references and copy only the primitive types that they refer to. The result is that the two instances do not contain any duplicate references.

The key statement in the code above is the first statement where we use the new command to create a new instance of a `Point` object.

Notice that in this example, since the classes we are using have public instance variables, we were able to just directly access the `x` and `y` values of the `location` variables of the `inNewCircle` parameter. If we were dealing with ADTs we would have to use the appropriate observer methods. Alternately, if the `Point` class included its own copy constructor, we could use it to create the new `Point` object:

```
location = new Point(inNewCircle.location);
```

Figure 2.17 summarizes our discussion of copying objects. It shows the results of all three approaches to copying a `Circle` object: using a simple assignment statement, using a shallow copy, and using a deep copy. In the figure, both `oneCircle` and `twoCircle` are objects of type `NewCircle`.

Exceptions

When creating our own ADTs it is possible to identify exceptional situations that require special processing. If it is the case that the special processing cannot be determined ahead of time. It is application dependent; we should use the Java *exception* mechanism to throw the problem out of the ADT and force application programmers to handle the exceptional situation on their own. On the other hand, if handling the exceptional situation can be hidden within the ADT, then there is no need to burden the application programmers with the task of handling exceptions.

For an example of an exception created to support a programmer-defined ADT, let's return to our `Date` class. As currently defined, a `Date` constructor could be used to create dates with nonexistent months—for example, 15/15/2000 or even −5/15/2000. We could avoid the creation of such dates by checking the legality of the month argument

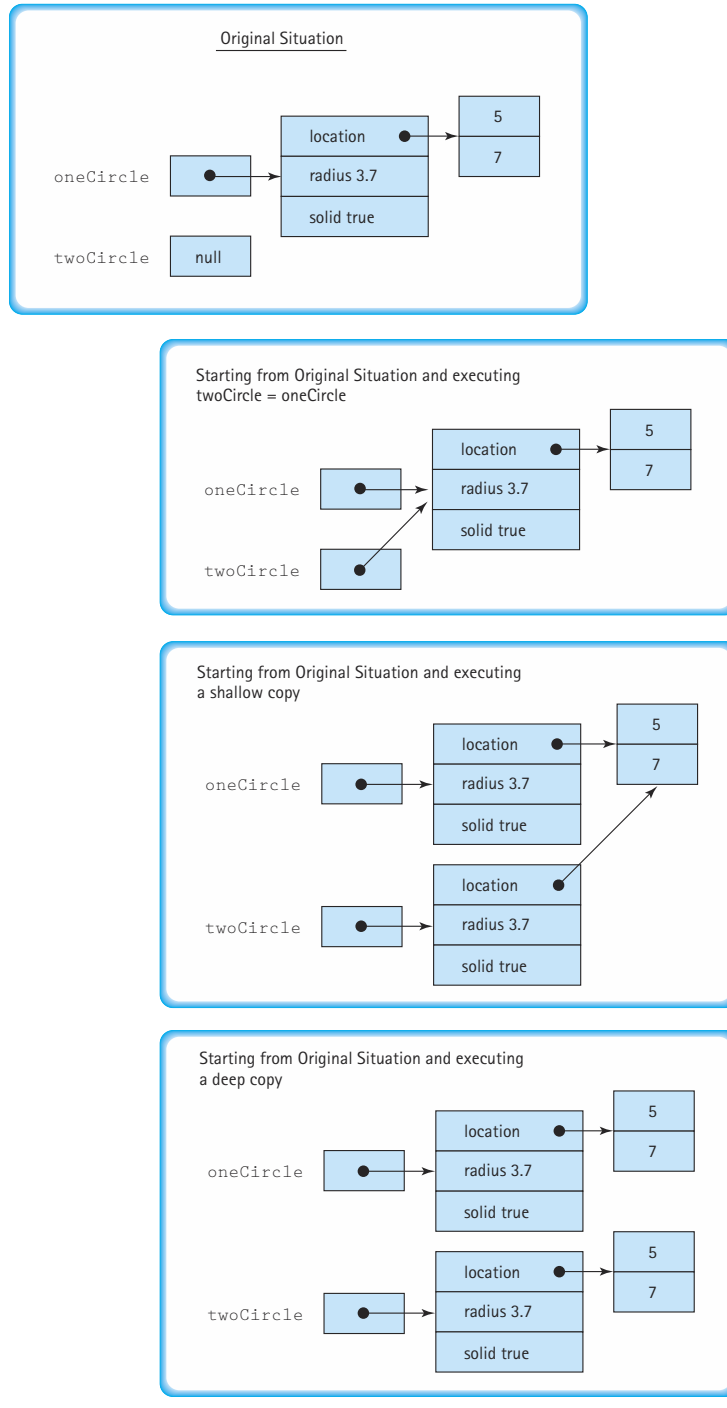


Figure 2.17 *Copying objects*

passed to the constructor. But what should our constructor do if it discovers an illegal argument? Some options:

- Write a warning message to the output stream. That's not a very good option because within the Date ADT we don't really know what output stream is being used by the application.
- Instantiate the new `Date` object to some default date, perhaps 0/0/0. The problem with this approach is that the application program may just continue processing as if nothing is wrong, and produce erroneous results. In general it is better for a program to "bomb" then to produce erroneous results that may be used to make bad decisions.
- Throw an exception. This way, normal processing is interrupted and the constructor does not have to return a new object; instead, the application program is forced to acknowledge the problem and either handle it or throw it out to the next level.

Once we have decided to handle the situation with an exception, we must decide whether to use one of the library's predefined exceptions, or to create one of our own. A study of the library in this case reveals a candidate exception called `DataFormatException`, to be used to signal data format errors. We could use that exception but we decide it doesn't really fit, since its not the format of the data that is the problem in this case, it is the values of the data.

So, we decided to create our own exception, `DateOutOfBounds`. We could call it "MonthoutofBounds" but we decide that we want to use the exception to indicate other potential problems with dates, and not just problems with the month value. For example, in the `Date` class we defined a class variable `MINYEAR` (set to 1583), representing the first complete year in which the Gregorian calendar was in use. Application programmers should not use our `Date` class to represent dates earlier than that year. The idea is that date calculations get very complicated if you allow dates before 1583. For one thing, leap year rules were different; for another, there were 10 days that were skipped in the middle of 1582. We are imagining that we have added methods to the class that would be affected by such things, for example a method that returns the number of days between two dates. Therefore, we wish to disallow such dates.

We create our `DateOutOfBounds` exception by extending the library `Exception` class. It is customary when creating your own exceptions to define two constructors, mirroring the two constructors of the `Exception` class. In fact, the easiest thing to do is define the constructors so that they just call the corresponding constructors of the superclass:

```
public class DateOutOfBoundsException extends Exception
{
  public DateOutOfBoundsException()
  {
    super();
  }
```

```
  public DateOutOfBoundsException(String message)
  {
    super(message);
  }
}
```

The first constructor is used to create an exception without an associated message; the second constructor creates an exception with a message equal to the string argument passed to the constructor.

Next we need to consider when, within our Date ADT, we throw the exception. All places within our ADT where a date value is created or changed should be examined to see if the resultant value could be an illegal date. If so, we should create an object of our exception class with an appropriate message, and throw the exception. Here is how we might write a `Date` constructor to check for legal months and years. (Checking for legal days is much more complicated and we leave that as an exercise.)

```
public Date(int newMonth, int newDay, int newYear) throws DateOutOfBound-
sException
{
  if ((newMonth <= 0) || (newMonth > 12))
    throw new DateOutOfBoundsException("month must be in range 1 to 12");
  else
    month = newMonth;

  day = newDay;

  if (newYear < MINYEAR)
    throw new DateOutOfBoundsException("year " + newYear +
                                       " is too early");
  else
    year = newYear;
 }
```

Notice that the message defined for each *throws* clause pertains to the problem discovered at that point in the code. This should help the application program that is handling the exception, or at least provide pertinent information to the user of the program if the exception is propagated all the way out to the user level.

Finally, let's see how an application program might now use the `Date` class. Consider a program called `UseDates` that prompts the user for a month, day, and year, and create a `Date` object based on the user's responses. In the following code we ignore the

details of how the prompt and response are handled, to concentrate on the topics of our current discussion:

```
public class UseDates
{
  public static void main(String[] args) throws DateOutOfBoundsException
  {
    Date theDate;
    // Program prompts user for a date
    // M is set equal to user's month
    // D is set equal to user's day
    // Y is set equal to user's year
    theDate = new Date(M, D, Y);

    // Program continues
  }
}
```

When this program runs, and the user responds with a legal month, day, and year, there is no problem. However, if the user responds with an illegal value—for example, a year value of 1051—the `DateOutOfBoundsException` is thrown by the `Date` constructor; since it is not caught within the program, it is thrown out to the interpreter. The interpreter stops execution of the program after displaying a message like this:

```
Exception in thread "main" DateOutOfBoundsException: year 1051 is too early
          at Date.<init>(Date.java:18)
          at UseDates.main(UseDates.java:57)
```

The interpreter's message includes the name and message string of the exception, and a trace of what calls were made leading up to the exception being thrown.

Alternately, the `UseDates` class could be defined to catch and handle the exception itself, rather than throwing it to the interpreter. The application programmer could reprompt for the date in the case of the exception being raised. Then `UseDates` might be written as follows (again we ignore the user interface details):

```
public class UseDates
{
  public static void main(String[] args)
  {
    Date theDate;
    boolean DateOK = false;

    while (!DateOK)
```

```
    {
      // Program prompts user for a date
      // M is set equal to user's month
      // D is set equal to user's day
      // Y is set equal to user's year
      try
      {
        theDate = new Date(M, D, Y);
        DateOK = true;
      }
      catch(DateOutOfBoundsException OB)
      {
        output.println(OB.getMessage());
      }
    }

    // Program continues
  }
}
```

If the *new* statement executes without any trouble, meaning the `Date` constructor did not throw an exception, then the `DateOK` variable is set to `true` and the *while* loop terminates. On the other hand, if the `DateOutOfBounds` exception is thrown by the `Date` constructor, it is caught by the *catch* statement. This in turn prints out the message associated with the exception and the *while* loop is re-executed, again prompting the user for a date. The program repeatedly prompts for date information until it is given a legal date.

Notice that the `main` method no longer throws `DateOutOfBoundsException`, since it handles the exception itself.

There are several factors to consider when determining how to use exceptions when creating our own ADTs. First of all, we should decide what types of events can trigger exceptions. Remember that exceptions can be used to signal any out-of-the-ordinary event that requires special processing—there is no language-based rule that says the event must be error related. For example, it would be possible to break out of an input loop in reaction to an exception you raise when you try to read past the end of a file. Reading the end-of-file marker is not really an error; it is something we expect to happen eventually when we read files. It is, in a sense, an exceptional condition, and we can use Java's exception mechanisms to help us handle its occurrence.

To simplify our ADT definitions, and to support a common approach to the way we define our ADTs, we throw programmer-defined exceptions from our ADTs only in situations involving errors. For example, unexpected date values being passed to a method or illegal sequencing of methods calls are errors. However, this does not mean we always use exceptions in these cases.

When dealing with error situations within our ADT methods, we have several options:

1. We can detect and handle the error within the method itself. This is the best approach if the error can be handled internally and if it does not greatly complicate design.
2. We can throw an exception related to the error and force the calling method to either handle the exception or to rethrow it. If it is not clear how to handle a particular error situation, the best approach might be to throw it out to a level where it can be handled.
3. We can ignore the error situation. Recall the "programming by contract" discussion, related to preconditions, in the Designing for Correctness section of Chapter 1. If the preconditions of a method are not met, the method is not responsible for the consequences. This approach is best if we are confident that the contract is usually met by the application classes.

Therefore, when we define our ADTs, we partition potential error situations into three sets: those to be handled internally to the ADT, those to be thrown as an exception back to the calling process, and those that are assumed not to occur. We document this third approach in the preconditions of the appropriate methods. We attempt to strike a balance between the complexity required to handle all possible error situations internally, and the lack of safety involved with handling everything by contract.

As a general rule, an exceptional situation should be handled at the lowest level that "knows" how to handle it. If the information needed to handle the exception is not available at a level, then the exception should be thrown. As we create ADTs to be used in applications we see that quite often it is the application level that can best handle the exceptions raised within the ADTs. We see examples of this as we proceed through the text.

The feature section below suggests a sequence of steps to follow when designing and creating ADTs. The steps include many of the techniques introduced in this subsection.

Designing ADTs

When you design and create your own ADTs you can follow these steps:

1. Determine the general purpose of the ADT; determine how the application programmers use the ADT to help solve their problems in a general sense.
2. List the specific types of operations the application program performs with the ADT. If possible, note how often the different operations are used, that is, the expected relative frequency of operation calls.
3. Identify a set of public methods to be provided by the ADT class that allow the application program to perform the desired operations. Note that there might not be a one-to-one correspondence between the desired operations and the exported methods. It may be that a single operation requires several method invocations. For example, in Chapter 3 we define a list ADT with methods `lengthIs`, `reset`, and `getNextItem`. An application program

must use all three of these methods to implement a "Print List" operation. In addition, a specific method might be needed for more than one operation. For example, the `lengthIs` list method might be used by a "Print List" operation and by a "Report List Size" operation.

4. Identify other public methods, based on experience and general guidelines, which help make the ADT generally usable. For example, the copy constructor described in the earlier section titled Copying Objects is usually a good method to include. You might organize all your identified methods into constructors, observers, transformers, and iterators.
5. Identify potential error situations and classify into
   a. Those that are handled by throwing an exception
   b. Those that are handled by contract
   c. Those that are ignored
6. Define the needed exception classes.
7. Decide how to structure the data to best support the needed operations and identified methods. Remember that alternate organizations may support some operations better than others. This is where the frequency of operation information may be useful.
8. Decide on a protection level for the identified data. Hide the data as much as possible.
9. Identify private structures and methods that support the required public methods. Functional decomposition of the required actions of the public methods may help identify common requirements that can be supported by shared private methods.
10. Implement the ADT, possibly collecting all related files into a single package.
11. Create a test driver like the one at the end of Chapter 1 and test your ADT with a wide variety of operations.

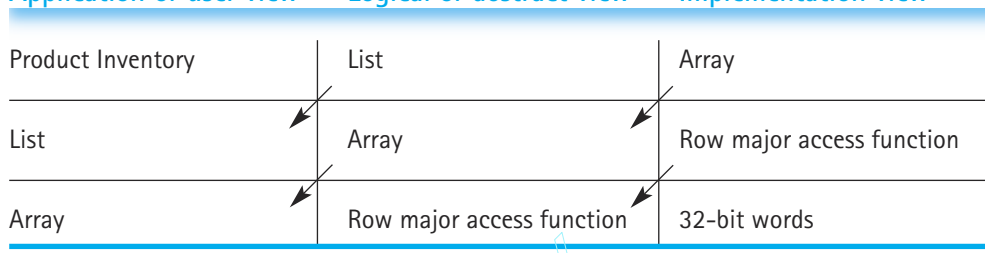
Note that the classic data structures, modeled as ADTs created in the remainder of this text have evolved over the last 50 years. Therefore, we can draw from a great deal of previous research and experience when designing these structures, instead of analyzing specific problem situations as suggested above.

Summary

We have discussed how data can be viewed from multiple perspectives, and we have seen how Java encapsulates the implementations of its predefined types and allows us to encapsulate our own class implementations.

As we create data structures, using built-in data types such as arrays and classes to implement them, we see that there are actually many levels of data abstraction. The abstract view of an array might be seen as the implementation level of the programmer-defined data structure `List`, which uses an array to hold its elements. At the logical level, we do not access the elements of `List` through their array indexes but through a set of accessing operations defined especially for objects of `List` type. A data type that is designed to hold other objects is called a *container* or collection type. Moving up a level, we might see the abstract view of `List` as the implementation level of another programmer-defined data type, `ProductInventory`, and so on.

Perspectives on Data



What do we gain by separating the views of the data? First, we reduce complexity at the higher levels of the design, making the program easier to understand. Second, we make the program more easily modifiable: The implementation can be completely changed without affecting the program that uses the data structure. We use this advantage in this text, developing various implementations of the same objects in different chapters. Third, we develop software that is *reusable*: The structure and its accessing operations can be used by other programs, for completely different applications, as long as the correct interfaces are maintained. You saw in the first chapter of this book that the design, implementation, and verification of high-quality computer software is a very laborious process. Being able to reuse pieces that are already designed, coded, and tested cuts down on the amount of work we have to do.

In the chapters that follow we extend these ideas to build other container classes: lists, stacks, queues, priority queues, trees, and graphs. While the Java Class Library provides many of these data structures (along with generic algorithms and iterator structures), the techniques for building these structures is so important in computer science that we believe you should learn them now.

We consider these data structures from the logical view. What is our abstract picture of the data, and what accessing operations can we use to create, assign to, and manipulate the data elements? We express our logical view as an abstract data type (ADT) and record its description in a data specification.

Next, we take the application view of the data, using an instance of the ADT in a short example.

Finally, we change hats and turn to the implementation view of the ADT. We consider the Java type declarations that represent the data structure, as well as the design of the methods that implement the specifications of the abstract view. Data structures can be implemented in more than one way, so we often look at alternative representations and methods for comparing them. In some of the chapters, we include a longer Case Study in which instances of the ADT are used to solve a problem.

Summary of Classes and Support Files

Here are the classes defined in Chapter 2. The classes are listed in the order in which they appear in the text. The summary includes the name of the class file, the page on which the file is first referenced, and a few notes. The notes explain how the class was used in the text, followed by additional notes if appropriate. Note that we do not include classes defined within other classes (inner classes), such as the `Circle` class that was defined within the `TestCircle` class, in the table. The class files are available on our web site in the `ch02` subdirectory.

Classes Defined in Chapter 2

File	First Ref.	Notes
`TestCircle.java`	page 83	Illustrates records and record component selection.
`FigureGeometry.java`	page 88	An example of an interface.
`Point.java`	page 93	Very small class; it is used to build an example of an aggregate object.
`NewCircle.java`	page 93	Example of a class that defines aggregate objects. `NewCircle` includes an instance variable of the class `Point`.

Other than the `Exception` class, which was discussed in Section 2.3, no Java Library Classes were used in any examples for the first time in this text within this chapter. Of course, many library classes were discussed; but they were not used in programs.

Exercises

2.1 Different Views of Data

1. Why are primitive types sometimes called atomic types?
2. Explain what we mean by data abstraction.
3. What is data encapsulation? Explain the programming goal "to protect our data abstraction through encapsulation."
4. Describe the four categories of operations that can be performed on encapsulated data. Give an example of each operation using a Library analogy.
5. Name three different perspectives from which we can view data. Using the logical data structure "a list of student academic records," give examples of what each perspective might tell us about the data.
6. Consider the abstract data type `GroceryStore`.
   a. At the application level, describe `GroceryStore`.
   b. At the logical level, what grocery store operations might be defined for the customer?

c. Specify (at the logical level) the operation `CheckOut`.

d. Write an algorithm (at the implementation level) for the operation `CheckOut`.

e. Explain how parts (c) and (d) represent information hiding.

2.2 Java's Built-in Types

7. What primitive types are predefined in the Java language?

8. What composite types are predefined in the Java language?

9. Describe the component selector for classes, when they are used as records.

10. Define a `toString` method for the circle class listed on the following pages:

    a. page 83

    b. page 93

11. What is an alias? Show an example of how it is created by a Java program. Explain the dangers of aliases.

12. Assume that `date1` and `date2` are objects of type `IncDate` as defined in Chapter 1. What would be the output of the following code?

```
date1 = new IncDate(5, 5, 2000);
date2 = date1;
System.out.println(date1);
System.out.println(date2);
date1.increment();
System.out.println(date1);
System.out.println(date2);
```

13. Assume that `date1` and `date2` are objects of type `IncDate` as defined in Chapter 1. What would be the output of the following code?

```
date1 = new IncDate(5, 5, 2000);
date2 = new IncDate(5, 5, 2000);
if (date1 == date2)
  System.out.println("equal");
else
  System.out.println("not equal");
date1 = date2;
if (date1 == date2)
  System.out.println("equal");
else
  System.out.println("not equal");
date1.increment();
if (date1 == date2)
  System.out.println("equal");
else
  System.out.println("not equal");
```

14. What is garbage? Show an example of how it is created by a Java program.
15. What is an abstract method?
16. What sorts of constructs can be declared in a Java interface?
17. Briefly describe four uses for Java interfaces.
18. What are the fundamental differences between classes and arrays?
19. Describe the component selectors for one-dimensional arrays.
20. Write a program that declares a ten-element array of `int`, uses a *for* loop to initialize each element to the value of its index squared, and then uses another *for* loop to print the contents of the array, one integer per line.
21. Define a three-dimensional array at the logical level.
22. Suggest some applications for three-dimensional arrays.
23. Indicate which Java types would most appropriately model each of the following (more than one may be appropriate for each):
    a. A chessboard
    b. Information about a single product in an inventory-control program
    c. A list of famous quotations
    d. The casualty figures (number of deaths per year) for highway accidents in Texas from 1954 to 1974
    e. The casualty figures for highway accidents in each of the states from 1954 to 1974
    f. The casualty figures for highway accidents in each of the states from 1954 to 1974, subdivided by month
    g. An electronic address book (name, address, and phone information for all your friends)
    h. A collection of hourly temperatures for a 24-hour period

2.3 Class-Based Types

24. What Java construct is used to represent abstract data types?
25. Explain the difference between using a Java class to create a record and to create an ADT
26. Explain how packages are used to organize Java files.
27. List and briefly describe the contents of five Java library packages.
28. List the eight Java Library "wrapper" classes that support the objectification of Java's primitive types.
29. List and describe five Java Library classes that are not described in this chapter.
30. Research the Java Library `Random` class. Use it in a program to do the following.
    a. Generate a sequence of 10,000 random integers between 1 and 100 and output the average value generated

b. Play the high/low guessing game with the user; the program generates a random integer between 1 and 100,000. The user must repeatedly guess the number until it is correct. After each guess, the program informs the user if the secret number is higher or lower than the guess.

Be sure to carefully test your program(s).

31. Write a program that declares a ten-element array of `Integer`, uses a *for* loop to initialize each element to the value of its index squared, and then uses another *for* loop to print the contents of the array, one integer per line.

32. Describe the output of the following code that uses `String` variables `S1`, `S2`, and `S3`.

```
S1 = "Alex";
S2 = "Bob";
S3 = S1 + S2;
System.out.println(S3);
S2 = S1.toUpperCase();
System.out.println(S2);
S3 = "Chris".
if (S1.compareTo(S3) < 0)
  System.out.println("less than zero");
else
  System.out.println("not less than zero");
```

33. Explain the differences between arrays and array lists.

34. For each of the following situations, state whether it is best to use an array list or an array.

a. To hold student test grades, where the size of the class of students is always between 15 and 20

b. To hold student test grades, where the size of classes varies widely

c. To hold the number of miles traveled each day of a month

d. To hold a list of items, where you need to repeatedly insert elements into random locations in the list

e. To hold a list of items, where you insert and remove items only from the far end of the list.

35. Describe each of the four levels of visibility provided by Java's visibility modifiers.

36. Illustrate with a figure the difference between a shallow copy and a deep copy of an aggregate object.

37. Consider an ADT `SquareMatrix`. (A square matrix can be represented by a two-dimensional array with n rows and n columns.)

a. Write the specification for the ADT, assuming a maximum size of 50 rows and columns. Include the following operations:

`MakeEmpty(n)`, which sets the first `n` rows and columns to zero

`StoreValue(i, j, value)`, which stores value into the position at row `i`, column `j`

`Add`, which adds two matrices together

`Subtract`, which subtracts one matrix from another

`Copy`, which copies one matrix into another

b. Convert your specification to a Java class declaration.

c. Implement the member methods.

d. Write a test plan for your class.

38. Expand your solution to Exercise 34 of Chapter 1, where you implemented the `Date` and `IncDate` classes, to include the appropriate throwing of the `DateOutOfBoundsException`, as described in this chapter.

39. Write a class `Array` that encapsulates an array and provides bounds checked access. The private instance variables should be `int index` and `int array[10]`. The public members should be a default constructor and methods (signatures shown below) to provide read and write access to the array:

```
void insert(int location, int value);

int retrieve(int location);
```

If the `location` is within the correct range for the array, the `insert` method should set that location of the array to the value. Likewise, if the `location` is within the correct range for the array, the `retrieve` method should return the value of that location of the array. In either case, if the `location` is not within the correct range, the method should throw an exception of type `ArrayoutofBoundsException`. Write a driver to check the array accesses. Your driver should assign values to the array by using the `insert` method, using the `retrieve` method to read these values back from the array. It should also try calling both methods with illegal location values. Catch any exceptions thrown by placing the calls in a *try* block with an appropriate *catch* block following.

40. Describe the steps to follow when designing your own ADTs and implementing them with the Java class mechanism.

Chapter 3

ADTs Unsorted List and Sorted List

Goals

Measurable goals for this chapter include that you should be able to

- describe the List ADT at a logical level
- classify list operations into the categories constructor, iterator, observer, and transformer
- identify the pre- and postconditions of a given list operation
- use the list operations to implement utility routines such as the following application-level tasks:
  - Print the list of elements
  - Create a list of elements from a file of element information
  - Store a list of elements on a file
- implement the following list operations for both unsorted lists and sorted lists:
  - Create a list
  - Determine whether the list is full
  - Determine the size of the list
  - Insert an element
  - Retrieve an element
  - Delete an element
  - Reset the list and repeatedly return the next item from the list
- explain the use of Big-O notation to describe the amount of work done by an algorithm
- compare the unsorted list operations and the sorted list operations in terms of Big-O approximations
- describe uses of Java's *abstract class* and *interface* constructs with respect to defining ADTs
- design and create classes for use with a generic list
- use a List ADT as a component of a solution to an application problem

This chapter centers on the List ADT: its definition, its implementation, and its use in problem solving. In addition to learning about this important data structure, this material should help you understand the relationships among the logical, application, and implementation levels of an ADT. In the course of the exploration of these topics, several Java constructs for supporting abstraction are introduced. Seeing how these constructs are used should enhance your appreciation for the power of abstraction. We also introduce in this chapter an analysis tool, Big-O notation, which allows us to compare the efficiency of different ADT implementations.

3.1 Lists

We all know intuitively what a list is; in our everyday lives we use lists all the time—grocery lists, lists of things to do, lists of addresses, lists of party guests.

In computer programs, lists are very useful abstract data types. They are members of a general category of abstract data types called containers; containers hold other objects. There are languages in which the list is a built-in structure. In Lisp, for example, the list is the main data type provided in the language. Although list classes are provided in the Java Class Library, the techniques for building lists and other abstract data types are so important that we show you how to design and write your own.

Linear relationship Each element except the first has a unique predecessor, and each element except the last has a unique successor

Length The number of items in a list; the length can vary over time

Unsorted list A list in which data items are placed in no particular order; the only relationship between data elements is the list predecessor and successor relationships

Sorted list A list that is sorted by the value in the key; there is a semantic relationship among the keys of the items in the list

From a programming point of view, a list is a homogeneous collection of elements, with a linear relationship between its elements. A linear relationship means that, at the logical level, each element on the list except the first one has a unique predecessor and each element except the last one has a unique successor. (At the implementation level, there is also a relationship between the elements, but the physical relationship may not be the same as the logical one.) The number of items on the list, which we call the length of the list, is a property of a list. That is, every list has a length.

Lists can be unsorted—their elements may be placed into the list in no particular order—or they can be sorted. For instance, a list of numbers can be sorted by value, a list of strings can be sorted alphabetically, and a list of grades can be sorted numerically.

When the elements in a sorted list are of composite types, we can define their logical order in many different ways. For example, suppose we have a list of student information, with each student represented by their first name, last name, identification number, and three test scores. Some of the ways we can sort such a list are:

- by last name, alphabetically
- by last name, alphabetically, and then by first name, alphabetically (in other words, the first name is used to determine relative ordering if two or more last names are identical)
- by identification number
- by average test score

If the sort order is determined directly by using the student information, such as in the first three approaches, we say that that information represents the key for the list element. In the first approach, the <last name> is the key; in the second approach, the combination of <last name – first name> is the key; and in the third approach, the <identification number> is the key. If a list cannot contain items with duplicate keys, it is said to have a unique key. In this example, the best candidate for use as a unique key is the identification number, since it is likely to have a unique value for each student in a school.

Key The attributes that are used to determine the logical order of the items on a list

This chapter deals with many kinds of lists. We make the assumption that our lists are composed of unique elements. We point out the ramifications of dropping this assumption on our list abstractions and implementations at various places within the chapter. When sorted, our lists are sorted from smallest to largest key value, though it is certainly possible to sort them largest to smallest should your application need this.

There are two basic approaches to implementing container structures such as lists: the "by copy" approach and the "by reference" approach. For our lists in this chapter, we use the "by copy" approach. This means that when a client program inserts an item into our lists, it is actually a copy of the item that is placed on the list. In addition, when an item is retrieved from our list by a client program, it is a copy of the item on the list that is returned to the program. We use the alternate approach, storing and returning references to the items instead of copies of the items, for other container structures starting in Chapter 4. At that point we discuss more thoroughly the important differences between the two approaches.

Progressing through the chapter, we develop unsorted and sorted lists of strings, sorted lists of generic elements, and in the case study, a sorted list of house information for a real estate application. As we progress, we introduce both the Java *abstract class* mechanism and the Java *interface* mechanism to help refine our list ADTs and make them more generally usable. Each time we implement a new form of list, we include the corresponding UML diagram. Each figure that displays a UML diagram includes all of the previous diagrams, so that you easily can compare the implementation approaches.

3.2 Abstract Data Type Unsorted List

Logical Level

There are many different operations that programmers can provide for lists. For different applications we can imagine all kinds of things users might need to do to a list of elements. In this chapter we formally define a list and develop a set of general-purpose operations for creating and manipulating lists. By doing this, we are building an abstract data type.

To create the definition of a list as an abstract data type, we must identify a set of operations that allow us to access and manipulate the list. In this section we design the specifications for a List ADT where the items on the list are unsorted; that is, there is no

semantic relationship between an item and its predecessor or successor. They simply appear next to one another on the list.

Abstract Data Type Operations

Designing an ADT to be used by many applications is not the same as designing an application program to solve a specific problem. In the latter case we can use CRC cards to enact scenarios of the application's use, allowing us to identify and fix holes in our design before turning to implementation. Identifying scenarios for use of a general ADT is not as straightforward. We must stand back and consider what operations every user of the data type would want it to provide.

Recall that there are four categories of operations: constructors, transformers, observers, and iterators. We begin by reviewing each category and considering which List ADT operations fit into the respective categories.

Constructors A constructor creates a new instance of the data type. In Java, it is a public method with the same name as the ADT's class name. There is one piece of information that our ADT needs from the client to construct an instance of the list data type: the maximum number of items to be on the list. As this information varies from application to application, it is logical for the client to have to provide it. We can also define a default list size to be used in case the client does not provide the information.

At the end of the previous chapter we suggested that it is a good idea to include a copy constructor when defining an ADT. A *copy constructor* accepts an instance of the ADT as a parameter and creates a copy of it. Copy constructors are most appropriate when the ADT implements an unstructured composite type, such as the `Date` and `Circle` examples of the previous chapters. Although there can be situations in which a copy constructor can be helpful for an application programmer who is using a structured composite type such as a list, these situations are rare. We do not define a copy constructor for our List ADTs.

Transformers Transformers are operations that change the content of the structure in some way. A common transformer is one that makes the structure empty. However, in Java, the constructor methods associate a new, empty structure with the current instance of the ADT, effectively making it empty. Therefore, we do not need another method for making the list empty. We do need transformers to put an item into the structure, or to remove a specific item from the structure. For our Unsorted List ADT, let's call these transformers `insert` and `delete`.

Note that, since we implement our operations as object methods, the list is the object through which the method is invoked, and therefore the list itself is available to the method for manipulation. The `insert` and `delete` methods need an additional parameter: the item to be inserted or deleted. For this Unsorted List ADT, let's assume

that the item to be inserted is not currently on the list and the item to be deleted is on the list.

A transformer that takes two sorted lists and merges them into one sorted list or appends one list to another is a *binary transformer*. The specification for such an operation is given in the exercises, where you are asked to implement it.

Observers Observers also come in several forms. They ask true/false questions[1] about the ADT (Is the structure empty?). They select or access a particular item (Give me a copy of the last item.). Or they return a property of the structure (How many items are in the structure?). The Unsorted List ADT needs at least two observers: `isFull` and `lengthIs`. The `isFull` observer method returns `true` if the list is full, `false` otherwise; `lengthIs` tells us how many items are on the list, as opposed to the maximum capacity of the list.

If an abstract data type places limits on the component type, we could define other observers. For example, if we know that our abstract data type is a list of numerical values, we could define statistical observers such as `minimum`, `maximum`, and `average`. Here, at the logical level, we are interested in generality; we know nothing about the type of the items on the list, so we use only general observers.

If we make the client responsible for checking for error conditions, we must make sure that the ADT gives the user the tools with which to check for the conditions. The operations that allow the client to determine whether an error condition occurs are observers. Since we are assuming that our list does not include duplicate elements, we should provide an observer that searches the list for an item with a particular key and returns whether or not the item has been found. Let's call this one `isThere`. The application programmer can use the `isThere` observer to prevent insertion of a duplicate item into the list. For example:

```
if (!list.isThere(item)) list.insert(item);
```

Iterators Iterators are used with composite types to allow the user to process an entire structure, component by component. To give the user access to each item in sequence, we provide two operations: one to initialize the iteration process (analogous to Reset or Open with a file) and one to return a copy of the "next component" each time it is called. The user can then set up a loop that processes each component. Let's call these operations `reset` and `getNextItem`. Note that `reset` is not an iterator, but is an auxiliary operation that supports the iteration. Another type of iterator is one that takes an operation and applies it to every element on the list.

Element Types Before we can formalize the specification for the Unsorted List ADT, we must consider the type of items to be held on the list. Later in the chapter we

[1]A method that returns a boolean value defined on a set of objects is sometimes called a *predicate*, with the term *observer* used for methods that inquire about an instance variable of an object.

learn how to define a generic list—a list that can hold elements of many different types. For now, so that we can concentrate on the definition and implementation of the list operations, we limit ourselves to working with a list of strings. Therefore, we call our ADT `UnsortedStringList`. In order to keep our analysis as generally applicable as possible, we still refer to list components as "elements" or "items," rather than as "strings," and we call our ADT the Unsorted List ADT in much of our discussion.

Unsorted List ADT Specification

Structure:

The list elements are Strings. The list contains unique elements; i.e., no duplicate elements as defined by the key of the list. The list has a special property called the current position—the position of the next element to be accessed by `getNextItem` during an iteration through the list. Only `reset` and `getNextItem` affect the current position.

Definitions (provided by user):

maxItems: An integer specifying the maximum number of items to be on this list.

Operations (provided by Unsorted List ADT):

void UnsortedStringList (int maxItems)

Effect: Instantiates this list with capacity of `maxItems` and initializes this list to empty state.

Precondition: `maxItems` > 0

Postcondition: This list is empty.

void UnsortedStringList ()

Effect: Instantiates this list with capacity of 100 and initializes this list to empty state.

Postcondition: This list is empty.

boolean isFull ()

Effect: Determines whether this list is full.

Postcondition: Return value = (this list is full)

int lengthIs ()

Effect: Determines the number of elements on this list.

Postcondition: Return value = number of elements on this list

boolean isThere (String item)

Effect:	Determines whether `item` is on this list.
Postcondition:	Return value = (`item` is on this list)

void insert (String item)

Effect:	Adds copy of `item` to this list.
Preconditions:	This list is not full. `item` is not on this list.
Postcondition:	`item` is on this list.

void delete (String item)

Effect:	Deletes the element of this list whose key matches `item` 's key.
Precondition:	One and only one element on this list has a key matching `item` 's key.
Postcondition:	No element on this list has a key matching the argument `item` 's key.

void reset ()

Effect:	Initializes current position for an iteration through this list.
Postcondition:	Current position is first element on this list.

String getNextItem ()

Effect:	Returns a copy of the element at the current position on this list and advances the value of the current position.
Preconditions:	Current position is defined. There exists a list element at current position. No list transformers have been called since most recent call to `reset`.
Postconditions:	Return value = (a copy of element at current position) If current position is the last element then current position is set to the beginning of this list; otherwise, it is updated to the next position.

In this specification, the responsibility of checking for error conditions is put on the user through the use of preconditions that prohibit the operation's call if these conditions exist. Recall that we call this approach programming "by contract." We have given the user the tools, such as the `isThere` operation, with which to check for the

conditions. Another alternative would be to define an error variable, have each operation record whether an error occurs, and provide operations that test this variable. A third alternative would be to let the operations detect error conditions and throw appropriate exceptions. We use programming by contract in this chapter so that we can concentrate on the list abstraction and the Java constructs that support it, without having to address the extra complexity of formally protecting the operations from misuse. We use other error-handling techniques in later chapters.

The specification of the list is somewhat arbitrary. For instance, the overall assumption about the uniqueness of list items could be dropped. This is a design choice. If we were designing a specification for a specific application, then the design choice would be based on the requirements of the problem. We made an arbitrary decision not to allow duplicates. Allowing duplicates in this ADT implies changes in several operations. For example, instead of deleting an element based on its value, we might require a method that deletes an element based on its position on the list. This, in turn, might require a method that returns the position of an item on the list based on its key value.

Additionally, assumptions about specific operations could be changed—for example, we specified in the preconditions of `delete` that the element to be deleted must exist on the list. It would be just as legitimate to specify a delete operation that does not require the element to be on the list and leaves the list unchanged if the item is not there. Perhaps that version of the `delete` operation would return a `boolean` value, indicating whether or not an element had been deleted. We could even design a list ADT that provided both kinds of delete operations. In the exercises you are asked to explore and make some of these changes to the List ADTs.

Application Level

The set of operations that we are providing for the Unsorted List ADT may seem rather small and primitive. However, this set of operations gives you the tools to create other special-purpose routines that require knowledge of what the items on the list represent. For instance, we have not included a print operation. Why? We don't include it because in order to write a good print routine, we must know what the data members represent. The application programmer (who does know what the data members look like) can use the `lengthIs`, `reset`, and `getNextItem` operations to iterate through the list, printing each data member in a form that makes sense within the application. In the code that follows, we assume the desired form is a simple numbered list of the string values. We have emphasized the lines that use the list operations.

```
void printList(PrintWriter outFile, UnsortedStringList list)
// Effect: Prints contents of list to outFile
// Pre:    List has been instantiated
//         outFile is open for writing
// Post:   Each component in list has been written to outFile
//         outFile is still open
{
  int length;
  String item;
```

```
  list.reset();
  length = list.lengthIs();
  for (int counter = 1; counter <= length; counter++)
  {
    item = list.getNextItem();
    outFile.println(counter + ". " + item);
  }
}
```

For example, if the list contains the strings "Anna Jane," "Joseph," and "Elizabeth," then the output would be:

1. Anna Jane
2. Joseph
3. Elizabeth

Note that we defined a local variable `length`, stored the result of `list.lengthIs()` in it, and used the local variable in the loop. We could have just used the method call directly in the loop:

```
for (int counter = 1; counter <= list.lengthIs(); counter++)
```

We used the other approach for efficiency reasons. That way the `lengthIs` method is called only once, saving the overhead of extra method calls.

In the `printList` method, we made calls to the list operations specified for the Unsorted List ADT, printing a list without knowing how the list is implemented. At an application level, the operations we used (`reset`, `lengthIs`, and `getNextItem`) are logical operations on a list. At a lower level, these operations are implemented by Java methods, which manipulate an array or other data-storing medium that holds the list's elements. There are many functionally correct ways to implement an abstract data type. Between the user picture and the eventual representation in the computer's memory, there are intermediate levels of abstraction and design decisions. For instance, how is the logical order of the list elements reflected in their physical ordering? We address questions like this as we now turn to the implementation level of our ADT.

Implementation Level

There are two standard ways to implement a list. We look at a *sequential array-based list implementation* in this chapter. The distinguishing feature of this implementation is that the elements are stored sequentially, in adjacent slots in an array. The order of the elements is implicit in their placement in the array.

The second approach, which we introduce in Chapter 5, is a *linked-list implementation*. In a linked implementation, the data elements are not constrained to be stored in physically contiguous, sequential order; rather, the individual elements are stored "somewhere in memory," and their order is maintained by explicit links between them.

Before we go on, let's establish a design terminology for our list algorithms that's independent of the implementation and type of items stored on the list. Doing this allows us to describe algorithms that are valid no matter which of the standard approaches we use.

List Design Terminology Assuming that location "accesses" a particular list element,

location.node( )	Refers to all the data at location, including implementation-specific data.
location.info( )	Refers to the application data at location.
last.info( )	Refers to the application data at the last location on the list.
location.next( )	Gives the location of the node following location.node( ). If location is the end of the list, it gives the first location of the list.

A few clarifications are needed. What is meant by "all the data" at a location, and "the application data" at a location? Remember that although we are currently dealing with lists of strings, we eventually expand the kinds of elements we can use to any kind of data. The "application data" refers to the data from the application associated with a list element. In addition to the application data, a list element might have certain information associated with it, related to the implementation of the list; for example, a variable holding the location of the next list element. By "all the data" we mean the application data plus the implementation data, if there is any.

What then is location? For an array-based implementation, location is an index, because we access array slots through their indexes. For example, the design statement

```
Print location.info( )
```

means "Print the application data in the array slot at index location;" eventually it might be coded in Java within the array-based implementation as

```
outFile.println(list.info[location]);
```

When we look at the linked implementation in Chapter 5, the code implementing the design statement is quite different, but the design statement itself remains the same. Thus, using this design notation, we define implementation-independent algorithms for our Unsorted List ADT. Hopefully, we can design our list algorithms just once using the design notation and then implement them using either of the implementation approaches.

What does location.next( ) mean in an array-based sequential implementation? To answer this question, consider how we access the next list element stored in an array: We increment the location index. The design statement

```
Set location to location.next()
```

might be coded in Java within the array-based implementation as

```
if (location == numItems - 1)       // Location is an array index
  location = 0;
else
  location++;
```

We have not introduced this list design terminology just to force you to learn the syntax of another computer language. We simply want to encourage you to think of the list, and the parts of the list elements, as *abstractions*. At the design stage, the implementation details can be hidden. There is a lower level of detail that is encapsulated in the "methods" node, info, and next. Using this design terminology, we hope to record algorithms that can be coded for both array-based and linked implementations.

Instance Variables

In our implementation, the elements of a list are stored in an array of `String` objects.

```
String[] list;
```

There are two size-related attributes of the list: capacity and current length. The *capacity* of the list is the maximum number of elements that can be stored on the list. We do not need an instance variable to hold the capacity of the list since we can use the array attribute `length` to determine the capacity of the list at any point within our implementation. In other words, the capacity of our list is the length of the underlying array: `list.length`.

However, we do need an instance variable to keep track of the current number of items we have stored in the array (also known as the current length of the list). We name this variable `numItems`. This variable can also be used to record where the last item was stored. Because the list items are unsorted, when we put the first item into the list, we place it into the first slot; the second item goes in the second slot, and so forth. Because our language is Java, we must remember that the first slot is indexed by 0, the second slot by 1, and the last slot by `list.length - 1`. Now we know where the list begins—in the first array slot. Where does the list end? The array ends at the slot with index `list.length - 1`, but the list ends in the slot with index `numItems - 1`. For example, if the list currently holds 5 items, they are kept in array locations 0 through 4; the value of the `numItems` instance variable is 5; and the next array location to insert a new item is also 5.

Is there any other information about the list that we must include? Both operations `reset` and `getNextItem` refer to a "current position." What is this current position? It is the index of the last element accessed in an iteration through the list. We need an instance variable to keep track of the current position. Let's call it `currentPos`. The method `reset` initializes `currentPos` to 0. The method `getNextItem` returns the value in `list[currentPos]` and increments `currentPos`. The ADT specification states that only `reset` and `getNextItem` affect the current position. Figure 3.1 illustrates the instance variables of our class `UnsortedStringList`. Here is the beginning of the class

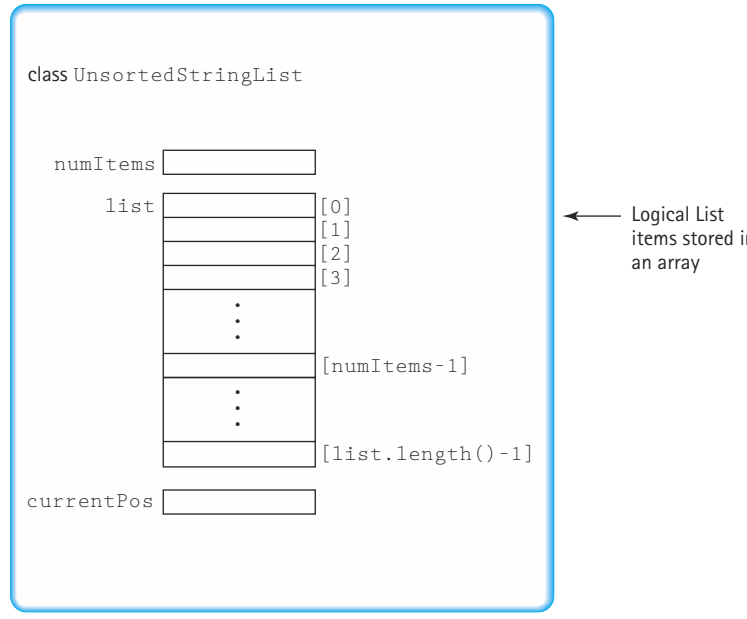


Figure 3.1 *Instance variables of Unsorted List ADT*

file, which includes the variable declarations. Note that it also includes an introductory comment and a *package* statement. The `UnsortedStringList` class is the first of several string list classes we develop. We collect all these classes together into a single package called `ch03.stringLists` (the class files can be found in the subdirectory `stringLists` or in the subdirectory `ch03` of the directory `bookFiles` on our web site).

```
//--------------------------------------------------------------------------
// UnsortedStringList.java          by Dale/Joyce/Weems                Chapter 3
//
// Defines all constructs for an array-based list of strings that is not
// kept sorted
//--------------------------------------------------------------------------

package ch03.stringLists;

public class UnsortedStringList
{
  protected String[] list;     // Array to hold list elements
  protected int numItems;      // Number of elements on this list
  protected int currentPos;    // Current position for iteration
⋮
```

Notice that we use the `protected` visibility modifier for each of the variables. Recall that this means that the variables can be "seen" only by the methods of the `UnsortedStringList` class or its subclasses. We use this approach because we create a subclass later in this chapter that needs access to the variables. This type of visibility still protects the variables from direct access by the applications that use the class.

A design choice we wish to point out, but choose not to use, is to write an `ArrayList`-based class for use here. Since the `ArrayList` class provides a variable-sized array, we could allow the underlying implementation to shrink and grow to mirror the changes in the size of the list. We would not have to deal with a "max items" constraint, so we would not need to list preconditions such as "list is not full." You are asked to investigate this alternative in the exercises.

Constructors

Now let's look at the operations that we have specified for the Unsorted List ADT. The first two operations are constructors that create empty lists. Remember that a class constructor is a method having the same name as the class, but having no return type. A constructor's purpose is to instantiate an object of the class, to initialize variables and, if necessary, to allocate resources (usually memory) for the object being constructed. Like any other method, a constructor has access to all variables and methods of the class. A new list is created empty; that is, the number of items is 0.

Our first constructor requires a positive integer parameter, which indicates the size for the underlying array.

```
public UnsortedStringList(int maxItems)
// Instantiates and returns a reference to an empty list object with
// room for maxItems elements
{
  numItems = 0;
  list = new String[maxItems];
}
```

The code for this constructor is straightforward and requires no further explanation. We have decided not to include a restatement of the method preconditions and postconditions, established in the ADT specification, when listing our code. In some cases we provide multiple versions of the same method, and we believe repeated listing of these conditions is redundant and would make for tedious reading. Therefore, we list these conditions only when we define the logical-level view of our ADTs. Nevertheless, we encourage you to always include preconditions and postconditions in comments at the beginning of your methods. Code that is meant to be used needs such documentation, but in this text, where we're already explaining the code in great detail, the comments aren't as necessary.

Our second constructor does not have a parameter. In this case, the default size of the underlying array is 100.

```
public UnsortedStringList()
// Instantiates and returns a reference to an empty list object
// with room for 100 elements
{
  numItems = 0;
  list = new String[100];
}
```

Notice that these two methods have the same name: `UnsortedStringList`. How is this possible? Remember that in the case of methods, Java uses more than just the name to identify them; it also uses the parameter list. A method's name, the number and type of parameters that are passed to it, and the arrangement of the different parameter types within the list, combine into what Java calls the signature of the method.

Signature The distinguishing features of a method heading. The combination of a method name with the number and type(s) of its parameters in their given order

Overloading The repeated use of a method name with a different signature

Java allows us to use the name of a method as many times as we wish, as long as each one has a different signature. When we use a method name more than once, we are overloading its identifier. The Java compiler needs to be able to look at a method call and determine which version of the method to invoke. The two constructors in class `UnsortedStringList` both have different signatures: One takes no arguments, the other takes an `int`. Java decides which version to call according to the arguments in the statement that invokes `UnsortedStringList`.

Simple Observers

The first nonconstructor operation, `isFull`, just checks to see whether the current number of items on the list is equal to the length of the array.

```
public boolean isFull()
// Returns whether this list is full
{
  return (list.length == numItems);
}
```

The body of the observer object method `lengthIs` is also just one statement.

```
public int lengthIs()
// Returns the number of elements on this list
{
  return numItems;
}
```

So far, we have not used our special design terminology. The algorithms have all been straightforward. The next operation, `isThere`, is more complex.

isThere Operation

The `isThere` operation allows the application programmer to determine whether a list item with a specified key exists on the list. In the case of the string list, the key is simply the string value. This string value is input to the method in the parameter `item`. A `boolean` value is returned by the method—if the string `item` matches a string on the list, `true` is returned; otherwise, `false` is returned.

Because the list items are unsorted, we must use a linear search. We begin at the first component on the list and loop until either we find a component equal to the parameter or there are no more strings to examine. Recall from Chapter 2 that we have two ways to see if two strings are the same; we could use the `equals` method of the `String` class or the `compareTo` method of the `String` class. We choose to use the `compareTo` method, since we also use it in other parts of the list implementation. Recall that this method returns a 0 if the strings are equal. Therefore, we can code

```
if (item.compareTo(list[location]) == 0)
  found = true;
```

But how do we know when to stop searching if we do not find the string? If we have examined the last element of the list, we can stop. Thus, in our design terminology, we keep looking as long as we have not examined last.info( ).We summarize these observations in the algorithm below.

isThere (item): returns boolean

```
Initialize location to position of first list element
Set found to false
Set moreToSearch to (have not examined last.info())

while moreToSearch AND NOT found
  if item.compareTo(location.info()) == 0
    Set found to true
  else
    Set location to location.next()
    Set moreToSearch to (have not examined last.info())

return found
```

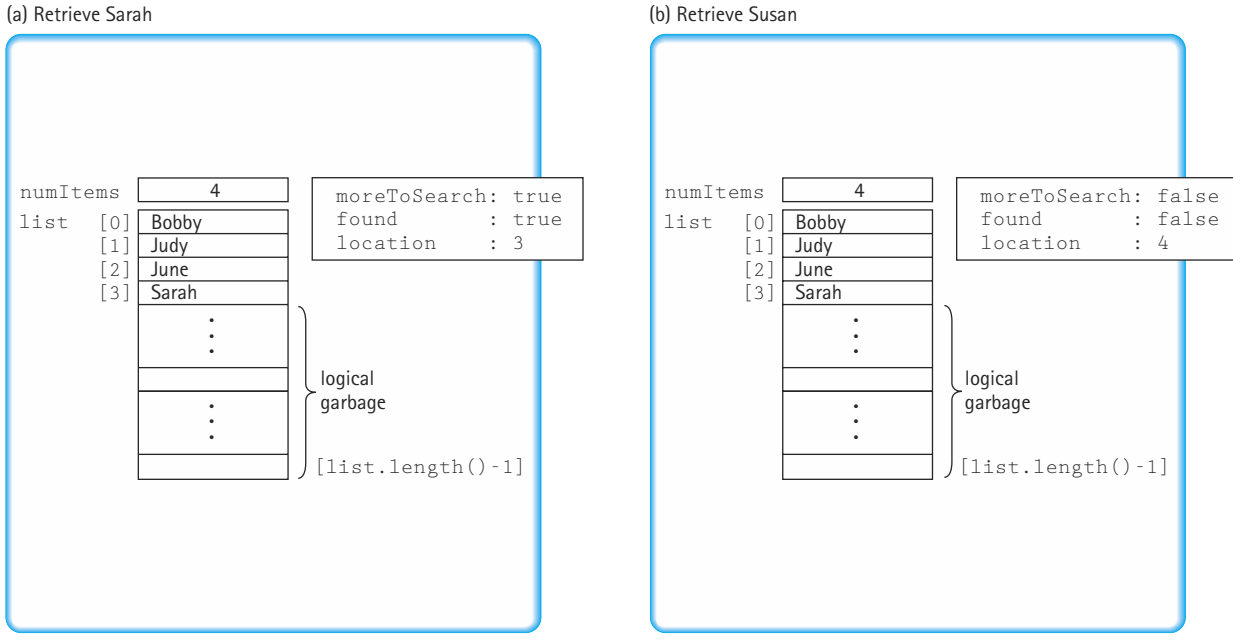


Figure 3.2 *Retrieving an item in an unsorted list.*

Before we code this algorithm, let's look at the cases where we find the item on the list and where we examine last.info( ) without finding it. We represent these cases in Figure 3.2 in an Honor Roll list. We first retrieve Sarah (see Figure 3.2(a)). Sarah is on the list, so when the search is completed, `moreToSearch` is `true`, `found` is `true`, and `location` is 3. The loop is exited because `found` became `true` when `item` was equal to the contents of location 3. Next, we retrieve Susan (see Figure 3.2(b)). Susan is not on the list, so when the search is completed `moreToSearch` is `false`, `found` is `false`, and `location` is equal to `numItems`. The loop is exited because `moreToSearch` became `false` after we examined the last information on the list.

Now we are ready to code the algorithm replacing the general design notation with the equivalent array notation. The substitutions are straightforward except for initializing `location` and determining whether we have examined last.info( ). To initialize `location` in an array-based implementation in Java, we set it to 0. We know we have not examined last.info( ) as long as `location` is less than `numItems`. Be careful: Because Java indexes arrays from 0, the last item on the list is at index `numItems - 1`. Here is the coded algorithm.

```
public boolean isThere (String item)
// Returns true if item is on this list, otherwise returns false
{
  boolean moreToSearch;
  int location = 0;
  boolean found = false;
```

```
    moreToSearch = (location < numItems);

    while (moreToSearch && !found)
    {
      if (item.compareTo(list[location]) == 0)  // If they match
        found = true;
      else
      {
        location++;
        moreToSearch = (location < numItems);
      }
    }

   return found;
  }
```

insert Operation

Because the list elements are not sorted by value, we can put the new item anywhere. A straightforward strategy is to place the item in the `numItems` position and increment `numItems`.

insert (item)

Set numItems.info() to copy of item
Increment numItems

This algorithm is translated easily into Java.

```
  public void insert (String item)
  // Adds a copy of item to this list
  {
    list[numItems] = new String(item);
    numItems++;
  }
```

delete Operation

The `delete` method takes an item whose value indicates which item to delete. There are clearly two parts to this operation: finding the item to delete and removing it. We can use the `isThere` algorithm to search the list. When `compareTo` returns a nonzero value, we increment `location`; when `compareTo` returns 0, we exit the loop and remove the element. Because the preconditions for `delete` state that an item with the same key is definitely on the list, we do not need to test for reaching the end of the list.

How do we remove the element from the list? Let's look at the example in Figure 3.3. Removing Sarah from the list is easy, for hers is the last element on the list (see Figures 3.3a and 3.3b). If Bobby is deleted from the original list, however, we need to move up all the elements that follow to fill in the space—or do we? See Figure 3.3(c). If the list is sorted by value, we would have to move all the elements up as shown in Figure 3.3(c), but because the list is unsorted, we can just swap the item in the `numItems - 1` position with the item being deleted (see Figure 3.3(d)). In an array-based implementation, we do not actually remove the element; instead, we cover it up with the elements that previously followed it (if the list is sorted) or the element in the last position (if the list is unsorted). Finally, we decrement `numItems`.

```
public void delete (String item)
// Deletes the element that matches item from this list
{
  int location = 0;

  while (item.compareTo(list[location]) != 0)
    location++;

  list[location] = list[numItems - 1];
  numItems--;
}
```

Iterator Operations

The `reset` method is analogous to the Open operation for a file in which the file pointer is positioned at the beginning of the file so that the first input operation accesses the first component of the file. Each successive call to an input operation gets the next item in the file. Therefore, `reset` must initialize `currentPos` to indicate the first item on the list.

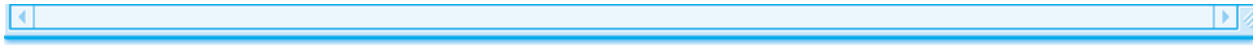
The `getNextItem` operation provides access to the next item on the list by returning currentPos.info( ) and incrementing `currentPos`. To do this, it must first "record" currentPos.info( ), then increment `currentPos`, and finally return the recorded information.

reset

Initialize currentPos to position of first list element

getNextItem: returns String

Set next to currentPos.info()
Set currentPos to currentPos.next()
return copy of next



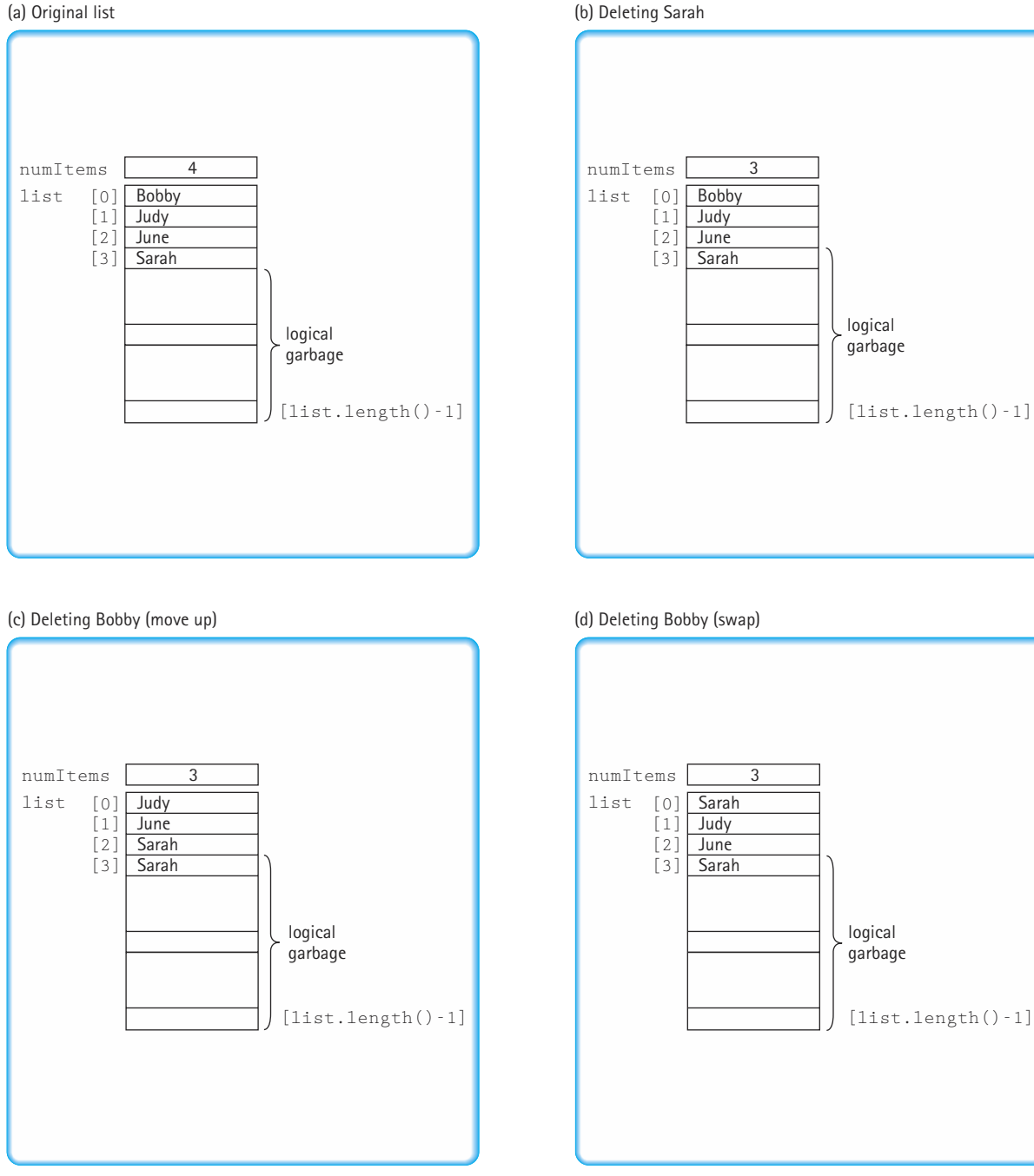


Figure 3.3 *Deleting an item in an unsorted list*

The currentPos value always indicates the next item to be processed in an iteration. To be safe, we decided to reset it automatically, in the getNextItem method, when the end of the list is reached. Therefore, there are two places where currentPos can be set to 0: in the reset method, and in the getNextItem method when the end of the list is reached. The code for the iteration operations is as follows:

```
public void reset()
// Initializes current position for an iteration through this list
{
  currentPos  = 0;
}

public String getNextItem ()
// Returns copy of the next element on this list
// And advances the current position
{
  String next = list[currentPos];
  if (currentPos == numItems-1)
    currentPos = 0;
  else
    currentPos++;
  return new String(next);
}
```

The getNextItem method could also be implemented using the modulus operation:

```
currentPos = (currentPos++) % (numItems - 1);
```

The getNextItem method returns a String variable. That means that it returns a reference to a string object. Notice that we have elected to create a new string object using the String class's copy constructor, and to return a reference to the new object, rather than a reference to the string object that is actually on the list. As we stated before, we implement our lists "by copy." Why did we do this? The answer is that we wish to maintain information hiding. If we return a reference into the list, we have given the application an alias of a hidden list element. So, rather than do that, we create a copy of the string, and return a reference to the copy. The list user is never allowed to directly see or manipulate the contents of the list. These details of the list implementation are encapsulated by the ADT.

In this case we are being overly protective; since strings are immutable objects there would be no potential harm in returning a reference to the actual string that is on the list. The application program cannot change the string, so in this case the work of copying the list object is unnecessary. Nevertheless, we wish to emphasize the need for care when returning values from within our ADTs. As mentioned previously, in Chapter 4 we follow the alternate approach, namely, returning references to the objects contained in our ADTs, and consider the strengths and drawbacks of each approach.

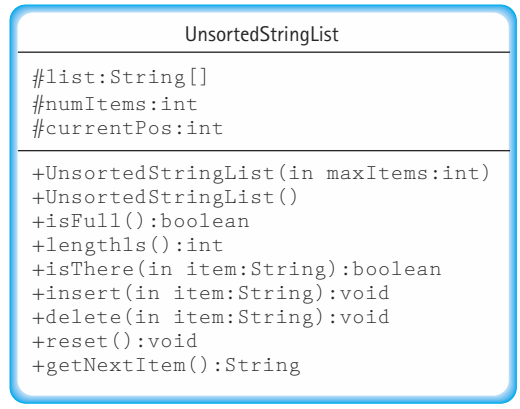


Figure 3.4 *UML diagram of* `UnsortedStringList`

Consider how the application programmer might use the list iteration methods. The programmer can use the length of the list to control a loop asking to see each item in turn. What happens if the program inserts or deletes an item in the middle of an iteration? Nothing good, you can be sure! Adding and deleting items changes the length of the list, making the termination condition of the iteration-counting loop invalid. Depending on whether an addition or deletion occurs before or after the iteration point, our iteration loop could end up skipping or repeating items.

We have several choices of how to handle this possibly dangerous situation. The list can throw an exception, the list can reset the current position when inserting or deleting, or the list can disallow transformer operations while an iteration is taking place. We choose the latter here by way of a precondition in the documentation.

The UML class diagram in Figure 3.4 represents our `UnsortedStringList` implementation.

Test Plan

To test our Unsorted List ADT, we create a test driver program similar to the one we created at the end of Chapter 1 to test the IncDate ADT. That test driver accepted a sequence of instructions from an input file that indicated which method of `IncDate` to invoke next. The test input also included any parameter values required by the `IncDate` methods. Results of the method invocations were printed to an output file. Meanwhile, a final count of the number of test cases was indicated in an output window.

As we planned when we created that test driver, it is not difficult to transform it into a test driver for a different ADT. To use it to test our Unsorted List ADT, we simply change the value assigned to the `testname` variable near the start of the program, change the declaration of the variables to appropriate ones for testing our list ADT, and rewrite the sequence of *if-else* statements to invoke and report on the list methods instead of the date methods.

We do not go into all the details of the code for the test driver. Note that since there are two constructors for the Unsorted List we must assign them two separate "code names" for our test input file. We simply chose to use "UnsortedStringList1" and "UnsortedStringList2". Here is the beginning of the main processing loop within the test driver:

```
// Process commands
while(!command.equals("quit"))
  {
    if (command.equals("UnsortedStringList1"))
  {
    size = Integer.parseInt(dataFile.readLine());
    list = new UnsortedStringList(size);
    outFile.println("The list is instantiated with size " + size);
  }
  else
  if (command.equals("UnsortedStringList2"))
  {
    list = new UnsortedStringList();;
    outFile.println("The list is instantiated with default size");
  }
  else
  if (command.equals("isFull"))
  {
    outFile.println("The list is full is " + list.isFull());
  }
   .
   .
   .
```

You can study the entire `TDUnsortedStringList.java` program (it's on our web site). What is important for us now is planning how to use the test driver to test our ADT.

The constructors `UnsortedStringList (int maxItems)` and `UnsortedStringList ()` can be exercised throughout our tests every time we create an `UnsortedStringList` object.

`lengthIs`, `insert`, and `delete` can be tested together. That is, we insert several items and check the length; we delete several items and check the length. How do we know that `insert` and `delete` work correctly? We can make calls to the `reset` and `getNextItem` methods to examine the structure of the list; a good approach would be to use `reset` and `getNextItem` to create a "print list" test method (such as defined in the application-level subsection), that could be called many times during the testing process. A `PrintList` method is included in the `TDUnsortedStringList.java` program.

To test the `isFull` operation, we can instantiate a list of size 5, insert four items and print the result of the test, and then insert the fifth item and print the result of the test. To test `isThere`, we must search for items that we know are on the list and for items that we know are not on the list.

How do we organize our test plan? We should classify our test possibilities. For example, an item can be in the first position on the list, in the last position on the list, or somewhere else on the list. So we must be sure that our `delete` can correctly delete items in these positions. We must also check that `isThere` can find items in these same positions. We should also check the `lengthIs` method at the boundary cases of an

empty list and a full list. Notice that this test plan is mostly a black-box strategy. We are looking at the list as described in the interface, not in the code.

These observations are summarized in the following test plan, which concentrates on the observer methods and the `insert` method. To be complete the plan must be expanded to use both constructors, to test the `delete` method, to test various combinations of `insert` and `delete`, and, if program robustness is desired, to test how the software responds to situations precluded by the method preconditions—for example, insertion into a full list. The tests are shown in the order in which they should be implemented.

Operation to be Tested and Description of Action	Input Values	Expected Output
`UnsortedStringList`		
print `lengthIs`		0
print `isFull`		false
print `isThere("Tom")`		false
Print List		empty list
`UnsortedStringList`	5	
`insert`	Tom	
print `lengthIs`		1
print `isFull`		false
print `isThere`	Tom	true
Print List		Tom
`insert`	Julie	
`insert`	Nora	
`insert`	Maeve	
print `lengthIs`		4
print `isFull`		false
print `isThere`	Tom	true
print `isThere`	Julie	true
print `isThere`	Maeve	true
print `isThere`	Kevin	false
Print List		Tom, Julie, Nora, Maeve
`insert`	Kevin	
print `lengthIs`		5
print `isFull`		true
print `isThere`	Tom	true
print `isThere`	Julie	true
print `isThere`	Kevin	true
Print List		Tom, Julie, Nora, Maeve, Kevin
etc.		

The file `testlist1.dat`, that accompanies the program file on our web site, provides an example of a set of test data. It is not a complete test. The file `testout1.dat` shows the results of the following program invocation:

```
java TDUnsortedStringList testlist1.dat testout1.dat
```

The key to properly testing any software is in the plan: It must be carefully thought out and it must be written. We have discussed the basic approach needed for testing the Unsorted List ADT, listed a partial test plan, and provided a test driver (in the file `TDUnsortedStringList.java`). We leave the creation of the complete written test plan as an exercise.

3.3 Abstract Classes

We have just completed the design and implementation of an Unsorted List ADT. In the next section we follow the same basic approach to create a Sorted List ADT. However, before we do that we take a look at Java's abstract class mechanism. We can use an abstract class to take advantage of the similarities between the Unsorted and Sorted List ADTS.

Relationship between Unsorted and Sorted Lists

Suppose you are given the task of creating a Sorted List ADT. The first step you might take is to identify the logical operations that you need to include. As you start to identify the operations, you might have the feeling that you have done this exercise before. Let's see, you'll need constructors to create your list. You'll need some way to put things onto the list, so you need an `insert` method. Of course, you might want to remove things from the list, so you need a `delete` method.

Sound familiar? As you think about it, you realize that all of the logical operations we defined for the Unsorted List ADT are also needed for your Sorted List ADT. The logical definition of those operations did not rely on whether or not the list was sorted. If you look at the Unsorted List ADT specification, you can see that the entire specification may be reused. The only changes that need to be made are to the preconditions and postconditions of the transformer methods `insert` and `delete`: They must specify that the list is sorted. `insert` and `delete` are the only two methods that affect the underlying ordering of the list items. The condition

"The list is sorted."

can be added to both their preconditions and postconditions, and you are all set.

Since you were able to reuse most of the specification of the Unsorted List ADT to specify your Sorted List ADT, maybe you can also reuse some of the implementation. In fact, assuming you again wish to use an array-based implementation, you can reuse the entire class except the implementations of the `insert` and `delete` methods. We look at

the implementations of these methods in the next subsection. We also look at a variant of the `isThere` method. Although you can just reuse the `isThere` method of the Unsorted List ADT, we are able to create a more efficient version of the method under the assumption that the list is sorted.

Reuse Options

There are several ways we could reuse the code of the Unsorted List ADT to create the code for the Sorted List ADT. Let's look at three approaches: Cut and Paste, Direct Inheritance, and Abstract Classes.

Cut and Paste

We could create a Sorted List class completely independent of the Unsorted List ADT class. Just create a new file called `SortedStringList.java`, "cut and paste" the code that we are able to reuse into the new file, rename the constructors to match the file name, and create the new code for the three methods that we want to change. Once we are finished there is no longer any formal link between the two classes: Cut and paste, direct inheritance, and abstract classes.

However, this lack of connection between the two classes can be detrimental. Consider, for example, if someone using the Sorted List class discovers an error in the `getNextItem` method. Suppose they fix the method but do not realize that a "copy" of the method exists in another class? This means that although a bug has been detected, and a solution devised and implemented, the same bug is still plaguing another class. If we could somehow formally link the two classes together, so that the code for the common methods only appears in one place, then both classes would share any updates made to these methods.

Direct Inheritance

Since the Sorted List ADT class can use several of the methods of the Unsorted List class, maybe we should make the former a subclass of the latter:

```
public class SortedStringList extends UnsortedStringList
...
```

Within the Sorted List class we can redefine the three methods that need to be changed. With this approach we do create a formal link between the two classes, and changes to the shared methods would affect both classes.

While this approach is probably better than the previous approach, it still has some problems. The main problem is that the inheritance relationship just doesn't make sense. Recall that in Chapter 1 we stated that the inheritance relationship usually represents an "is a" relationship. In the example in Chapter 1, an `IncDate` *is a* `Date`; an `IncDate` object was a special kind of `Date` object. Here, that relationship doesn't make sense. Saying that a sorted list *is a* unsorted list sounds like nonsense.

Just because the *is a* relationship does not make sense doesn't mean that you can't use inheritance. It does, however, often mean that using inheritance might lead to problems later. For example, due to Java's rules for assignment of object variables, it would

be possible for an application program to include the following code, assuming that the Sorted List ADT inherited from the Unsorted List ADT:

```
UnsortedStringList unsorted;
SortedStringList sorted = new SortedStringList(10);
unsorted = sorted;
```

This creates the rather confusing situation in which an Unsorted List variable is referencing a Sorted List object. This is completely legal in the world of Java—and it usually makes sense if the *is a* relationship makes sense. But as you can see, in this case it seems illogical. So, although using inheritance solves the problems identified in the previous subsection, another approach might be more appropriate.

Abstract Classes

Java offers another construct, called an *abstract class*, which resolves the deficiencies of both of the previous approaches.

An *abstract method* is one that is declared without a method body. For the sake of this discussion, let's call a normal method that is declared with a body a *concrete method*.

We discussed abstract methods in Chapter 2 when we looked at the Java `interface` construct. You may recall that a Java interface was not allowed to contain any concrete methods; it could only contain abstract methods. An abstract class, on the other hand, can contain both concrete methods and abstract methods. It must contain at least one abstract method. To indicate that a class is abstract, we use the Java keyword `abstract` in its definition. You'll see an example of this syntax in the next subsection. An abstract class cannot be instantiated. It must be extended by another class, which provides the missing implementations of the abstract methods.

We previously pointed out that it does not make sense to say that a sorted list *is a* unsorted list. Similarly, it doesn't make sense to reverse that; it does not make sense to say an unsorted list *is a* sorted list. What then is the relationship between a sorted list and an unsorted list? Easy, they are both lists! We can use an abstract class to model this relationship.

We first create an abstract list class; its concrete methods provide the operations that our two list ADTs share in common and its abstract methods provide the operations that are not shared. We can then create two concrete classes that extend the abstract list class, one that implements an unsorted list and the other that implements a sorted list. With this approach we maintain the common code for the shared methods and we create a reasonable *is a* inheritance structure: an unsorted list is a list and a sorted list is a list.

An Abstract List Class

Our abstract list class is very straightforward. It is based on the `UnsortedStringList` class developed in the previous section. We simply change the name of the class, and the

constructor, to `StringList`, add the keyword `abstract` to the header line, and remove the method bodies from the `insert`, `delete`, and `isThere` methods. We now declare these three methods as abstract, and end their declaration lines with a semicolon. You should also notice that we have only retained one of the constructors, the one which accepts an integer parameter `maxItems`. The other constructor is redundant in this scheme, as you see when we extend this class with the concrete classes in the next section. Finally, notice that we place `StringList` in the same package as we placed `UnsortedStringList`.

```
//--------------------------------------------------------------------------
// StringList.java             by Dale/Joyce/Weems                  Chapter 3
//
// Defines all constructs for an array-based list that do not depend
// on whether or not the list is sorted
//--------------------------------------------------------------------------

package ch03.stringLists;

public abstract class StringList
{
  protected String[] list;            // Array to hold this list's elements
  protected int numItems;             // Number of elements on this list
  protected int currentPos;           // Current position for iteration

  public StringList(int maxItems)
  // Instantiates and returns a reference to an empty list object
  // with room for maxItems elements
  {
    numItems = 0;
    list = new String[maxItems];
  }

  public boolean isFull()
  // Returns whether this list is full
  {
    return (list.length == numItems);
  }

  public int lengthIs()
  // Returns the number of elements on this list
  {
    return numItems;
  }
```

```
  public abstract boolean isThere (String item);
  // Returns true if item is on this list; otherwise, returns false

  public abstract void insert (String item);
  // Adds a copy of item to this list

  public abstract void delete (String item);
  // Deletes the element that matches item from this list.

  public void reset()
  // Initializes current position for an iteration through this list
  {
    currentPos  = 0;
  }

  public String getNextItem ()
  // Returns copy of the next element on this list
  {
    String next = list[currentPos];
    if (currentPos == numItems-1)
      currentPos = 0;
    else
      currentPos++;
    return new String(next);
  }
}
```

Extending the Abstract Class

Now we can create an Unsorted List ADT class by extending the abstract list class. To differentiate this Unsorted List class from the one developed in the previous section, we call it `UnsortedStringList2`. Since constructors cannot be inherited, we must implement our own constructors for this class. Notice how our code for the two constructors both use the single constructor provided in the abstract list class. Additionally, we must complete the definitions of the three abstract classes. We simply reuse the code from the previous implementations. The code for the new unsorted string list is shown below.

```
//--------------------------------------------------------------------------
// UnsortedStringList2.java        by Dale/Joyce/Weems              Chapter 3
//
// Completes the definition of the StringList class under the assumption
// that the list is not kept sorted
//--------------------------------------------------------------------------

package ch03.stringLists;
```

```
public class UnsortedStringList2 extends StringList
{
  public UnsortedStringList2(int maxItems)
  // Instantiates and returns a reference to an empty list object
  // with room for maxItems elements
  {
    super(maxItems);
  }

  public UnsortedStringList2()
  // Instantiates and returns a reference to an empty list object
  // with room for 100 elements
  {
    super(100);
  }

  public boolean isThere (String item)
  // Returns true if item is on this list; otherwise, returns false
  {
    boolean moreToSearch;
    int location = 0;
    boolean found = false;

    moreToSearch = (location < numItems);

    while (moreToSearch && !found)
    {
      if (item.compareTo(list[location]) == 0)  // if they match
        found = true;
      else
      {
        location++;
        moreToSearch = (location < numItems);
      }
    }

   return found;
  }

  public void insert (String item)
  // Adds a copy of item to this list
  {
    list[numItems] = new String(item);
    numItems++;
  }
```

```
  public void delete (String item)
  // Deletes the element that matches item from this list
  {
    int location = 0;

    while (item.compareTo(list[location]) != 0)
      location++;

    list[location] = list[numItems - 1];
    numItems--;
  }
}
```

The UML class diagram in Part (b) of Figure 3.5 models both the abstract `StringList` class and the `UnsortedStringList2` class. Note that the diagram displays the `isThere`, `insert`, and `delete` methods defined in the `StringList` class, and the name of the class itself, in an italic font to indicate that they are abstract classes. Part (a) of the diagram models our original `UnsortedStringList` class, to allow comparison.

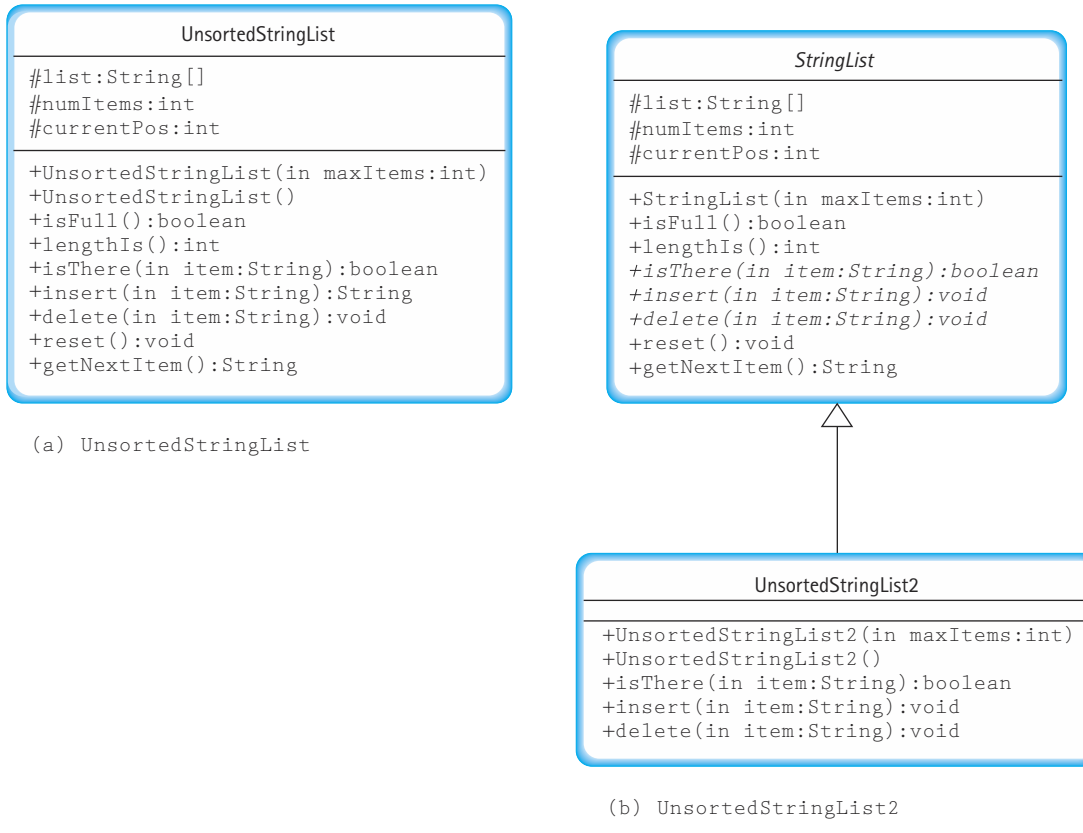


Figure 3.5 *UML diagrams for our list implementations*

3.4 Abstract Data Type Sorted List

At the beginning of this chapter, we said that a list is a linear sequence of items; from any item (except the last) you can access the next one. We looked at the specifications and implementation for the operations that manipulate a list and guarantee this property.

We now want to add an additional property: The key member of any item (except the last) comes before the key member of the next one. We call a list with this property a *sorted list*.

Logical Level

When we defined the specifications for the Unsorted List ADT, we made no requirements with respect to the order in which the list elements are stored and maintained. Now, we have to change the specifications to guarantee that the list is sorted. As was noted in the section Relationship between Unsorted and Sorted Lists of Section 3.3, we must add preconditions and postconditions to those operations for which order is relevant. The only ones that must be changed are `insert` and `delete`.

We call our new class the `SortedStringList` class. We must define new constructors, since their names are directly related to the name of the class.

Sorted List ADT Specification (partial)

Structure:

The list elements are Strings. The list contains unique elements, i.e., no duplicate elements as defined by the key of the list. The strings are kept in alphabetical order. The list has a special property called the current position—the position of the next element to be accessed by `getNextItem` during an iteration through the list. Only `reset` and `getNextItem` affect the current position.

Definitions (provided by user):

maxItems: An integer specifying the maximum number of items to be on this list.

Operations (provided by Sorted List ADT):

void SortedStringList (int maxItems)

Effect: Instantiates this list with capacity of `maxItems` and initializes this list to empty state.

Precondition: `maxItems` > 0

Postcondition: This list is empty.

void SortedStringList ( )

Effect: Instantiates this list with capacity of 100 and initializes this list to empty state.

Postcondition: This list is empty.

void insert (String item)

Effect:	Adds `item` to list.
Preconditions:	List is not full. `item` is not on the list. **List is sorted.**
Postconditions:	`item` is on the list. **List is still sorted.**

void delete (String item)

Effect:	Deletes the element whose key matches `item`'s key.
Preconditions:	One and only one element in list has a key matching `item`'s key. **List is sorted.**
Postconditions:	No element in list has a key matching the argument `item`'s key. **List is still sorted.**

The remaining operations use the same definitions as the Unsorted List ADT.

Application Level

The application level for the Sorted List ADT is the same as for the Unsorted List ADT. As far as the user is concerned, the interfaces are the same. The only functional difference is that when `getNextItem()` is called in the Sorted List ADT, the element returned is the next one in order by key.

Implementation Level

We continue to use the generic list design terminology, created to describe the algorithms for the Unsorted List ADT operations, to describe the algorithms in this section.

insert Operation

To add an element to a sorted list, we must first find the place where the new element belongs, which depends on the value of its key. We use an example to illustrate the insertion operation. Let's say that Becca has made the Honor Roll. To add the element Becca to the sorted list pictured in Figure 3.6(a), maintaining the alphabetic ordering, we must accomplish three tasks:

1. Find the place where the new element belongs.
2. Create space for the new element.
3. Put the new element on the list.

The first task involves traversing the list comparing the new item to each item on the list until we find an item where the new item (in this case, Becca) is less. Recall from Chapter 2

that the `String` method `compareTo` takes a string as a parameter and returns 0 if the parameter string and the object string are equal, returns a positive integer if the parameter string is "less than" the object string, and returns a negative integer if the parameter string is "greater than" the object string. Therefore, we set `moreToSearch` to `false` when we reach a point where `item.compareTo(location.info())` is negative. At this point, `location` is where the new item should go (see Figure 3.6b). If we don't find a place

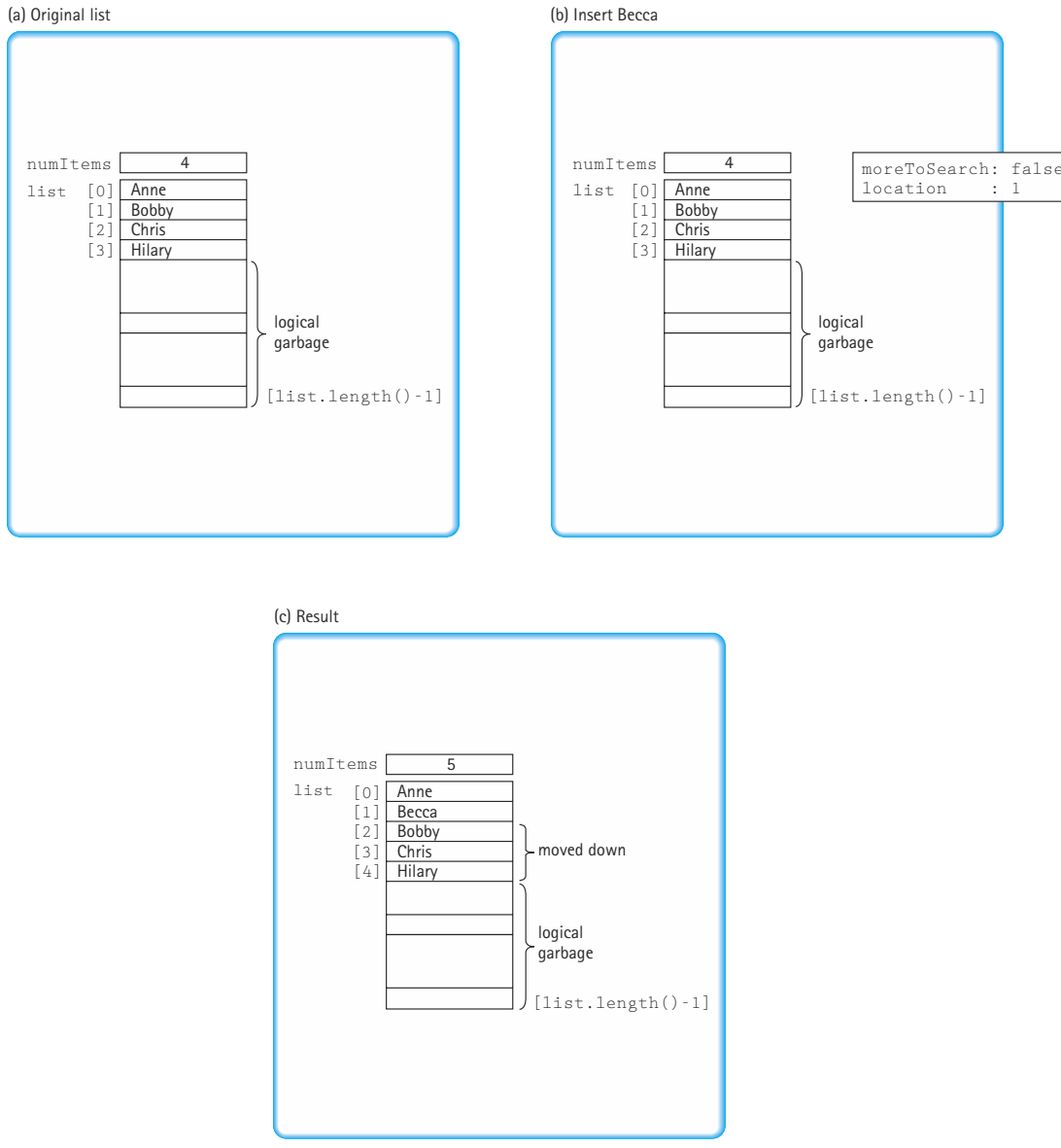


Figure 3.6 *Inserting into a sorted list*

where `item.compareTo(location.info())` is negative, then the item should be put at the end of the list. This is true when `location` equals `numItems`.

Now that we know where the element belongs, we need to create space for it. Because the list is sequential, Becca must be put into the list at location.info( ). But this position may be occupied. To "create space for the new element," we must move down all the list elements that follow it, from `location` through `numItems - 1`. Now we just assign `item` to location.info( ) and increment `numItems`. Figure 3.6(c) shows the resulting list.

Let's summarize these observations in algorithmic form before we write the code.

insert (item)

```
Initialize location to position of first element
Set moreToSearch to (have not examined last.info())
while moreToSearch
  if (item.compareTo(location.info()) < 0)
    Set moreToSearch to false
  else
    Set location to location.next()
    Set moreToSearch to (have not examined last.info())
for index going from numItems DOWNTO location + 1
  Set index.info() to (index-1).info()
Set location.info() to copy of item
Increment numItems
```


Remember that the preconditions on `insert` state that item does not exist on the list, so we do not need to check whether the `compareTo` method returns a zero. Translating the design notation into the array-based implementation gives us the following method.

```
public void insert (String item)
// Adds a copy of item to this list
{
  int location = 0;
  boolean moreToSearch = (location < numItems);

  while (moreToSearch)
  {
    if (item.compareTo(list[location]) < 0)  // Item is less
      moreToSearch = false;
```

```
    else                                        // Item is more
    {
      location++;
      moreToSearch = (location < numItems);
    }
  }

  for (int index = numItems; index > location; index--)
    list[index] = list[index - 1];

  list[location] = new String(item);
  numItems++;
}
```

Does this method work if the new element belongs at the beginning or end of the list? Draw a picture to see how the method works in each of these cases.

delete Operation

When discussing the method `delete` for the Unsorted List ADT, we commented that if the list is sorted, we would have to move the elements up one position to cover the one being removed. Moving the elements up one position is the mirror image of moving the elements down one position. The loop control for finding the item to delete is the same as for the unsorted version.

delete (item)

```
Initialize location to position of first element
while (item.compareTo(location.info()) != 0)
   Set location to location.next()
for index going from location + 1 TO numItems - 1
  Set (index-1).info() to index.info()
Decrement numItems
```

Examine this algorithm carefully and convince yourself that it is correct. Try cases where you are deleting the first item and the last one.

```
public void delete (String item)
// Deletes the element that matches item from this list
{
  int location = 0;
```

```
    while (item.compareTo(list[location]) != 0)    // while not a match
      location++;

    for (int index = location + 1; index < numItems; index++)
      list[index - 1] = list[index];

    numItems--;
  }
```

Improving the isThere Operation

If the list is not sorted, the only way to search for an item is to start at the beginning and look at each element on the list, comparing the key member of the item for which we are searching to the key member of each element on the list in turn. This was the algorithm used in the `isThere` operation in the Unsorted List ADT.

If the list is sorted by key value, there are two ways to improve the searching algorithm. The first way is to stop searching when we pass the place where the item would be if it were there. Look at Figure 3.7(a). If you are searching for Chris, a comparison with Judy would show that Chris is less, that is, the `compareTo` method returns a positive integer. This means that you have passed the place where Chris would be if it were there. At this point you can stop and return `found` as `false`. Figure 3.7(b) shows what happens when you are searching for Susy: `location` is equal to 4, `moreToSearch` is `false`, and `found` is `false`. In this case the search ends because there is nowhere left to look.

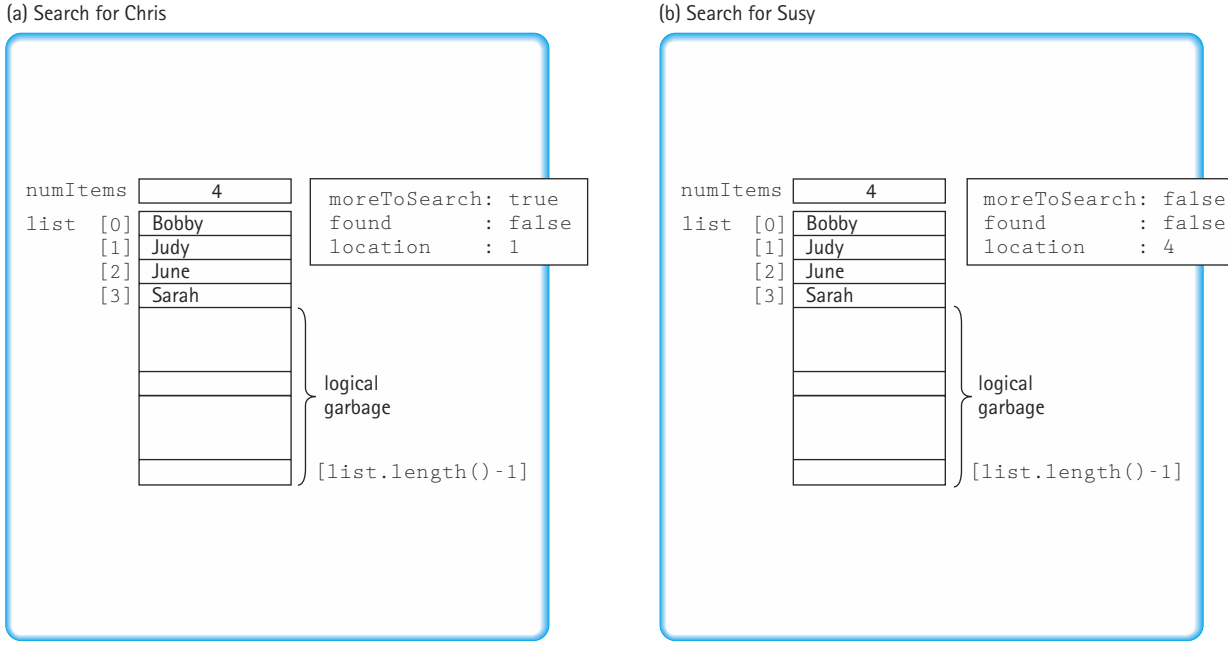


Figure 3.7 *Retrieving in a sorted list*

If the item we are looking for is on the list, the search is the same for the unsorted list and the sorted list. It is when the item is not there that this algorithm is better. We do not have to search all of the elements to determine that the one we want is not there. The second way to improve the algorithm, using a binary search approach, helps in both the case when the item is on the list and the case when the item is not on the list.

Binary Search Algorithm

Think of how you might go about finding a name in a phone book, and you can get an idea of a faster way to search. Let's look for the name "David." We open the phone book to the middle and see that the names there begin with M. M is larger than (comes after) D, so we search the first half of the phone book, the section that contains A to M. We turn to the middle of the first half and see that the names there begin with G. G is larger than D, so we search the first half of this section, from A to G. We turn to the middle page of this section, and find that the names there begin with C. C is smaller than D, so we search the second half of this section—that is, from C to G—and so on, until we are down to the single page that contains the name "David." This algorithm is illustrated in Figure 3.8.

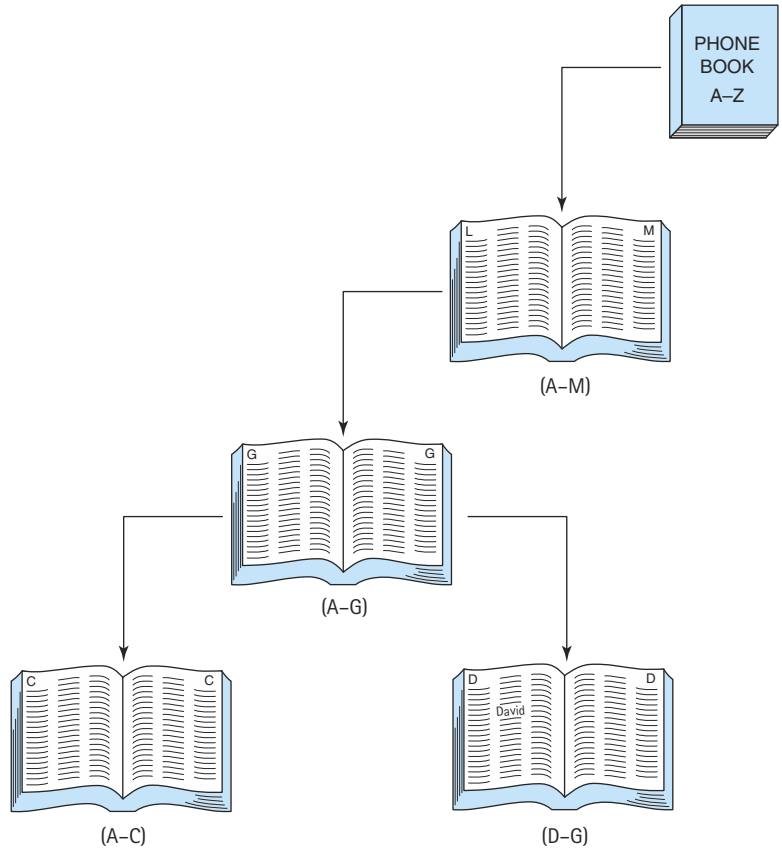


Figure 3.8 *A binary search of the phone book*

The algorithm presented here depends directly on the array-based implementation of the list. This algorithm cannot be implemented with the linked implementation presented in Chapter 5. Therefore, in discussing this algorithm we abandon our generic list design terminology in favor of using array-related terminology.

We begin our search with the whole list to examine; that is, our current search area goes from `list[0]` through `list[numItems - 1]`. In each iteration, we split the current search area in half at the midpoint, and if the item is not found there, we search the appropriate half. The part of the list being searched at any time is the current search area. For instance, in the first iteration of the loop, if a comparison shows that the item comes before the element at the midpoint, the new current search area goes from index 0 through `midpoint - 1`. If the item comes after the element at the midpoint, the new current search area goes from index `midpoint + 1` through `numItems - 1`. Either way, the current search area has been split in half. It looks as if we can keep track of the boundaries of the current search area with a pair of indexes, `first` and `last`. In each iteration of the loop, if an element with the same key as `item` is not found, one of these indexes is reset to shrink the size of the current search area.

How do we know when to quit searching? There are two possible terminating conditions: `item` is not on the list and item has been found. The first terminating condition occurs when there's no more to search in the current search area. Therefore, we only continue searching if `(first <= last)`. The second terminating condition occurs when `item` has been found.

isThere (item): returns boolean

```
Set first to 0
Set last to numItems – 1
Set found to false
Set moreToSearch to (first <= last)
while moreToSearch AND NOT found
  Set midPoint to (first + last) / 2
  compareResult = item.compareTo(midPoint.info())
  if compareResult == 0
    Set found = true
  else if compareResult < 0
    Set last to midPoint – 1
    Set moreToSearch to (first <= last)
  else
    Set first to midPoint + 1
    Set moreToSearch to (first <= last)
return found
```

Notice that when we look in the lower half or upper half, we can ignore the midpoint because we know it is not there. Therefore, `last` is set to `midPoint - 1`, or `first` is set to `midPoint + 1`. The coded version of our algorithm follows.

```
public boolean isThere (String item)
// Returns true if item is on this list; otherwise, returns false
{
  int compareResult;
  int midPoint;
  int first = 0;
  int last = numItems - 1;
  boolean moreToSearch = (first <= last);
  boolean found = false;

  while (moreToSearch && !found)
  {
    midPoint = (first + last) / 2;
    compareResult = item.compareTo(list[midPoint]);

    if (compareResult == 0)
      found = true;
    else if (compareResult < 0)  // Item is less than element at location
    {
      last = midPoint - 1;
      moreToSearch = (first <= last);
    }
    else                         // Item is greater than element at location
    {
      first = midPoint + 1;
      moreToSearch = (first <= last);
    }
  }

 return found;
}
```

Let's do a walk-through of the binary search algorithm. The item being searched for is "bat". Figure 3.9 (a) shows the values of `first`, `last`, and `midpoint` during the first iteration. In this iteration, "bat" is compared with "dog," the value in `list[midpoint]`. Because "bat" is less than (comes before) "dog," `last` becomes `midpoint - 1` and `first` stays the same. Figure 3.9(b) shows the situation during the second iteration. This time, "bat" is compared with "chicken," the value in `list[midpoint]`. Because "bat" is less than (comes before) "chicken," `last` becomes `midpoint - 1` and `first` again stays the same.

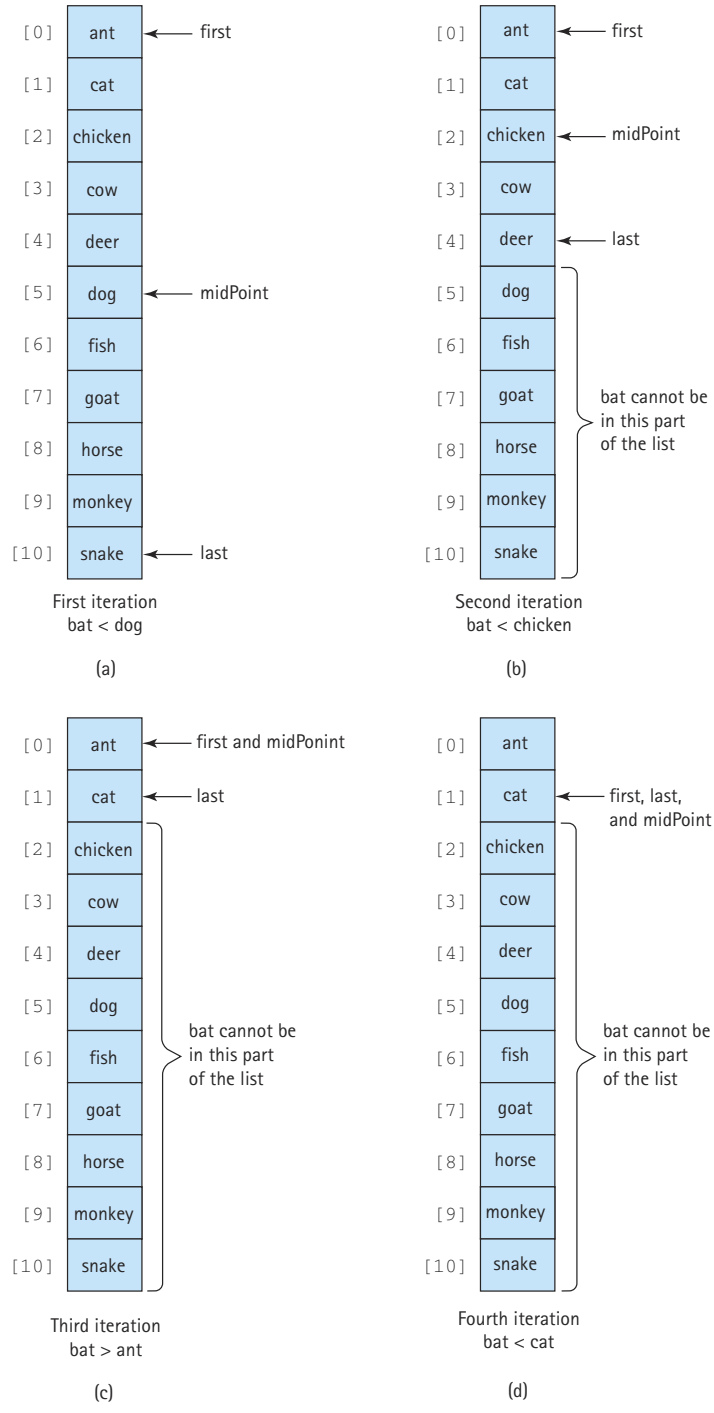


Figure 3.9 *Trace of the binary search algorithm*

In the third iteration (Figure 3.9c), `midpoint` and `first` are both 0. The item "bat" is compared with "ant," the item in `list[midpoint]`. Because "bat" is greater than (comes after) "ant," `first` becomes `midpoint + 1`. In the fourth iteration (Figure 3.9d), `first`, `last`, and `midpoint` are all the same. Again, "bat" is compared with the item in `list[midpoint]`. Because "bat" is less than "cat," `last` becomes `midpoint -1`. Now that `last` is less than `first`, the process stops; `found` is `false`.

The binary search is the most complex algorithm that we have examined so far. The following table shows `first`, `last`, `midpoint`, and `list[midpoint]` for searches of the items "fish," "snake," and "zebra," using the same data as in the previous example. Examine the results of Table 3.1 carefully.

Notice that the loop never executes more than four times. It never executes more than four times in a list of 11 components because the list is being cut in half each time through the loop. Table 3.2 compares a linear search and a binary search in terms of the average number of iterations needed to find an item.

If the binary search is so much faster, why not use it all the time? It is certainly faster in terms of the number of times through the loop, but more computations are executed within the binary search loop than in the other search algorithms. So if the number of components on the list is small (say, under 20), linear search algorithms are faster because they perform less work at each iteration. As the number of components on the list increases, the binary search algorithm becomes relatively more efficient. Remember, however, that the binary search requires the list to be sorted and sorting takes time.

The UML diagram for the `SortedStringList` class is displayed in Figure 3.10, along with the diagrams for the previous list implementations for comparison purposes.

Table 3.1 *Trace of binary search algorithm*

Iteration	first	last	midPoint	list[midPoint]	Terminating Condition
item: fish					
First	0	10	5	dog	
Second	6	10	8	horse	
Third	6	7	6	fish	found is true
item: snake					
First	0	10	5	dog	
Second	6	10	8	horse	
Third	9	10	9	camel	
Fourth	10	10	10	snake	found is true
item: zebra					
First	0	10	5	dog	
Second	6	10	8	horse	
Third	9	10	9	camel	
Fourth	10	10	10	snake	
Fifth	11	10			last < first

Table 3.2 *Comparison of linear and binary search*

	Average Number of Iterations	
Length	Linear Search	Binary Search
10	5.5	2.9
100	50.5	5.8
1,000	500.5	9.0
10,000	5000.5	12.0

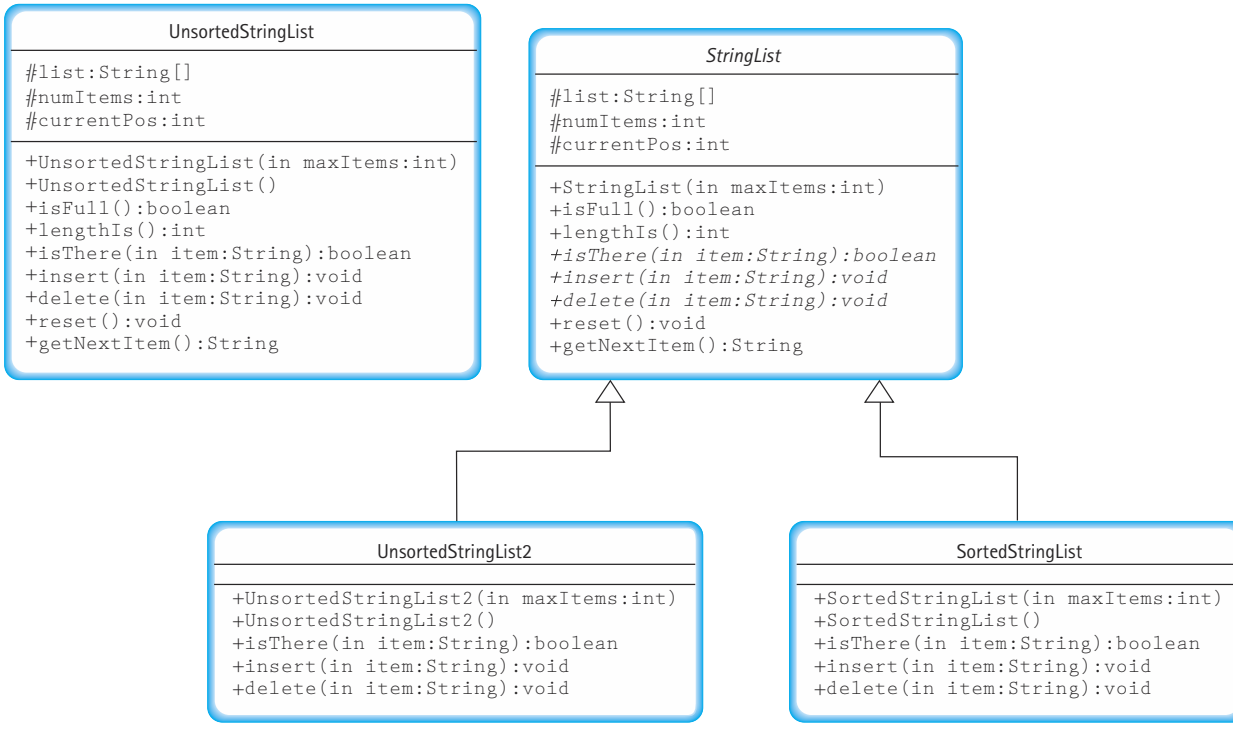


Figure 3.10 *UML diagrams for our list implementations*

Test Plan

We can use the same test plan that we used for the unsorted list, with the expected outputs changed to reflect the ordering. However, we should add some test cases to explicitly address the fact that the list is sorted. For example, we should insert a sequence of strings in reverse alphabetical order and check if the ADT correctly orders them. Note that the sorted list implementation described in this section can be found in the file `SortedStringList.java` on our web site.

3.5 Comparison of Algorithms

As we have shown in this chapter, there is more than one way to solve most problems. If you were asked for directions to Joe's Diner (see Figure 3.11), you could give either of two equally correct answers:

1. "Go east on the big highway to the Y'all Come Inn, and turn left."
2. "Take the winding country road to Honeysuckle Lodge, and turn right."

The two answers are not the same, but because following either route gets the traveler to Joe's Diner, both answers are functionally correct.

If the request for directions contained special requirements, one solution might be preferable to the other. For instance, "I'm late for dinner. What's the quickest route to

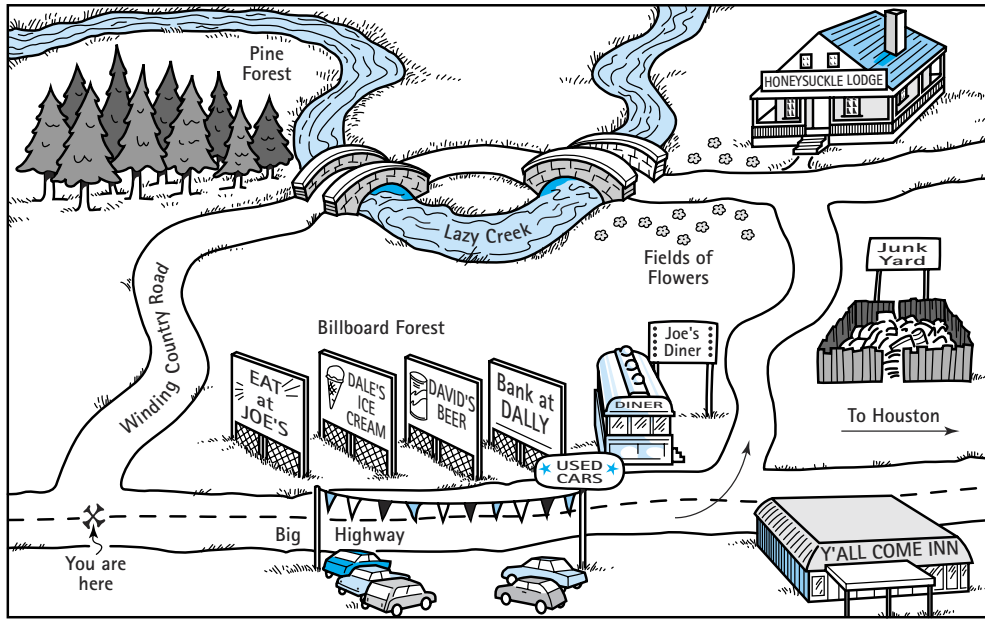


Figure 3.11 *Map to Joe's Diner*

Joe's Diner?" calls for the first answer, whereas "Is there a scenic road that I can take to get to Joe's Diner?" suggests the second. If no special requirements are known, the choice is a matter of personal preference—which road do you like better?

In this chapter, we have presented many algorithms. How we choose between two algorithms that do the same task often depends on the requirements of a particular application. If no relevant requirements exist, the choice may be based on the programmer's own style.

Often the choice between algorithms comes down to a question of efficiency. Which one takes the least amount of computing time? Which one does the job with the least amount of work? We are talking here of the amount of work that the computer does. Later we also compare algorithms in regard to how much work the programmer does. (One is often minimized at the expense of the other.)

To compare the work done by competing algorithms, we must first define a set of objective measures that can be applied to each algorithm. The analysis of algorithms is an important area of theoretical computer science; in advanced courses students undoubtedly see extensive work in this area. In this text you learn about a small part of this topic, enough to let you determine which of two algorithms requires less work to accomplish a particular task.

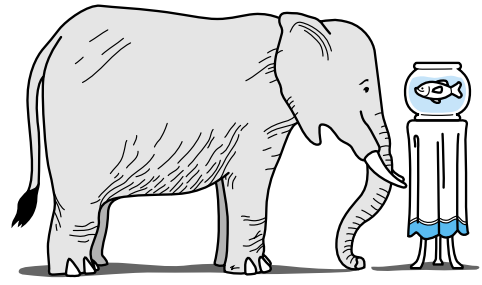
How do programmers measure the work that two algorithms perform? The first solution that comes to mind is simply to code the algorithms and then compare the execution times for running the two programs. The one with the shorter execution time is clearly the better algorithm. Or is it? Using this technique, we really can determine only that program A is more efficient than program B on a particular computer at a particular time. Execution times are specific to a *particular computer*, since different computers run at different speeds. Sometimes they are dependent on what else the computer is doing in the background, for example if the Java run-time engine is performing garbage collection, it can affect the execution time of the program. Of course, we could test the algorithms on many possible computers at various times, but that would be unrealistic and too specific (new computers are becoming available all the time). We want a more general measure.

A second possibility is to count the number of instructions or statements executed. This measure, however, varies with the programming language used, as well as with the style of the individual programmer. To standardize this measure somewhat, we could count the number of passes through a critical loop in the algorithm. If each iteration involves a constant amount of work, this measure gives us a meaningful yardstick of efficiency.

Another idea is to isolate a particular operation fundamental to the algorithm and count the number of times that this operation is performed. Suppose, for example, that we are summing the elements in an integer list. To measure the amount of work required, we could count the integer addition operations. For a list of 100 elements, there are 99 addition operations. Note, however, that we do not actually have to count the number of addition operations; it is some *function* of the number of elements (N) on the list. Therefore, we can express the number of addition operations in terms of N: For a list of N elements, there are $N - 1$ addition operations. Now we can compare the algorithms for the general case, not just for a specific list size.

Sometimes an operation so dominates an algorithm that the other operations fade into the background "noise." If we want to buy elephants and goldfish, for example, and we are considering two pet suppliers, we only need to compare the prices of elephants;

the cost of the goldfish is trivial in comparison. Suppose we have two files of integers, and we want to create a new file of integers based on the sums of pairs of integers from the existing files. In analyzing an algorithm that solves this problem, we could count both file accesses and integer additions. However, file accessing is so much more expensive than integer addition in terms of computer time, that the integer additions could be a trivial factor in the efficiency of the whole algorithm; we might as well count only the file accesses, ignoring the integer additions. In analyzing algorithms, we often can find one operation that dominates the algorithm, effectively relegating the others to the "noise" level.



Big-O

We have been talking about work as a function of the size of the input to the operation (for instance, the number of elements on the list to be summed). We can express an approximation of this function using a mathematical notation called order of magnitude, or Big-O notation. (This is a letter O, not a zero.) The order of magnitude of a function is identified with the term in the function that increases fastest relative to the size of the problem. For instance, if

Big-O notation A notation that expresses computing time (complexity) as the term in a function that increases most rapidly relative to the size of a problem

$$f(N) = N^4 + 100N^2 + 10N + 50$$

then $f(N)$ is of order N^4—or, in Big-O notation, $O(N^4)$. That is, for large values of N, some multiple of N^4 dominates the function for sufficiently large values of N.

How is it that we can just drop the low-order terms? Remember the elephants and goldfish that we talked about earlier? The price of the elephants was so much greater that we could just ignore the price of the goldfish. Similarly, for large values of N, N^4 is so much larger than 50, $10N$, or even $100N2$ that we can ignore these other terms. This doesn't mean that the other terms do not contribute to the computing time; it only means that they are not significant in our approximation when N is "large."

What is this value N? N represents the size of the problem. Most of the rest of the problems in this book involve data structures—lists, stacks, queues, and trees. Each structure is composed of elements. We develop algorithms to add an element to the structure and to modify or delete an element from the structure. We can describe the work done by these operations in terms of N, where N is the number of elements in

the structure. Yes, we know. We have called the number of elements in a list the length of the list. However, mathematicians talk in terms of *N*, so we use *N* for the length when we are comparing algorithms using Big-O notation.

Suppose that we want to write all the elements in a list into a file. How much work is that? The answer depends on how many elements are on the list. Our algorithm is

Write List Elements

```
Open the file
while more elements in list
    Write the next element
```

If *N* is the number of elements on the list, the "time" required to do this task is

*(N * time-to-write-one-element) + time-to-open-the-file*

This algorithm is O(*N*) because the time required to perform the task is proportional to the number of elements (*N*)—plus a little to open the file. How can we ignore the open time in determining the Big-O approximation? If we assume that the time necessary to open a file is constant, this part of the algorithm is our goldfish. If the list has only a few elements, the time needed to open the file may seem significant, but for large values of *N*, writing the elements is an elephant in comparison with opening the file.

The order of magnitude of an algorithm does not tell you how long in microseconds the solution takes to run on your computer. Sometimes we need that kind of information. For instance, a word processor's requirements state that the program must be able to spell-check a 50-page document (on a particular computer) in less than 120 seconds. For information like this, we do not use Big-O analysis; we use other measurements. We can compare different implementations of a data structure by coding them and then running a test, recording the time on the computer's clock before and after. This kind of "benchmark" test tells us how long the operations take on a particular computer, using a particular compiler. The Big-O analysis, however, allows us to compare algorithms without reference to these factors.

Common Orders of Magnitude

O(1) is called bounded time. The amount of work is bounded by a constant and is not dependent on the size of the problem. Assigning a value to the *i*th element in an array of *N* elements is O(l) because an element in an array can be accessed directly through its index. Although bounded time is often called constant time, the amount of work is not necessarily constant. It is, however, bounded by a constant.

$O(\log_2 N)$ is called *logarithmic* time. The amount of work depends on the log of the size of the problem. Algorithms that successively cut the amount of data to be processed in half at each step typically fall into this category. Finding a value in a list of sorted elements using the binary search algorithm is $O(\log_2 N)$.

$O(N)$ is called *linear* time. The amount of work is some constant times the size of the problem. Printing all the elements in a list of N elements is $O(N)$. Searching for a particular value in a list of unsorted elements is also $O(N)$ because you must potentially search every element on the list to find it.

$O(N \log_2 N)$ is called (for lack of a better term) $N \log_2 N$ time. Algorithms of this type typically involve applying a logarithmic algorithm N times. The better sorting algorithms, such as Quicksort, Heapsort, and Mergesort discussed in Chapter 10, have $N \log_2 N$ complexity. That is, these algorithms can transform an unsorted list into a sorted list in $O(N \log_2 N)$ time.

$O(N^2)$ is called *quadratic* time. Algorithms of this type typically involve applying a linear algorithm N times. Most simple sorting algorithms are $O(N^2)$ algorithms. (See Chapter 10.)

$O(2^N)$ is called *exponential* time. These algorithms are extremely costly. An example of a problem for which the best known solution is exponential is the traveling salesman problem—given a set of cities and a set of roads that connect some of them, plus the lengths of the roads, find a route that visits every city exactly once and minimizes total travel distance. As you can see in Table 3.3, exponential times increase dramatically in relation to the size of N. (It also is interesting to note that the values in the last column grow so quickly that the computation time required for problems of this order may exceed the estimated life span of the universe!)

Note that throughout this discussion we have been talking about the amount of work the computer must do to execute an algorithm. This determination does not necessarily relate to the size of the algorithm, say, in lines of code. Consider the following two algorithms to initialize to zero every element in an N-element array.

Algorithm Init1

```
items[0] = 0;
items[1] = 0;
items[2] = 0;
items[3] = 0;
.
.
.
items[N-1] = 0;
```

Algorithm Init2

```
for (index = 0; index < N; index++)
    items[index] = 0;
```

Both algorithms are $O(N)$, even though they greatly differ in the number of lines of code.

Table 3.3 *Comparison of rates of growth*

N	$\log_2 N$	$N \log_2 N$	N^2	N^3	2^N
1	0	1	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4,096	65,536
32	5	160	1,024	32,768	4,294,967,296
64	6	384	4,096	262,144	About 1 month's worth of instructions on a supercomputer
128	7	896	16,384	2,097,152	About 10^{12} times greater than the age of the universe in nanoseconds (for a 6-billion-year estimate)
256	8	2,048	65,536	16,777,216	Don't ask!

Now let's look at two different algorithms that calculate the sum of the integers from 1 to *N*. Algorithm Sum1 is a simple *for* loop that adds successive integers to keep a running total:

Algorithm Sum1

```
sum = 0;
for (count = 1; count <= n; count++)
   sum = sum + count;
```

That seems simple enough. The second algorithm calculates the sum by using a formula. To understand the formula, consider the following calculation when $N = 9$.

```
  1 +  2 +  3 +  4 +  5 +  6 +  7 +  8 +  9
+ 9 +  8 +  7 +  6 +  5 +  4 +  3 +  2 +  1
-------------------------------------------
 10 + 10 + 10 + 10 + 10 + 10 + 10 + 10 + 10  =  10 * 9 = 90
```

We pair up each number from 1 to N with another, such that each pair adds up to $N + 1$. There are N such pairs, giving us a total of $(N + 1)*N$. Now, because each number is included twice, we divide the product by 2. Using this formula, we can solve the problem: ((9 + 1) * 9)/2 = 45. Now we have a second algorithm:

Algorithm Sum2

```
sum = ((n + 1) * n) / 2;
```

Both of the algorithms are short pieces of code. Let's compare them using Big-O notation. The work done by Sum1 is a function of the magnitude of N; as N gets larger, the amount of work grows proportionally. If N is 50, Sum1 works 10 times as hard as when N is 5. Algorithm Sum1, therefore, is $O(N)$.

To analyze Sum2, consider the cases when $N = 5$ and $N = 50$. They should take the same amount of time. In fact, whatever value we assign to N, the algorithm does the same amount of work to solve the problem. Algorithm Sum2, therefore, is $O(1)$.

Does this mean that Sum2 is always faster? Is it always a better choice than Sum1? That depends. Sum2 might seem to do more "work," because the formula involves multiplication and division, whereas Sum1 is a simple running total. In fact, for very small values of N, Sum2 actually might do more work than Sum1. (Of course, for very large values of N, Sum1 does a proportionally larger amount of work, whereas Sum2 stays the same.) So the choice between the algorithms depends in part on how they are used, for small or large values of N.

Another issue is the fact that Sum2 is not as obvious as Sum1, and thus it is harder for the programmer (a human) to understand. Sometimes a more efficient solution to a problem is more complicated; we may save computer time at the expense of the programmer's time.

So, what's the verdict? As usual in the design of computer programs, there are tradeoffs. We must look at our program's requirements and then decide which solution is better. Throughout this text we examine different choices of algorithms and data structures. We compare them using Big-O, but we also examine the program's requirements and the "elegance" of the competing solutions. As programmers, we design software solutions with many factors in mind.

Family Laundry: An Analogy

How long does it take to do a family's weekly laundry? We might describe the answer to this question with the function

$$f(N) = c * N$$

where N represents the number of family members and c is the average number of minutes that each person's laundry takes. We say that this function is O(N) because the total laundry time depends on the number of people in the family. The "constant" c may vary a little for different families—depending on the size of their washing machine and how fast they can fold clothes, for instance. That is, the time to do the laundry for two different families might be represented with these functions:

$$f(N) = 100 * N$$
$$g(N) = 90 * N$$

But overall, we describe these functions as O(N).

Now what happens if Grandma and Grandpa come to visit the first family for a week or two? The laundry time function becomes

$$f(N) = 100 * (N + 2)$$

We still say that the function is O(N). How can that be? Doesn't the laundry for two extra people take any time to wash, dry, and fold? Of course it does! If N is small (the family consists of Mother, Father, and Baby Sierra), the extra laundry for two people is significant. But as N grows large (the family consists of Mother, Father, 8 kids, and a dog named Waldo), the extra laundry for two people doesn't make much difference. (The family's laundry is the elephant; the guest's laundry is the goldfish.) When we compare algorithms using Big-O, we are concerned with what happens when N is "large."

If we are asking the question "Can we finish the laundry in time to make the 7:05 train?" we want a precise answer. The Big-O analysis doesn't give us this information. It gives us an approximation. So, if 100 * N, 90 * N, and 100 * (N + 2) are all O(N), how can we say which is better? We can't—in Big-O terms, they are all roughly equivalent for large values of N. Can we find a better algorithm for getting the laundry done? If the family wins the state lottery, they can drop all their dirty clothes at a professional laundry 15 minutes' drive from their house (30 minutes round trip). Now the function is

$$f(N) = 30$$

This function is O(1). The answer is not dependent on the number of people in the family. If they switch to a laundry 5 minutes from their house, the function becomes

$$f(N) = 10$$

This function is also O(1). In terms of Big-O, the two professional-laundry solutions are equivalent: No matter how many family members or houseguests you have, it takes a constant amount of the family's time to do the laundry. (We aren't concerned with the professional laundry's time.)

3.6 Comparison of Unsorted and Sorted List ADT Algorithms

In order to determine the Big-O notation for the complexity of these algorithms, we must first determine the size factor. Here we are considering algorithms to manipulate items in a list. Therefore, the size factor is the number of items on the list: `numItems`.

Many of our algorithms are identical for the Unsorted List ADT and the Sorted List ADT. We capitalized on this fact in Section 3.4 when we brought the corresponding methods together in our abstract list class. Let's examine these first. The `lengthIs` and `isFull` methods each contain only one statement: `return numItems` and `return (list.length == numItems)`. Since the number of statements executed in these methods does not depend on the number of items on the list, they have O(1) complexity. The `reset` method contains one assignment statement and `getNextItem` contains an assignment statement, an *if-then-else* statement, and a *return* statement. Neither of these methods is dependent on the number of items on the list, so they also have O(1) complexity. The other methods are different for the two implementations.

Unsorted List ADT

The algorithm for `isThere` requires that the list be searched until an item is found or the end of the list is reached. We might find the item in any position on the list, or we might not find it at all. How many places must we examine? At best only one, at worst `numItems`. If we took the best case as our measure of complexity, then all of the operations would have O(1) complexity. But this is a rare case. What we want is the average case or worst case, which in this instance are the same: O(`numItems`). True, the average case would be O(`numItems/2`), but when we are using order notation, O(`numItems`) and O(`numItems/2`) are equivalent. In some cases that we discuss later, the average and the worst cases are not the same.

The `insert` algorithm has two parts: find the place to insert the item and insert the item. In the unsorted list, the item is put in the `numItems` position and `numItems` is incremented. Neither of these operations is dependent on the number of items on the list, so the complexity is O(1).

The `delete` algorithm has two parts: find the item to delete and delete the item. Finding the item uses essentially the same algorithm as `isThere`. The only difference is that since it is guaranteed that the item is on the list, we do not have to test for the end-of-list condition. But that difference does not affect the number of times we may have to traverse the search loop, so the complexity of that part is O(`numItems`). To delete the item, we put the value in the `numItems - 1` position into the location of the item to be deleted and decrement `numItems`. This store and decrement are not dependent on the number of items on the list, so this part of the operation has complexity O(1). The entire delete algorithm has complexity O(`numItems`) because O(`numItems`) plus O(1) is O(`numItems`). (Remember, the O(1) is the goldfish.)

Sorted List ADT

We looked at three different algorithms for `isThere`. We said that the Unsorted List ADT algorithm would work for a sorted list but that there were two more efficient algorithms: a linear search in the sorted list that exits when the place where the item would be is passed and a binary search.

A linear search in a sorted list is faster than in an unsorted list when searching for an item that is not on the list, but is the same when searching for an item that is on the list. Therefore, the complexity of the linear search in a sorted list is the same as the complexity in an unsorted list: O(`numItems`). Does that mean that we shouldn't bother taking advantage of the ordering in our search? No, it just means that the Big-O complexity measures are the same.

What about the binary search algorithm? We showed a table comparing the number of items searched in a linear search versus a binary search for certain sizes of lists. How do we describe this algorithm using Big-O notation? To figure this out, let's see how many times we can split a list of N items in half. Assuming that we don't find the item we are looking for at one of the earlier midpoints, we have to divide the list $\log_2 N$ times at the most, before we run out of elements to split. In case you aren't familiar with logs,

$$2^{log_2 N} = N$$

The definition of $\log_2 N$ is "the number that you raise 2 to, to get N". So, if we raise 2 to that number, $2^{\log_2 N}$, the result is N. Consider, for example, that if $N = 1024$, $\log_2 N = 10$, and $2^{10} = 1024$. How does that apply to our searching algorithms? The sequential search is $O(N)$; in the worst case, we would have to search all 1024 elements of the list. The binary search is $O(\log_2 N)$; in the worst case we would have to make $\log_2 N + 1$, or 11, search comparisons. A heuristic (a rule of thumb) tells us that a problem that is solved by successively splitting it in half is an $O(\log_2 N)$ algorithm. Figure 3.12 illustrates the relative growth of the linear and binary searches, measured in number of comparisons.

The `insert` algorithm still has the same two parts: finding the place to insert the item and inserting the item. Because the list must remain sorted, we must search for the position into which the new item must go. Our algorithm used a linear search to find

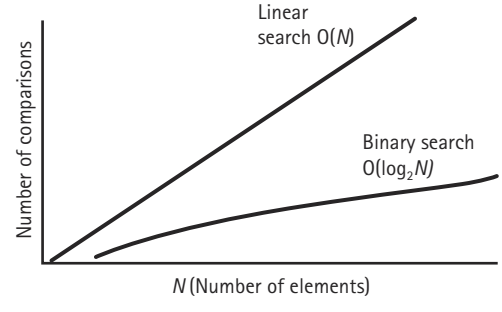


Figure 3.12 *Comparison of linear and binary searches*

the appropriate location: O(numItems). Inserting requires that we move all those elements from the insertion point down one place in the array. How many items must we move? At most numItems, giving us O(numItems). O(numItems) plus O(numItems) is O(numItems) because we disregard the constant 2. Note, however, that the constant 2 does not actually occur here. We actually access each item on the list only once except for the item at the insertion point: We access those to the place of insertion and we move those items stored from numItems - 1 through that place. Therefore, only the element in the insertion location is accessed twice: once to find the insertion point and once to move it.

You may have thought of an even more efficient way to insert the item. You could start at the end of the list and repeatedly test to see if that is where you need to put the item. If the item is larger then the element at the end of the list, you just insert it following that element; if it is not you move the list element at the end of the list down one array position, and check the next to last list element, repeating the same pattern of compare and move. By the time you find out where to insert the item, you have already shifted all of the elements that are greater than down one location in the array, and you can just insert it into the open location. With this approach, on average, you only have to access half of the elements in the array, instead of all of the elements. However, it is still the same complexity as the other approach, since O(numItems/2) is equal to O(numItems).

The delete algorithm also still has the same two parts: finding the item to delete and deleting the item. The algorithm for finding the item is the mirror image of finding the insertion point: O(numItems). Deleting the item in a sorted list requires that all the elements from the deletion location to the end of the list must be moved forward one position. This shifting algorithm is the reverse of the shifting algorithm in the insertion and, therefore, has the same complexity: O(numItems). Hence the complexities of the insertion and deletion algorithms are the same in the Sorted List ADT.

Table 3.4 summarizes these complexities. We have replaced numItems with N, the generic name for the size factor.

In the deletion operation, we could improve the efficiency by using the binary search algorithm to find the item to delete. Would this change the complexity? No, it would not. The find would be $O(\log_2 N)$, but the removal would still be $O(N)$; since $O(\log_2 N)$ combined with $O(N)$ is $O(N)$ we have not changed the overall complexity of the algorithm. (Recall that the term with the largest power of N dominates.) Does this mean that we should not use the binary search algorithm? No, it just means that as the length of the list grows, the cost of the removal dominates the cost of the find.

Think of the common orders of complexity as being bins into which we sort algorithms (Figure 3.13). For small values of the size factor, an algorithm in one bin may actually be faster than the equivalent algorithm in the next-more-efficient bin. As the size factor increases, the differences among algorithms in the different bins get larger. When choosing between algorithms within the same bin, you look at the constants to determine which to use.

Table 3.4 *Big-O comparison of list operations*

Operation	Unsorted List	Sorted List
`length`	O(1)	O(1)
`isFull`	O(1)	O(1)
`reset`	O(1)	O(1)
`getNextItem`	O(1)	O(1)
`isThere`	O(*N*)	O(*N*)
		O($\log_2 N$) binary search
`insert`		
Find	O(1)	O(*N*)
Put	O(1)	O(*N*)
Combined	O(1)	O(*N*)
`delete`		
Find	O(*N*)	O(*N*)
Put	O(1)	O(*N*)
Combined	O(*N*)	O(*N*)

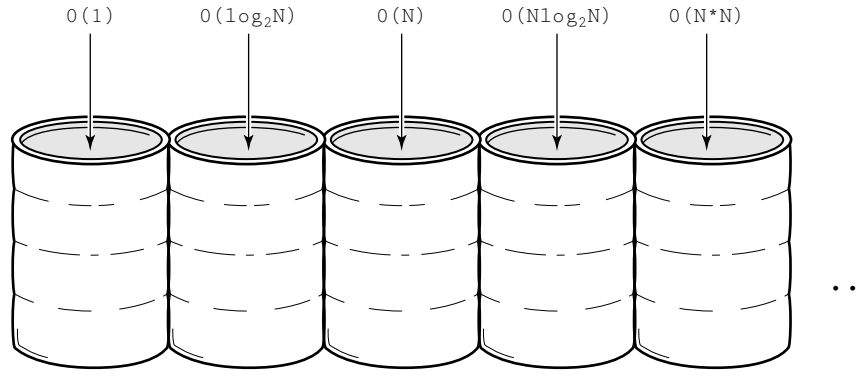


Figure 3.13 *Complexity bins*

3.7 Generic ADTs

So far in this chapter we have created several variations of list ADTs: a "standalone" unsorted string list, an unsorted string list that extended an abstract list class, and a sorted string list, that also extended the abstract list class. These string lists are very useful to an application programmer who is creating a system that requires lists of strings. But what if the programmer wanted some other kind of list: a list of integers, a list of dates, a list of circles, a list of real estate information?

The list ADTs we have constructed so far have all been constrained to holding data of one specific type, namely strings. While useful, think of how much more useful they would be if they could hold any kind of information. A generic data type is one for which the operations are defined but the types of the items being manipulated are not. We can make our lists generic by using Java's *interface* construct. We limited ourselves to lists of strings up until now, because we wanted to concentrate on the list operations without dealing with the extra complexity of interfaces. Now, however, we are ready to see how we can construct more generally usable ADTs.

Generic data type A type for which the operations are defined but the types of the items being manipulated are not

We use a new package, `ch03.genericLists`, to organize our files related to generic lists. As required, the files are placed in a subdirectory `genericLists` of the subdirectory `ch03` of the directory `bookFiles`. Additionally, each of the class files must begin with the line

```
package ch03.genericLists;
```

Lists of Objects

One approach to creating generic ADTs is to have our ADTs use variables of type `Object`. Since all Java classes ultimately inherit from `Object`, such an ADT should be able to "hold" a variable of any class. If you try this approach, you soon see that it has severe limitations.

Consider what happens if you redefine our `SortedStringList` class to hold objects instead of strings. If you edit the file containing the class, and change every place where you see "String" with "Object", you have created a `SortedObjectList` class. At first glance this seems to have solved our problem. The list is implemented as an array of objects. We can insert objects into the list and delete them. Many of the methods, like `isFull` and `reset`, are not even affected by the change. However, when you try to compile the file you discover a few errors. For example, the following line from the `insert` method of the new file is flagged with a "method not found" message:

```
if (item.compareTo(list[location]) < 0)
```

Do you see why? Remember that the `item` referred to in the code is now of class `Object`. If you check the definition of the `Object` class you see that it does not include a `compareTo` method. Therefore, this statement, along with several other statements in the file, is syntactically illegal. The statement was OK when `item` was a string, since the `String` class includes a `compareTo` method, but it is not a legal statement when `item` is an object of the more general `Object` class.

The `String` class's `compareTo` method returns information about the relative ordering of two strings. Such a method is not defined for the `Object` class, since it might not always make sense to talk about the ordering of two objects. We cannot have a sorted list of just any type of objects. We can only have a sorted list of objects for which a relative ordering has been defined.

There is one other kind of statement that is flagged by the compiler. This statement also appears in the `insert` method:

```
list[location] = new Object(item);
```

You should recall that this statement is executed after the method has shifted the array values to make room for the new item and set the value of `location` to the insertion location. The previous form of this statement used the `String` class's copy constructor, `String(item)`, to create a new string object, which was then inserted into the list. The new form of this statement attempts to use a copy constructor from the `Object` class, but no such constructor exists. The reason we wish to insert a copy of the item into the list, instead of just inserting the item itself, is to preserve the information hiding aspect of our ADT.

To solve these problems we create a Java interface with abstract classes for comparing and copying objects.

The Listable Interface

To ensure that the objects that we place on our list support the necessary methods, we create a Java interface. Recall from Chapter 2 that an interface can only include abstract methods, that is, methods without bodies. Once the interface is defined we can create classes that implement the interface by supplying the missing method bodies.

For our lists we create an interface with two abstract methods; one to compare elements so that we can support sorted lists and the `isThere` operation, and one to support copying of list elements, so that we can maintain information hiding. We follow the Java convention used in the `String` class by naming the former method `compareTo` and by having it return integer values to indicate the result of the comparison. We call the latter method `copy`. It does not need any parameters; it simply returns a copy of the object on which it is invoked. Finally, we need a name for the interface itself. Let's call it `Listable`, since classes that implement this interface provide objects that can be listed.

Here is the code for the interface:

```
package ch03.genericLists;

public interface Listable
// Objects of classes that implement this interface can be used with lists
```

```
{
  public abstract int compareTo(Listable other);
  // Compares this Listable object to "other". If they are equal, 0 is
  //   returned
  // If this is less than the argument, a negative value is returned
  // If this is more than the argument, a positive value is returned

  public abstract Listable copy();
  // Returns a new object with the same contents as this Listable object
}
```

Whatever data we intend to store on a list must be contained in a class that implements the `Listable` interface. For example, to support a list of circles we might define a `ListCircle` class as follows (some of the code that is not pertinent to this discussion has been left out):

```
package ch03.genericLists;

public class ListCircle implements Listable
{
  private int xvalue;      // Horizontal position of center
  private int yvalue;      // Vertical position of center
  private float radius;
  private boolean solid;   // True means circle filled

  // Code for Constructors goes here

  public int compareTo(Listable otherCircle)
  {
    ListCircle other = (ListCircle)otherCircle;
    return (int)(this.radius - other.radius);
  }

  public Listable copy()
  {
    ListCircle result = new ListCircle(this.xvalue, this.yvalue, this.radius,
                                       this.solid);
    return result;
  }

  // More ListCircle methods as needed

}
```

Note the use of the cast operation `(ListCircle)` in the `compareTo` method:

```
ListCircle other = (ListCircle)otherCircle;
```

This is to ensure that the parameter `otherCircle` is a `ListCircle`. The method signature allows it to be any `Listable` type, yet on the following line we are assuming that it is a `ListCircle`, when we access its `radius` instance variable.

Since `ListCircle` implements `Listable`, it can be used anywhere something of type `Listable` is expected. In the next section we define a class that provides a list of `Listable` objects. This class could therefore be used to provide a list of `ListCircle` objects.

A Generic Abstract List Class

Now we can create our generic list ADT by defining a list of `Listable` elements; not just strings, not just plain objects, but objects of classes that implement the `Listable` interface. We can reuse the code from our previous list definitions, but we must replace the use of the `String` class with the `Listable` interface throughout the code; we also must replace the use of the `String` class's copy constructor with statements that use the `copy` method defined in the interface.

We no longer need to use the term "string" when defining our list classes, since they are no longer constrained to providing only lists of strings. We call our new abstract list class simply `List`. Below is the code for the abstract `List` class. There are several things to notice about the code. First, note the use of the term `Listable`, in place of a class or type name, throughout the code. Wherever `Listable` is used to represent a formal parameter, you can pass an object of a class that implements `Listable`, as the actual parameter. For example, you could use objects of type `ListCircle`, which was defined in the previous subsection. Alternately, if you have defined other classes that implement the `Listable` interface, you could use objects of those classes—perhaps a class of `ListStrings` or a class of `ListStudents`.

Also, note the invocation of the `copy` method on the `next` object, in the very last statement of the class. The `next` object is of "type" `Listable`, that is, it is an object of a class that implements `Listable`. Therefore, we can be assured that the creator of that class has included a definition of the `copy` method within the class.

Finally, you should notice the addition of a new list method, `retrieve`, and some small but important changes to the comments describing the effects of the methods `isThere` and `delete`. The switch from supporting lists of strings to lists of `Listable` objects means that we now can implement and use lists of composite elements. This raises some interesting questions about how we compare elements, and what it means for two elements to be "equal." These questions are discussed following the code listing.

```
//--------------------------------------------------------------------------
// List.java                by Dale/Joyce/Weems                  Chapter 3
//
// Defines all constructs for an array based list that do not depend
```

```
// on whether or not the list is sorted.
//--------------------------------------------------------------------------

package ch03.genericLists;

public abstract class List
{
  protected Listable[] list;          // Array to hold this list's elements
  protected int numItems;             // Number of elements on this list
  protected int currentPos;           // Current position for iteration

  public List(int maxItems)
  // Instantiates and returns a reference to an empty list object
  // with room for maxItems elements
  {
    numItems = 0;
    list = new Listable[maxItems];
  }

  public boolean isFull()
  // Returns whether this list is full
  {
    return (list.length == numItems);
  }

  public int lengthIs()
  // Returns the number of elements on this list
  {
    return numItems;
  }

  public abstract boolean isThere (Listable item);
  // Returns true if an element with the same key as item is on this list;
  // otherwise, returns false

  public abstract Listable retrieve(Listable item);
  // Returns a copy of the list element with the same key as item

  public abstract void insert (Listable item);
  // Adds a copy of item to this list

  public abstract void delete (Listable item);
  // Deletes the element with the same key as item from this list

  public void reset()
  // Initializes current position for an iteration through this list
```

```
  {
    currentPos  = 0;
  }

  public Listable getNextItem ()
  // Returns copy of the next element on this list
  {
    Listable next = list[currentPos];
    if (currentPos == numItems-1)
      currentPos = 0;
    else
      currentPos++;
    return next.copy();
  }
}
```

As mentioned above, the switch from supporting lists of strings to lists of `Listable` objects means that we now can implement and use lists of composite elements. This affects how we can compare elements, and what it means for two elements to be "equal." Consider the following `ListCircle` objects `C1`, `C2`, `C3`, and `C4`:

	C1	C2	C3	C4
xvalue	3	3	6	3
yvalue	4	4	12	4
radius	10	6	10	3
solid	true	false	false	true

Are any of the circles equal to each other? No, not in the strict sense of the word "equal." But what about equality as defined by the `compareTo` method of the `ListCircle` class? There, `ListCircle` objects are compared strictly on the basis of their radii. Based on that definition of equality, circles `C1` and `C3` are "equal." Although it might seem strange, this definition of equality could make perfect sense for a particular application, where the only important criteria for comparing circles is their size.

Remember that we are following the convention that our lists consist of unique objects. Are all of the circles in the table above unique? No, not in the "world" defined by the `ListCircle` class, where two circles are considered identical if they have the same `radius`. The `compareTo` method essentially defines the key for the list. In this case, the key is the radius. We should not insert both `C1` and `C3` on the same list. That would violate the precondition of the `insert` operation. Again, this seems like a strange restriction but might make sense within a particular application. (Please remember that the approach used here is not the only approach possible. For example, list ADTs could be developed that separate the concepts of key values and sort values.)

Let's look at another example. Earlier in this chapter we discussed different ways we might wish to sort a list of student records, with each record containing fields for first name, last name, identification number, and three test scores. For example, we could sort the list by name, or we could sort the list by identification number. Here is a table of values for student objects `S1`, `S2`, and `S3`.

	S1	S2	S3
first	Jones	Jones	Adams
last	David	Mary	Mark
IDnum	1234567	7654321	1111111
test1	89	92	100
test2	92	95	99
test3	95	89	100

In our approach, the field or fields that we use as a sorting criteria is the key for the list. If we were to define a `ListStudent` class—a class that allows us to maintain a list of students—then the definition of the `compareTo` operation in the `ListStudent` class would effectively define the key for the list elements. What would be the best choice for the key for a list of students? If we decide the last name is the key, then we are not able to hold both `S1` and `S2` on our list. That does not seem reasonable. Perhaps we could define the key to use the first name field as a tiebreaker when two last names are identical. That is better, but we could have two students who have identical first and last names, in which case we would be in trouble again. For this information, assuming a unique identification number has been assigned to each student, the `IDnum` field would be the best key. Therefore, the `compareTo` method of the `ListStudent` class should base its processing on a comparison of `IDnum` values.

Now it is clear why we changed the comment describing the effects of the `isThere` and `delete` operations. For example, when we were just using a list of strings the effect of `isThere` was "returns true if item is on the list ...." Now the effect is "returns true if an element with the same key as item is on this list ...." When dealing with lists of noncomposite elements, like strings, the entire element was in effect the key. That is no longer the case.

This brings us to the new list operation introduced in this section, the `retrieve` operation. Its definition in the abstract `List` class is

```
public abstract Listable retrieve(Listable item);
// Returns a copy of the list element with the same key as item
```

The application passes `retrieve` a `Listable` object and `retrieve` searches the list to find the element on the list that is "equal" (i.e., has the same key) to it. A copy of this element is returned. The specification of `retrieve` is as follows.

Listable retrieve (Listable item)

Effect:	Returns a copy of the list element with the same key as `item`.
Preconditions:	An element with a key that matches `item`'s key is on this list.
Postcondition:	Return value = (copy of list element that matches `item`)

Therefore, we can store information on a list and retrieve it later based on the item's key. For example, to retrieve student information about a student with an `IDnum` of 7654321, we instantiate a `ListStudent` object with dummy information for all of the fields except the `IDnum` field, which we initialize to 7654321. Then we pass this object to the `retrieve` operation, which returns a copy of the matching list element. This copy contains all the valid information about the student.

A Generic Sorted List ADT

Next we list the code for the generic sorted list class that completes the definition of the list class for the case of sorted lists. You can see that we use the binary search algorithm to implement the `isThere` and `retrieve` operations (although with the `retrieve` operation we do not need the `moreToSearch` variable because we know the item being retrieved is on the list). Note the use of the term `Listable` throughout the class, the use of the `copy` method invocation in the `insert` and `retrieve` methods, and several uses of the `compareTo` method. We call our new sorted list class `SortedList`.

```
//---------------------------------------------------------------------------
// SortedList.java              by Dale/Joyce/Weems                  Chapter 3
//
// Completes the definition of the List class under the assumption
// that the list is kept sorted
//---------------------------------------------------------------------------

package ch03.genericLists;

public class SortedList extends List
{
  public SortedList(int maxItems)
  // Instantiates and returns a reference to an empty list object
  // with room for maxItems elements
  {
    super(maxItems);
  }
```

```
public SortedList()
// Instantiates and returns a reference to an empty list object
// with room for 100 elements
{
  super(100);
}

public boolean isThere (Listable item)
// Returns true if an element with the same key as item is on this list;
// otherwise, returns false
{
  int compareResult;
  int midPoint;
  int first = 0;
  int last = numItems - 1;
  boolean moreToSearch = (first <= last);
  boolean found = false;

  while (moreToSearch && !found)
  {
    midPoint = (first + last) / 2;
    compareResult = item.compareTo(list[midPoint]);

    if (compareResult == 0)
      found = true;
    else if (compareResult < 0)  // item is less than element at location
    {
      last = midPoint - 1;
      moreToSearch = (first <= last);
    }
    else                         // item is greater than element at location
    {
      first = midPoint + 1;
      moreToSearch = (first <= last);
    }
  }

 return found;
}

public Listable retrieve (Listable item)
// Returns a copy of the list element with the same key as item
{
  int compareResult;
```

```
  int first = 0;
  int last = numItems - 1;
  int midPoint = (first + last) / 2;
  boolean found = false;

  while (!found)
  {
    midPoint = (first + last) / 2;
    compareResult = item.compareTo(list[midPoint]);

    if (compareResult == 0)
      found = true;
    else if (compareResult < 0)  // item is less than element at location
      last = midPoint - 1;
    else                         // item is greater than element at location
      first = midPoint + 1;
  }

 return list[midPoint].copy();
}



public void insert (Listable item)
// Adds a copy of item to this list
{
  int location = 0;
  boolean moreToSearch = (location < numItems);

  while (moreToSearch)
  {
    if (item.compareTo(list[location]) < 0)  // item is less
      moreToSearch = false;
    else                                     // item is more
    {
      location++;
      moreToSearch = (location < numItems);
    }
  }

  for (int index = numItems; index > location; index--)
    list[index] = list[index - 1];

  list[location] = item.copy();
  numItems++;
}
```

```
  public void delete (Listable item)
  // Deletes the element that matches item from this list
  {
    int location = 0;

    while (item.compareTo(list[location]) != 0)
      location++;

    for (int index = location + 1; index < numItems; index++)
      list[index - 1] = list[index];

    numItems--;
  }
}
```

The UML diagrams for the `List` and `SortedList` classes, plus the `Listable` interface, are displayed in Figure 3.14. Note the use of a dashed arrow labeled "uses," with an open arrowhead, to indicate the dependency of `List` and `Listable`. Although we

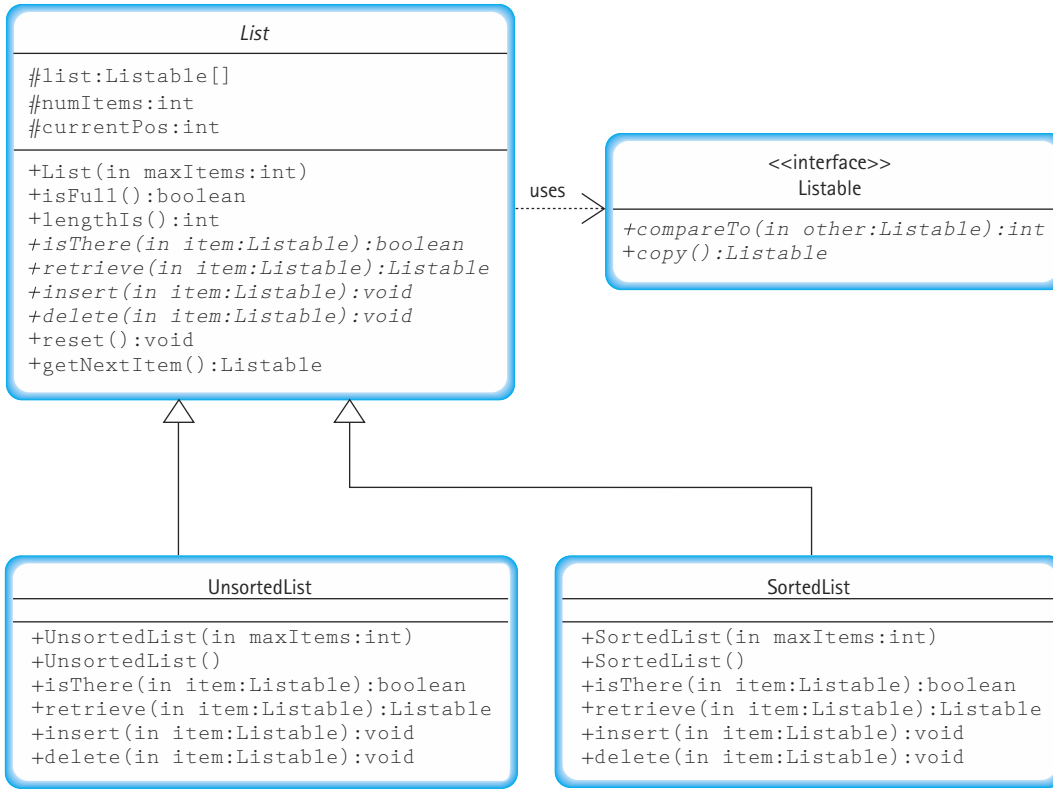


Figure 3.14 *UML diagrams for our list framework*

did not develop it, the figure also shows an `UnsortedList` class that extends `List`. This helps remind us that more than one class can extend the abstract list class. The implementation of the `UnsortedList` class is left as an exercise.

A Listable Class

Now that we have defined a generic list, a sorted list of `Listable` elements, we have to define a class that implements the `Listable` interface so that we have something to put on our lists. To keep our example straightforward, we continue to work with a list of strings. (In the case study of the next section, we provide a more complicated example of a class that implements `Listable`.)

We used lists of strings in the early part of this chapter so that we could introduce the reader gently to the topic of defining and implementing ADTs in Java. Knowing what we know now, about how to use interfaces to create generic lists, we would not have created a specific list implementation for lists of strings. Instead, we would use our generic list. But how do we use our generic list to provide a list of strings? We need to create a new class that hides a string variable and implements the `Listable` interface. We call this class `ListString`, since it provides strings that can be placed on our generic list.

Study the code for `ListString` below. Note that it contains a single object variable `key` that holds a string. It provides a constructor, plus the two methods needed to implement the `Listable` interface, the `copy` and `compareTo` methods. It also contains one other method, a `toString` method, which makes it easy for the application programmer to use objects of the class `ListString` as strings. When a class implements an interface, it must provide concrete methods for the abstract methods defined in the interface. As you can see, it can also include definitions for other methods.

```
package ch03.genericLists;

public class ListString implements Listable
{
  private String key;

  public ListString(String inString)
  {
    key = new String(inString);
  }

  public Listable copy()
  {
    ListString result = new ListString(this.key);
    return result;
  }
```

```
  public int compareTo(Listable otherListString)
  {
   ListString other = (ListString)otherListString;
   return this.key.compareTo(other.key);
  }

  public String toString()
  {
    return(key);
  }
}
```

Since `ListString` implements `Listable`, objects of class `ListString` can be used anywhere a `Listable` object is expected. Therefore, an object of class `ListString` can be passed to the `insert` method of the `SortedList` class. Furthermore, the same object can be placed on the hidden array within the `SortedList` class. And so on. We can use `ListString` objects with the `SortedList` class to provide a sorted list of strings.

If we wished to have a list of something else we would need to create another class that implements `Listable`. For example, if we wished to have a list of `Circle` objects we could complete our definition of the class `ListCircle`. This class would also implement `Listable`; therefore, it would contain its own versions of the `copy` and `compareTo` methods. What does it mean to compare circles? That depends on the intended use of the list of circles. Perhaps the comparison would be based on the size of the circles or on their positions. The `ListCircle` class requires a constructor; but would it require a `toString` method? Would it require any other methods? Again, the answers depend on the intended use of the list of circles. Being able to reuse our generic list ADT with list elements that have been defined for a specific application provides us with a powerful programming tool.

Using the Generic List

To create a sorted list of strings in an application program you simply instantiate an object of the class `SortedList`, using either of its constructors:

```
SortedList list1 = new SortedList();
SortedList list2 = new SortedList(size);
```

You also need to declare at least one object of class `ListString`, so that you have a variable to use as a parameter with the various `SortedList` methods:

```
ListString aString;
```

Once these declarations have been made you can instantiate `ListString` objects and place them on the list. For example, to place the string "`Amy`" on the list you might code:

```
aString = new ListString("Amy");
list.insert(aString);
```

We are not going to list an entire application program that uses a sorted list of strings. We did create a test driver (a form of application) that you can study and use; it is in the `TDSortedList.java` file of the `ch03` subdirectory of the `bookFiles` directory on our web site. Notice that it is not part of the `genericLists` package—instead it uses the package. So that the package classes are available to the test driver, it includes the following import statement:

```
import ch03.genericLists.*;
```

As long as the `bookFiles` directory is included on your computer's `ClassPath`, the compiler will know where to find the generic list files.

In the test driver you find uses of each of the sorted list methods with a `Listable` object:

```
outFile.println("The list is full is " + list.isFull());
outFile.println("Length of the list is " + list.lengthIs());
outFile.println(aString + " is on the list: " +
list.isThere(aString));
bString = (ListString)list.retrieve(aString);
list.insert(aString);
list.delete(aString);
list.reset();
aString = (ListString)list.getNextItem();
```

`SortedList.java` should be thoroughly tested. This job is left as an exercise.

The case study presented next shows another example of using the sorted list ADT. In the case study a list of real estate information is manipulated.

Case Study

Real Estate Listings

Problem Write a `RealEstate` program to keep track of a real estate company's residential listings. The program needs to input and keep track of all the listing information, which is currently stored on 3 × 5 cards in a box in their office.

The real estate salespeople must be able to perform a number of tasks using this data: add or delete a house listing, view the information about a particular house given the lot number, and look through a sequence of house information sorted by lot number.

We use the same design approach we described in Chapter 1 for this problem.

Write a program to keep track of a real estate company's residential listings. The program needs to input and keep track of all the listing information, which is currently stored on 3 x 5 cards in a box in their office.

The real estate salespeople must be able to perform a number of tasks using this data: add or delete a house listing, view the information about a particular house given the lot number, and look through a sequence of house information sorted by lot number.

Figure 3.15 *Problem statement with nouns circled and verbs underlined*

Brainstorming We said that nouns in the problem statement represent objects and that verbs describe actions. Let's approach this problem by analyzing the problem statement in terms of nouns and verbs. Let's circle nouns and underline verbs. The relevant nouns in the first paragraph are listings, information, cards, box, and office: circle them. The verbs that describe possible program actions are keep track, input, and stored: underline them. In the second paragraph, the nouns are salespeople, data, listing, information, house, lot number, and sequence: circle them. Possible action verbs are perform, add, delete, view, look through, and sorted: underline them. Figure 3.15 shows the problem statement with the nouns circled and the verbs underlined.

We did not circle program or underline write because these are instructions to the programmer and not part of the problem to be solved. Now, let's examine these nouns and verbs and see what insights they give us into the solution of this problem.

Filtering The first paragraph describes the current system. The objects are cards that contain information. These cards are stored in a box. Therefore, there are two objects in the office that we are going to have to simulate: 3 $\times$ 5 cards and a box to put them in. In the second paragraph, we discover several synonyms for the cards: data, listing, information, and house. We model these with the same objects that represent the cards. We also see what processing must be done with the cards and the box in which they are stored. The noun salespeople represents the outside world interacting with the program, so the rest of the paragraph describes the processing options that must be provided to the user of the program. In terms of the box of cards, the user must be able to add a new card, delete a card, view the information on the card given the lot number, and view a sequence of card information, sorted by lot number.

We can represent the cards by a class whose data members are the information written on the 3 $\times$ 5 cards. How do we represent the box of cards? We have just written several versions of the Abstract Data Type List. A list is a good candidate to simulate a box and the information on the list can be objects that represent the 3 $\times$ 5 cards. Since these objects represent the house information, and they should be kept on a list, let's call the class that models a card of house information `ListHouse`. We must make sure that our `ListHouse`

class implements the `Listable` interface, since we wish to maintain a list of `ListHouse` objects.

We now know that our program uses a list of `ListHouse` objects. But which version of a list shall we use? The unsorted version or the sorted version? Because the user must be allowed to look through the "house information sorted by lot number," the sorted version is a better choice. In the `ListHouse` class we base the definition of the `compareTo` method on the house's lot number. This ensures that the houses are kept sorted by lot number, just the way we need them.

So far, we have ignored the noun office and the fact that the program should "input and keep track" of the cards. A box of cards is stored permanently in the office. A list is a structure that exists only as long as the program in which it is defined is running. But how do we keep track of the information between runs of the program? That is, how do we simulate the office in which the box resides? A file is the structure that is used for permanent storage of information. Hence, there are two representations of the box of cards. When the program is running, the box is represented as a list. When the program is not running, the box is represented as a file. The program must move the information on the cards from a file to a list as the first action, and from a list to a file as the last action. We relegate the responsibility of interacting with the file to a class called `HouseFile`.

The `HouseFile` class hides the file of house information from the rest of the program. In this way, if the format of the file needs to be changed at a later time, the only part of the system that is affected is the `HouseFile` class. Limiting the scope of potential future changes is one of the main reasons we partition our systems into separate classes.

Let's capture the decisions we have made so far on CRC cards. On each card we record the main purpose of the class it represents, along with an initial set of responsibilities. Our cards show that our classes are already fairly well defined. The following table captures the information we record on the cards at this point (we display the final version of the cards after we finish the analysis section):

Class	Purpose	Responsibilities
`RealEstate`	Main program	Driver program, uses all the other classes to solve the problem; provides graphical user interface; implements actions represented by the interface buttons
`ListHouse`	Hold the information about a specific house.	Know all of its information; implement `Listable`; therefore, provide `copy` and `compareTo` methods
`SortedList`	Maintain a list of *ListHouse* elements.	See the List ADT specification.
`HouseFile`	Manage the file of house information.	Get house information from the file; save house information to the file.

User Interface Let's assume that the information on the 3 × 5 cards includes the owner's first and last names, the lot number, price, number of square feet, and number of bedrooms. The lot numbers are unique and therefore can be used as the key of the list. If an agent attempts to add a listing that duplicates an existing lot number, an error message is printed to the screen.

A review of the problem statement reveals that interaction with the user can take place one "house" at a time. Therefore, we design our graphical interface to display information about a house, and provide the user with buttons to initiate options related to that house (add, delete, clear) or to the overall system (reset, next, find). A count of the number of data fields, labels, and buttons needed, aided by some rough drafts drawn on scrap paper, leads us to a 9 × 2 grid layout for our interface. A sketch of our design is:

Lot Number:	45678
First Name:	John
Second Name:	Jones
Price:	96000
Square Feet:	1200
Number of Bedrooms:	3
Reset	Next
Add	Delete
Clear	Find

The user continues to manipulate the list of houses until he or she exits by closing the window.

Input Notice that there are three kinds of input: the file of houses saved from the last run of the program, the commands, and the data entered from the keyboard into the text fields in conjunction with the commands.

Output There are two kinds of output: the file of houses saved at the end of the run of the program, and screen output directed by one or more of the commands.

Data Objects There are house objects, represented in the program as `ListHouse` class objects. There are two container objects: the file of house objects retained from one run of the program to the next and the list into which the house objects are stored when the program is running (we call this object `list`). The collection of house listings is called our database.

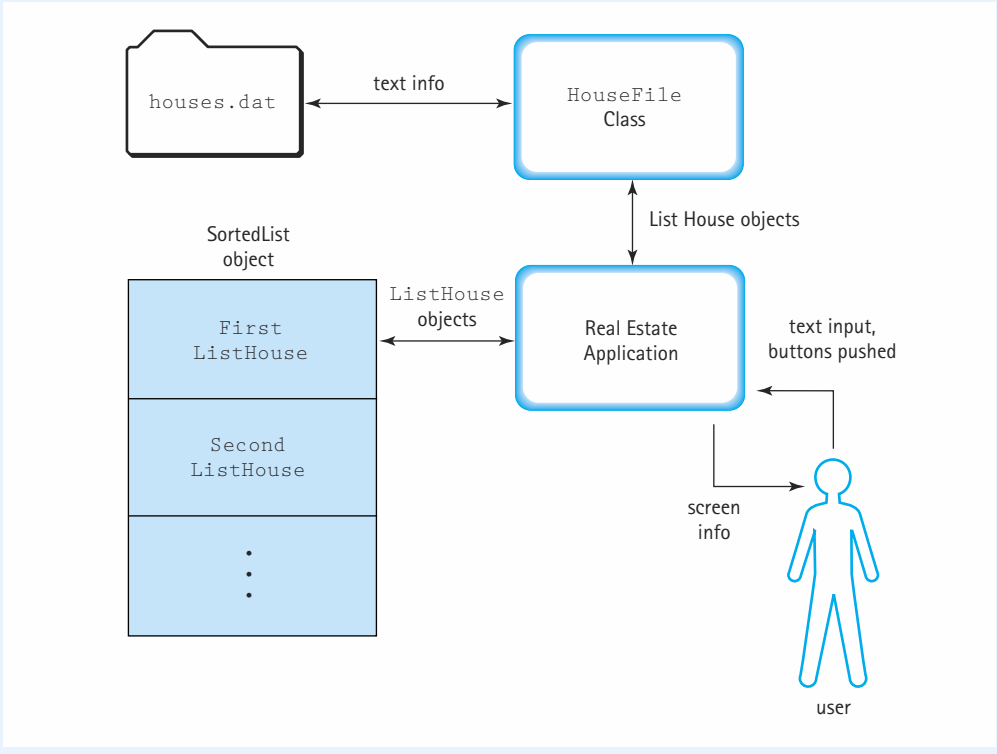


Figure 3.16 *Data flow of case study*

We name the physical file in which we retain the house objects `houses.dat`.

The diagrams in Figures 3.16 and 3.17 show the general flow of the processing and what the data objects look like. Note that we know the internal workings of the List ADT because we have just written the code earlier in the chapter. When we wrote that we were acting as the ADT programmer, creating a tool for use by application programmers. Now however, we are changing hats; we are acting as the application programmer. We write the program only using the interface as represented in the List ADT specification.

Scenario Analysis Where do we go from here? Scenario analysis lets us "test" our design. Using our CRC cards we can walk through several scenarios that represent the typical expected use of the system. This allows us to refine the responsibilities of our identified classes and begin to add detailed information about method names and interfaces. During the course of this analysis we may uncover holes in our identified classes or user interface.

We begin by working through a scenario in which a real estate salesperson runs the program and tries to get information about the house on lot number 45678. We realize that the first thing the system must do is to build the internal list of houses from the house information contained in the file. We need to decide which class should have this responsibility. We could

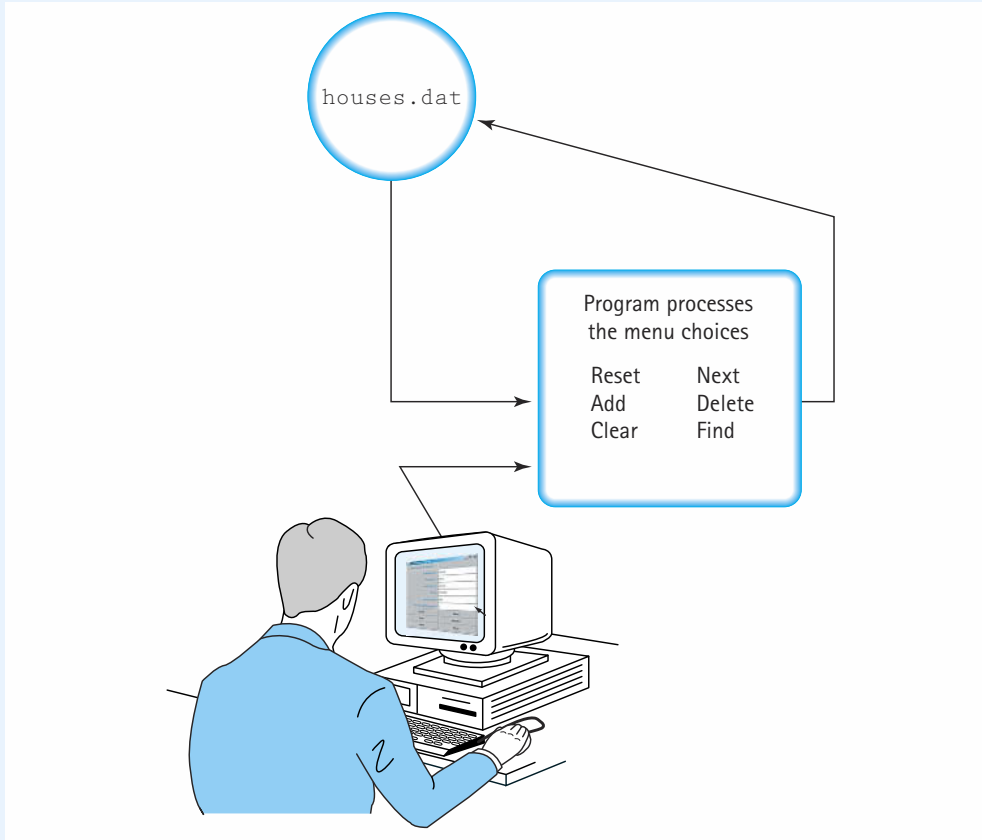


Figure 3.17 *The high-level processing of the case study*

assign this task to the `HouseFile` class, since it is able to get the house information from the file. However, we decide that such a task is outside its main purpose, which is to manage the file of house information. Therefore, we decide that the `RealEstate` class should perform this task. We add a notation to this effect to the list of responsibilities on its CRC card and move ahead.

Now we must decide how the `RealEstate` class gets the house information from the `HouseFile` class. Should the information be sent one field at a time or one house at a time? We decide to use the latter approach, since we have already defined a class that encapsulates house data, namely the `ListHouse` class. The `HouseFile` class can provide information to the `RealEstate` class in the form of `ListHouse` objects. And vice versa at the end of the program's execution, the `HouseFile` class can receive information from the `RealEstate` class in the form of `ListHouse` objects.

As our scenario continues we imagine the `RealEstate` class requesting house information from the `HouseFile` class. First it informs the `HouseFile` class that it wishes to begin reading house data. A standard name for this method is `reset`. Next, as long as there is more house data available, it asks the `HouseFile` class for data about another house. Therefore, `HouseFile` must provide both a `moreHouses` method that returns a `boolean`, and a `getNextHouse` method that returns an object of type `ListHouse`. A similar analysis of how the data can be saved to the file at the end of the program run leads to the identification of a `rewrite` method and a `putToFile` method. We also need a method to inform the `HouseFile` class that we are finished with the file and it should be closed. Finally, we decide to use our standard approach for reading from a file, so we note that `HouseFile` must collaborate with Java's `BufferedReader`, `FileReader`, `PrintWriter`, and `FileWriter` classes. We update the CRC card for `HouseFile`, and move ahead.

As the scenario unfolds we find that most of the operations needed for normal processing have already been assigned to one of our classes. The user clicks on the Clear button and the `RealEstate` class clears the information from the text fields; the user enters the lot number 45678 into the Lot Number text field and clicks on the Find button; the `RealEstate` class creates a `ListHouse` object with 45678 as its lot number, uses the list `isThere` operation to see if the house is on the list; if it is on the list then the list `retrieve` operation is used to obtain all of the house information, which is subsequently displayed in the text fields.

But what if the house is not found on the list? When doing scenario analysis it is important to consider all the variations of the scenario. In this case, the program should report to the user that the house was not found. How does it do that? We have uncovered a hole in our interface design. We need to include a way to communicate the results of operations to the user. So, we go back and rework our draft of the interface to include a status box in the upper left corner of the window. We decide we can use this status box to display a message in response to each option selected by the user. For example, if the user selects the Add button and the house is successfully added to the list, we display the message "House added to list." Now our interface is a 10×2 grid.

The investigation of other scenarios is left to the reader. The final set of CRC cards, created strictly for this application, is shown below. We do not include a card for the List ADT since that was not created for this application. Also, we do not include a card for the `RealEstate` main program, since that is the application that uses the classes represented by the other cards.

Class Name: ListHouse	Superclass: Object	Subclasses:
Primary Responsibility: Provide a house object to use with a list		

Responsibilities	Collaborations
Create itself (lastName, firstName, lotNumber, price, squareFeet, bedrooms)	None
Copy itself	None
return Listable	
Compare itself to another ListHouse (other ListHouse)	None
return int	
Know its information:	None
Know lastName, return String	
Know firstName, return String	
Know lotNumber, return int	
Know price, return int	
Know squareFeet, return int	
Know bedrooms, return int	

Class Name: HouseFile	Superclass: Object	Subclasses:
Primary Responsibility: Manage the file of house information		
Responsibilities	**Collaborations**	
Set up for reading	BufferedReader, FileReader	
know if there are more houses to read	None	
return boolean		
Get the info about the next house from the file	BufferedReader, Integer	
return ListHouse		
Set up for writing	PrintWriter, FileWriter	
Put info about a house to the file (house)	PrintWriter	
Close the file	BufferedReader, PrintWriter	

We now turn our attention to the design, implementation, and testing of the identified classes. Note that the `SortedList` class that we use has already been created and tested, so we can assume that it works properly. This is a prime benefit of creating ADTs—once created and tested they can be used with confidence in other systems. We create a package, `ch03.houses`, to hold the `ListHouse` and `HouseFile` classes. We use the package to hold the "helper" classes only; therefore, we do not include the `RealEstate` class, which is an application, as part of the package. You can find the `RealEstate.java` file in the `ch03` subdirectory of the `bookFiles` directory and the `ListHouse.java` and `HouseFile.java` files in the `houses` subdirectory of the `ch03` subdirectory. The files are available on our web site.

The ListHouse Class `ListHouse` must encapsulate house information and it must implement the `Listable` interface, since `ListHouse` objects are placed on a list. Its implementation is rather straightforward. It follows the same patterns established in the `ListCircle` and `ListString` classes developed earlier in the chapter. We must implement `compareTo` and `copy`, but we must also declare variables for all of the information about the house. That is, we must have instance variables for the last name, the first name, the lot

number, the price, the number of square feet, and the number of bedrooms. We also need to have observer operations for each of these variables.

```
//--------------------------------------------------------------------------
// ListHouse.java             by Dale/Joyce/Weems                  Chapter 3
//
// Provides elements for a list of house information
//--------------------------------------------------------------------------

package ch03.houses;

import ch03.genericLists.*;

public class ListHouse implements Listable
{
  // House information
  private String lastName;
  private String firstName;
  private int lotNumber;
  private int price;
  private int squareFeet;
  private int bedRooms;

  public ListHouse(String lastName, String firstName, int lotNumber,
                   int price, int squareFeet, int bedRooms )
  {
    this.lastName   = lastName;
    this.firstName  = firstName;
    this.lotNumber  = lotNumber;
    this.price      = price;
    this.squareFeet = squareFeet;
    this.bedRooms   = bedRooms;
  }

  public Listable copy()
  // Returns a copy of this ListHouse
  {
    ListHouse result = new ListHouse(lastName, firstName, lotNumber, price,
                                     squareFeet, bedRooms);
    return result;
  }

  public int compareTo(Listable otherListHouse)
  // Houses are compared  based on their lot numbers
```

```
    {
     ListHouse other = (ListHouse)otherListHouse;
     return (this.lotNumber - other.lotNumber);
    }

    // Observers
    public String lastName()
    {
      return lastName;
    }

    public String firstName()
    {
      return firstName;
    }

    public int lotNumber()
    {
      return lotNumber;
    }

    public int price()
    {
      return price;
    }

    public int squareFeet()
    {
      return squareFeet;
    }

    public int bedRooms()
    {
      return bedRooms;
    }
}
```

We should test the `ListHouse` class by itself and integrated with the `SortedList` class. We can test it by itself by creating a `TDListHouse` program, similar to the other test driver programs we have used. Our test cases first invoke the constructor, followed by calls to each of the observer methods to ensure that they return the correct information. This could be followed by a test of the copy operation, using it to create a copy of the original `ListHouse` and then repeating the observer method tests on the new object. Finally, the `compareTo` operation must be

tested in a variety of situations: compare houses with lot numbers that are less than, equal to, or greater than each other, compare a house to itself, compare a house to a copy of itself, and so on.

`ListHouse` can be tested with the `SortedList` class by repeating the sequence of tests previously used to test our sorted list of strings, replacing the strings with house information.

The HouseFile Class

This class manages the `houses.dat` file. When requested, it pulls data from the file, encapsulates the data into `ListHouse` objects, and returns the `ListHouse` objects to its client. Additionally, it takes `ListHouse` objects from its client and saves the information to the `houses.dat` file.

There is no need to create numerous `HouseFile` objects. Since the class always deals with the same file, we would not want to have several instances of the class interacting with the file at the same time. If we allowed that the file could become corrupted and the system could crash (for example if one instance of the class was trying to read from the file while another instance of the class was trying to write to the file). Therefore, we do not support objects of the class `HouseFile`. We code all of its methods as `static` methods. Recall that this means that the methods are invoked directly through the class itself, as opposed to being invoked through an object of the class. We also declare all of its variables to be `static` variables, that is, class variables as opposed to object variables.

A study of the CRC card for `HouseFile` combined with the analysis of the previous paragraph leads to the following abstract specification of the `HouseFile` class:

House File Specification

Structure:

The house information is kept in a text file called `houses.dat`. For each house the following information is kept, in the order listed, one piece of information per line: last name (`String`), first name (`String`), lot number (`int`), price (`int`), square feet (`int`), and number of bedrooms (`int`).

Operations:

static void reset

Effect:	Resets the file for reading
Throws:	`IOException`

static void rewrite

Effect:	Resets the file for writing
Throws:	`IOException`

static boolean moreHouses

Effect:	Determines whether there is still more house information to be read
Postcondition:	Return value = (there is more house information)

static ListHouse getNextHouse

Effect:	Reads the next house information from the file
Postcondition:	Return value = (a `ListHouse` object containing the next house information)
Throws:	`IOException`

static void putToFile (ListHouse house)

Effect:	Writes the house information to the file
Throws:	`IOException`

static void close

Effect:	Closes the file
Throws:	`IOException`

Reading information from a file and writing information to a file was used in the `TDIncDate` program at the end of Chapter 1. The Java Input/Output feature section that accompanied that program addresses the Java code used to provide those operations. That program only needs to read and write information of type `int`. The `HouseFile` class also performs input and output of `String` information. This is straightforward, since the methods provided by the `java.io` class directly support strings:

```
firstName = inFile.readLine();        // Input of String
outFile.printLn(house.firstName());   // Output of String
```

The `HouseFile` class must keep track of whether or not the `houses.dat` file is closed or opened, and if open, whether it is open for reading or open for writing. It must not allow reading from the file when it is open for writing; nor writing to the file when it is open for reading; nor reading or writing if the file is closed. The `boolean` class variables `inFileOpen` and `outFileOpen` are used to keep track of the status of the file.

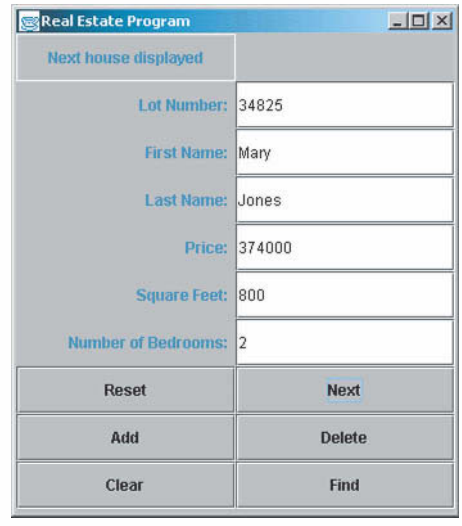
Here is the implementation:

```
//---------------------------------------------------------------------------
// HouseFile.java              by Dale/Joyce/Weems                  Chapter 3
//
// Manages file "houses.dat" of real estate information
//---------------------------------------------------------------------------

package ch03.houses;

import java.io.*;

public class HouseFile
// Manages file "houses.dat" of real estate information
```

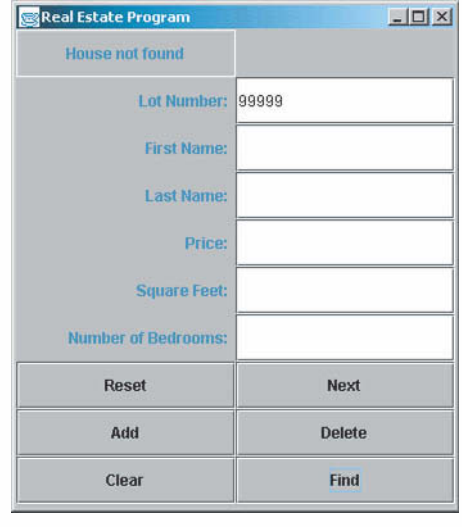
```
{
  private static BufferedReader inFile;
  private static PrintWriter outFile;
  private static boolean inFileOpen = false;
  private static boolean outFileOpen = false;
  private static String inString ="";          // Holds "next" line from file
                                               // Equals null if at end of file

  public static void reset() throws IOException
  // Reset file for reading
  {
    if (inFileOpen) inFile.close();
    if (outFileOpen) outFile.close();
    inFile = new BufferedReader(new FileReader("houses.dat"));
    inFileOpen = true;
    inString = inFile.readLine();
  }

  public static void rewrite() throws IOException
  // Reset file for writing
  {
    if (inFileOpen) inFile.close();
    if (outFileOpen) outFile.close();
    outFile = new PrintWriter(new FileWriter("houses.dat"));
    outFileOpen = true;
  }

  public static boolean moreHouses()
  // Returns true if file open for reading and there is still more house
  // information available in it
  {
    if (!inFileOpen || (inString == null))
      return false;
    else return true;
  }

  public static ListHouse getNextHouse() throws IOException
  // Gets and returns house information from the house info file
  // Precondition: inFile is open and holds more house information
  {
    String lastName = "xxxxx";
    String firstName = "xxxxx";
    int lotNumber = 0;
    int price = 0;
    int squareFeet = 0;
    int bedRooms =0;
```

```
    lastName = inString;
    firstName = inFile.readLine();
    lotNumber = Integer.parseInt(inFile.readLine());
    price = Integer.parseInt(inFile.readLine());
    squareFeet = Integer.parseInt(inFile.readLine());
    bedRooms = Integer.parseInt(inFile.readLine());

    inString = inFile.readLine();

    ListHouse house = new ListHouse(lastName, firstName, lotNumber, price,
                                    squareFeet, bedRooms);
    return house;
  }

  public static void putToFile(ListHouse house) throws IOException
  // Puts parameter house information into the house info file
  // Precondition: outFile is open
  {
    outFile.println(house.lastName());
    outFile.println(house.firstName());
    outFile.println(house.lotNumber());
    outFile.println(house.price());
    outFile.println(house.squareFeet());
    outFile.println(house.bedRooms());
  }

  public static void close() throws IOException
  // Closes house info file
  {
    if (inFileOpen) inFile.close();
    if (outFileOpen) outFile.close();
    inFileOpen = false;
    outFileOpen = false;
  }
}
```

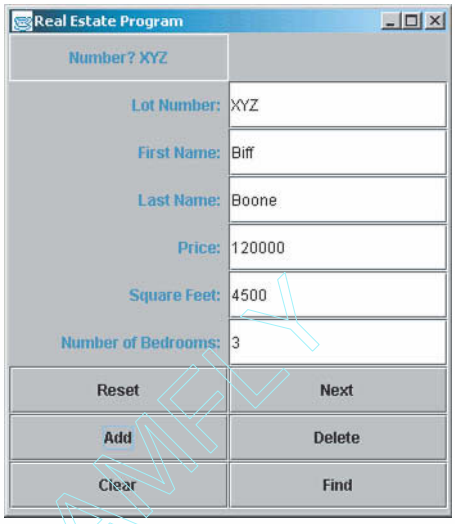
RealEstate Program We now look at the main program, the program that uses all of the other classes to solve the problem. The main program includes the user interface code, in fact that code makes up the majority of the program. Any input/output mechanisms used here that have not yet been encountered in this text are addressed in the feature section, Java Input/Output II that follows the case study. Here is a screen shot of the running program after the user has selected the option to display the "next" house:



Here is how the program reacts to an attempt to "find" a house that is not on the list:



This example shows what happens if the user tries to "add" a house with poorly formatted information:



The algorithm for the main program is as follows:

```
Get the house information from the HouseFile object and build the list of houses.
Present the initial frame
As long as the frame remains open
  Listen for and respond to user choices
    Reset – reset the list and display the first house from the list
    Next – display the next house from the list
    Add – if it is not already on the list, add the currently displayed house to the list
    Delete – if it is on the list, remove the house that matches the currently displayed
             lot number from the list
    Clear – clear the text fields
    Find – display the house from the list that matches the currently displayed lot
           number, if possible
Send the information about the houses from the list to the HouseFile object
```

Processing begins by using the `HouseFile` class to obtain the house information from the file, and using the `SortList` class to store the house information. This is accomplished through the following steps:

```
Create a new list
Reset the house file for reading
while there are still more houses to read
  Read the next house and
  Insert it into the list
```

The code that corresponds to this algorithm can be found in the main method, after the various interface labels, text fields, and buttons have been set up, and the display frame has been initialized.

So, that's how we get the information from the file onto the list at the beginning of our processing. How do we reverse this process? That is, how do we take the information from the list and save it back to the file? Actually, the save algorithm is very similar:

```
Reset the house file for writing
Rest the list
for each house on the list
  Get the house from the list and
  Store it in the file
```

Where should the code for this algorithm go in the program? We want this code to be one of the last things that the program does. Remember, this is an event-driven program, so we cannot just put the code at the end of the `main` method and expect it to be executed last. Instead, we define our own `WindowClosing` event handler and place the corresponding code there. In this way, when the user is finished and closes the application window, the information is saved.

Now that we have determined how we get the house information from the file to the list, and vice versa, the only processing that remains is what occurs in response to the user pressing buttons in the interface. There are six buttons. The processing required by each button is fairly well stated in the original algorithm above. Many of them require moving information from the display (the set of text fields) to the list, or vice versa. This leads us to design three helper methods

- `showHouse` – accepts a `ListHouse` object as a parameter and displays the information about the object in the text fields
- `getHouse` – obtains the information from the textboxes, turns it into a `ListHouse` object, and returns the object
- `clearHouse` – clears the information from the textboxes

The implementation of these methods is straightforward, and with their help we can implement the button-processing routines without much difficulty. For example, the algorithm to handle the Reset button is:

```
Reset the list
if the list is empty
  clearHouse
else
  Set house to the first house on the list
  showHouse(house)
Report "List reset" through the status label
```

The code that corresponds to this algorithm is placed in the `ActionHandler` class, and associated with the "Reset" event.

Study the code for the other event handlers to see how they use the helper methods to perform their tasks. Note that each of the event handlers that depends upon the user entering information into a text field, use Java's exception handling mechanism to protect the application from user input errors. For example, the algorithm for the Add event is:

```
try
  Set house to getHouse
  if house is already on the list
    Report "Lot number already in use" through the status label
  else
    Insert house into the list
    Report "House added to list" through the status label
catch NumberFormat Exception
  Report a problem with the house data through the status label
```

Since four of the house fields require `int` data, if the system raises a `NumberFormatException` it is because something other than an integer was listed in at least one of those fields. Therefore, the message displayed through the status label is "Number?", followed by an echo of the bad data. The bad data value is available through the exception object's `getMessage` method. You can see that similar protection is provided for the Delete and Find event handlers in the code.

Here is the listing for the Real Estate application:

```
//--------------------------------------------------------------------------
// RealEstate.java             by Dale/Joyce/Weems                  Chapter 3
//
// Helps keep track of a company's real estate listings
//--------------------------------------------------------------------------

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
import java.io.*;
import ch03.houses.*;
import ch03.genericLists.*;

public class RealEstate
{
  // The list of house information
  private static SortedList list = new SortedList();

  // Text fields
  private static JTextField lotText;              // Lot number field
  private static JTextField firstText;            // First name field
```

```
private static JTextField lastText;             // Last name field
private static JTextField priceText;            // Price field
private static JTextField feetText;             // Square feet field
private static JTextField bedText;              // Number of bedrooms field

// Status Label
private static JLabel statusLabel;              // Label for status info

// Display information about parameter house on screen
private static void showHouse(ListHouse house)
{
  lotText.setText(Integer.toString(house.lotNumber()));
  firstText.setText(house.firstName());
  lastText.setText(house.lastName());
  priceText.setText(Integer.toString(house.price()));
  feetText.setText(Integer.toString(house.squareFeet()));
  bedText.setText(Integer.toString(house.bedRooms()));
}

// Returns current screen information as a ListHouse
private static ListHouse getHouse()
{
  String lastName;
  String firstName;
  int lotNumber;
  int price;
  int squareFeet;
  int bedRooms;

  lotNumber = Integer.parseInt(lotText.getText());
  firstName = firstText.getText();
  lastName = lastText.getText();
  price = Integer.parseInt(priceText.getText());
  squareFeet = Integer.parseInt(feetText.getText());
  bedRooms = Integer.parseInt(bedText.getText());

  ListHouse house = new ListHouse(lastName, firstName, lotNumber, price,
                                  squareFeet, bedRooms);
  return house;
}

// Clears house information from screen
private static void clearHouse()
{
  lotText.setText("");
  firstText.setText("");
```

```
    lastText.setText("");
    priceText.setText("");
    feetText.setText("");
    bedText.setText("");
  }

  // Define a button listener
  private static class ActionHandler implements ActionListener
  {
    public void actionPerformed(ActionEvent event)
    // Listener for the button events
    {
      ListHouse house;

      if (event.getActionCommand().equals("Reset"))
      { // Handles Reset event
        list.reset();
        if (list.lengthIs() == 0)
          clearHouse();
        else
        {
          house = (ListHouse)list.getNextItem();
          showHouse(house);
        }
        statusLabel.setText("List reset");
      }
      else
      if (event.getActionCommand().equals("Next"))
      { // Handles Next event
        if (list.lengthIs() == 0)
          statusLabel.setText("list is empty!");
        else
          {
          house = (ListHouse)list.getNextItem();
          showHouse(house);
          statusLabel.setText("Next house displayed");
        }
      }
      else
      if (event.getActionCommand().equals("Add"))
      { // Handles Add event
        try
        {
          house = getHouse();
```

```
      if (list.isThere(house))
        statusLabel.setText("Lot number already in use");
      else
      {
        list.insert(house);
        statusLabel.setText("House added to list");
      }
    }
    catch (NumberFormatException badHouseData)
    {
      // Text field info incorrectly formated
      statusLabel.setText("Number? " + badHouseData.getMessage());
    }
  }
  else
  if (event.getActionCommand().equals("Delete"))
  { // Handles Delete event
    try
    {
      house = getHouse();
      if (list.isThere(house))
      {
        list.delete(house);
        statusLabel.setText("House deleted");
      }
      else
        statusLabel.setText("Lot number not on list");
    }
    catch (NumberFormatException badHouseData)
    {
      // Text field info incorrectly formated
      statusLabel.setText("Number? " + badHouseData.getMessage());
    }
  }
  else
  if (event.getActionCommand().equals("Clear"))
  { // Handles Clear event
    clearHouse();
    statusLabel.setText(list.lengthIs() + " houses on list");
  }
  else
  if (event.getActionCommand().equals("Find"))
  { // Handles Find event
    int lotNumber;
    try
```

```
        {
          lotNumber = Integer.parseInt(lotText.getText());
          house = new ListHouse("", "", lotNumber, 0, 0, 0);
          if (list.isThere(house))
          {
            house = (ListHouse)list.retrieve(house);
            showHouse(house);
            statusLabel.setText("House found");
          }
          else
          statusLabel.setText("House not found");
        }
        catch (NumberFormatException badHouseData)
        {
          // Text field info incorrectly formated
          statusLabel.setText("Number? " + badHouseData.getMessage());
        }
      }
    }
  }

  public static void main(String args[]) throws IOException

  {
    ListHouse house;
    char command;
    int length;

    JLabel blankLabel;          // To use up one frame slot

    JLabel lotLabel;            // Labels for input fields
    JLabel firstLabel;
    JLabel lastLabel;
    JLabel priceLabel;
    JLabel feetLabel;
    JLabel bedLabel;

    JButton reset;              // Reset button
    JButton next;               // Next button
    JButton add;                // Add button
    JButton delete;             // Delete button
    JButton clear;              // Clear button
    JButton find;               // Find button
    ActionHandler action;       // Declare listener
```

```
// Declare/Instantiate/Initialize display frame
JFrame displayFrame = new JFrame();
displayFrame.setTitle("Real Estate Program");
displayFrame.setSize(350,400);
displayFrame.addWindowListener(new WindowAdapter()  // handle window
                                                    //   closing
{
  public void windowClosing(WindowEvent event)
  {
    ListHouse house;
    displayFrame.dispose();                    // Close window
    try
    {
      // Store info from list into house file
      HouseFile.rewrite();
      list.reset();
      int length = list.lengthIs();
      for (int counter = 1; counter <= length; counter++)
      {
        house = (ListHouse)list.getNextItem();
        HouseFile.putToFile(house);
      }
      HouseFile.close();
    }
    catch (IOException fileCloseProblem)
    {
      System.out.println("Exception raised concerning the house info file "
                         + "upon program termination");
    }
    System.exit(0);                            // Quit the program
  }
});

// Instantiate content pane and information panel
Container contentPane = displayFrame.getContentPane();
JPanel infoPanel = new JPanel();

// Instantiate/initialize labels, and text fields
statusLabel = new JLabel("", JLabel.CENTER);
statusLabel.setBorder(new LineBorder(Color.red));
blankLabel = new JLabel("");
lotLabel = new JLabel("Lot Number:  ", JLabel.RIGHT);
lotText = new JTextField("", 15);
firstLabel = new JLabel("First Name:  ", JLabel.RIGHT);
firstText = new JTextField("", 15);
```

```
lastLabel = new JLabel("Last Name:  ", JLabel.RIGHT);
lastText = new JTextField("", 15);
priceLabel = new JLabel("Price:  ", JLabel.RIGHT);
priceText = new JTextField("", 15);
feetLabel = new JLabel("Square Feet:  ", JLabel.RIGHT);
feetText = new JTextField("", 15);
bedLabel = new JLabel("Number of Bedrooms:  ", JLabel.RIGHT);
bedText = new JTextField("", 15);

// Instantiate/register buttons
reset = new JButton("Reset");
next = new JButton("Next");
add = new JButton("Add");
delete = new JButton("Delete");
clear = new JButton("Clear");
find = new JButton("Find");

// Instantiate/register button listeners
action = new ActionHandler();
reset.addActionListener(action);
next.addActionListener(action);
add.addActionListener(action);
delete.addActionListener(action);
clear.addActionListener(action);
find.addActionListener(action);

// Load info from house file into list
HouseFile.reset();
while (HouseFile.moreHouses())
{
  house = HouseFile.getNextHouse();
  list.insert(house);
}

// If possible insert info about first house into text fields
list.reset();
if (list.lengthIs() != 0)
{
  house = (ListHouse)list.getNextItem();
  showHouse(house);
}

// Update status
statusLabel.setText(list.lengthIs() + " houses on list                ");
```

```
        // Add components to frame
        infoPanel.setLayout(new GridLayout(10,2));
        infoPanel.add(statusLabel);
        infoPanel.add(blankLabel);
        infoPanel.add(lotLabel);
        infoPanel.add(lotText);
        infoPanel.add(firstLabel);
        infoPanel.add(firstText);
        infoPanel.add(lastLabel);
        infoPanel.add(lastText);
        infoPanel.add(priceLabel);
        infoPanel.add(priceText);
        infoPanel.add(feetLabel);
        infoPanel.add(feetText);
        infoPanel.add(bedLabel);
        infoPanel.add(bedText);
        infoPanel.add(reset);
        infoPanel.add(next);
        infoPanel.add(add);
        infoPanel.add(delete);
        infoPanel.add(clear);
        infoPanel.add(find);

        // Set up and show the frame
        contentPane.add(infoPanel);
        displayFrame.show();
    }
}
```

Test Plan We assume classes `Listable` and `SortedList` have been thoroughly tested. This leaves classes `ListHouse`, `HouseFile`, and the Real Estate application program to test. To test the two classes we could create test driver programs to call the various methods and display results. But recall that these classes were created specifically for the Real Estate application. Therefore, we can use the main application as the test driver to test them. In other words, we can test everything together.

The first task is to create a master file of houses by using the Add command to input several houses and quit. We then need to input a variety of commands to add more houses, delete houses, find houses, and look through the list of houses with the Reset and Next buttons. We should try the operations with good data and with bad data (for example, nonintegral lot numbers). We should try the operations in as many different sequences as we can devise. The program must be run several times in order to test the access and preservation of the data base (file `houses.dat`). We leave the final test plan as an exercise.

In the discussion of object-oriented design in Chapter 1, we said that the code responsible for coordinating the objects is called a driver. Now, we can see why. A driver program in testing terminology is a program whose role is to call various subprograms and observe their

behavior. In object-oriented terminology, a program is a collection of collaborating objects. Therefore, the role of the main application is to invoke operations on certain objects, that is, get them started collaborating, so the term driver is appropriate. In subsequent chapters, when we use the term driver, the meaning should be clear from the context.

Java Input/Output II

Let's look at the graphical user interface of the Real Estate program. This interface is more complicated than that used by the test driver program we saw in Chapter 1. The test driver displayed only labels, whereas the Real Estate program displays labels, text fields, and buttons. However, the biggest difference is in how the frame is used by the user of the program. The test driver program simply displayed a few lines of information on its frame, and then waited for the user to close the frame. The frame for the Real Estate program, on the other hand, is changed based on actions performed by the user. It is truly interactive.

Throughout the following discussion, please review the code from the Real Estate program that corresponds to the particular discussion topic.

The Frame

First let's look at how the frame is constructed. The setup of the frame is similar to that performed by the test driver program. However, handling window closing is more complicated here, since we must perform some special processing (save the information from the list to the `houses.dat` file) instead of just exiting the system. We name our frame `displayFrame` and set its title and size as we did before.

```
JFrame displayFrame = new JFrame();
displayFrame.setTitle("Real Estate Program");
displayFrame.setSize(350,400);
```

Next we define the needed reaction to the window-closing event with the following commands (see the main method, after a sequence of label and button declarations):

```
displayFrame.addWindowListener(new WindowAdapter()  // Handle window
                                                    //   closing
{
. . .
  });
```

The actual code executed when the window is closed is represented by the "..."; it saves the current list of house information to the data file, and then exits the program. Let's discuss the `addWindowListener` method. As you know, when the frame is displayed, it appears in its own window. Normally, when you define a window listener from within a Java program, you must

define how the window reacts to various events: closing the window, resizing the window, activating the window, and so on. You must define methods to handle all of these events. However, in our program we only want to handle one of these events, the window closing event. The code above lets us directly handle the window closing event while we accept default "do nothing" handlers for all the other window events. In effect, a `WindowAdapter` object is a window that has "do nothing" events defined for all window events. We are adding a "window closing listener" to our frame that tells the program what processing to perform when someone closes the window, overriding the default "do nothing" event handler in this case.

As was done for the test driver, we next instantiate the content pane, and an information panel:

```
Container contentPane = displayFrame.getContentPane();
JPanel infoPanel = new JPanel();
```

Recall that the content pane is the part of a frame that is used to display information generated by a program, and a panel is a container, capable of holding other constructs, where the program organizes its information for display.

Components

Next we create components that are eventually added to our panel. We create labels, text fields, and buttons. We look at each in turn, starting with labels. You are familiar with labels from the Chapter 1 test driver. In the Real Estate program, we exploit a little more of the functionality provided by the `JLabel` class. Consider the two statements that set up the status label—the label used in the interface to display a message describing the result of a user action:

```
statusLabel = new JLabel("", JLabel.CENTER);
statusLabel.setBorder(new LineBorder(Color.red));
```

In addition to passing the `JLabel` constructor an initial string, we pass it the constant `CENTER`, defined in the `JLabel` class. This sets the label so that it displays text centered in the area allocated to it. That property persists until we change it with a call to the label's `setHorizontalAlignment` method. We follow the instantiation of `statusLabel` with a message to it, to set its border to a line border with the color red. Borders can be set for any Java Swing component that extends the `JComponent` class; that is, for most Swing components. Swing supports eight kinds of borders—we have elected to use the line border in this case. Note that we pass the `LineBorder` constructor a constant of the `Color` class. Also note, that to use borders, we must import `javax.swing.border.*` into the program. Finally, note that most of the labels used by the program are declared at the beginning of the main method, but the `statusLabel` label is declared outside the main method, since it needs to be visible to some of the helper methods.

Intermingled with the label instantiations are instantiations of text fields. This is a new construct for us. A text field is a box that allows the user to enter a single line of text. In the Real Estate program, we use them to both gather and present information to the user. An example of a test field instantiation is:

```
lotText = new JTextField("", 15);
```

The string parameter sets the initial text in the text field (in this case to the empty string); the integer parameter sets the width of the text field. Since all of our text fields are accessed by helper methods, they are all declared outside of the main method.

The text displayed in a text field can be changed with the `setText` method, as is done in the `showHouse` helper method to display the information about a house on the interface. Of course, the user can directly enter text into a text field box and change its contents. The `getText` method is used by a program to obtain the current information in a text field. See the `getHouse` helper method for examples of its use.

The final construct used in our interface is the button. Buttons are used to generate events, when pressed by the user. Button definition is easy. Just invoke the button constructor, passing it the string to be displayed on the face of the button, as follows:

```
reset = new JButton("Reset");
```

Although button-related events are handled by helper methods, the buttons themselves are only used within the `main` method, so all button declarations are at the beginning of `main`. There are six buttons altogether. The Real Estate program "listens" for its user to press one of the buttons, performs processing related to the pressed button, updates the frame appropriately, and then resumes listening. To understand how this works, and how we implement it, we must look at the Java event model.

The Event Model

In an event-driven Java program, there are two important entities, event sources and event listeners. The sources generate an event, usually due to some action by the user. The listeners are waiting for certain events to occur, and when they do, they react to the event by performing some related processing. In our program, the `JButton` object `reset` is an event source, and the `ActionHandler` object `action` is an event listener. These objects are declared and instantiated by the statements:

```
JButton reset;
reset = new JButton("Reset");

ActionHandler action;
action = new ActionHandler();
```

We have already examined the `JButton` statements. But, what is an `ActionHandler`? You won't find it defined in any Java library documentation because it is a class created just for the Real Estate program. It is an inner class. You can find its definition in the program listing just after the helper methods that manage the house information displayed on the screen. It looks like this (with many lines deleted):

```
private static class ActionHandler implements ActionListener
  {
    public void actionPerformed(ActionEvent event)
```

```
// Listener for the button events
{
  . . .
  if (event.getActionCommand().equals("Reset"))
  { // Handles Reset event
    . . .
  }
  else
  if (event.getActionCommand().equals("Next"))
  { // Handles Next event
    . . .
  }
  . . .
}
```

As you can see, `ActionHandler` implements the `ActionListener` interface. Therefore, `action` is also an `ActionListener`. Action listeners are just one of several Java listener types. Another example is window listeners, which we use to close our frames. We use action listeners for our user interfaces. We return to the definition of `ActionHandler` below. First, let's see how we "connect" the event source and the event listener.

The `action` listener is registered with the `reset` button with the command

```
reset.addActionListener(action);
```

As you can see in the program code, the action listener is also registered with the other five buttons.

The registration of an event listener with an event source means that whenever an event occurs to the event source, such as a button being pressed, an announcement of the event is passed to the event listener. There are all sorts of events supported by Java. In our case we are only interested in "action" events, a subset of the set of all potential events, so we use an `ActionListener` listener.

How does the event source send "an announcement" of an event to the listener? Through a call to one of the listener's methods, of course. In the case of action events, the source calls the listener's `actionPerformed` method, and passes it an `ActionEvent` object that represents the event that occurred. The `ActionListener` interface declares an abstract `actionPerformed` method, so we know that any class that implements `ActionListener`, like our `ActionHandler` class, must provide an implementation of `actionPerformed`. You do not see a call to the `actionPerformed` method anywhere in our program. We do not explicitly invoke the method in our code; it is automatically called when a button is pressed by the user. Such a method invocation is sometimes called an *implicit invocation*.

Let's review. In the Real Estate program, we have six buttons that are event sources. We have one event listener, *action*, which has been registered through `addActionListener` to all six buttons. When any of the buttons are pressed the `actionPerformed` method of the `action` object is invoked, and passed an `event` object that represents the button-pressed

event. At that point, the `actionPerformed` method must respond to the event. How does it do that?

Look again at the code listed above for the `ActionHandler` class. You can see that the `actionPerformed` method is implemented as a series of *if-else* statements.

```
if (event.getActionCommand().equals("Reset"))
{ // Handles Reset event
  ...
}
else
```

Each *if*-block handles a different button being pressed. The boolean expressions use the `getActionCommand` method of the `ActionEvent` class to obtain a string signifying the specific real event that the `event` object represents. In the case of button pressing events, this string is simply the string displayed on the face of the button. Therefore, when a button is pressed, the appropriate *if*-block is executed. Take a minute to browse the code in the *if*-blocks to see how the program handles each of the buttons being pressed.

Presenting the Interface

Now that we have created the frame, the labels, the text fields, the buttons and associated actions with each of the buttons, we are ready to build and display the interface. We use the same approach we did for the Chapter 1 test driver program. First, we set up a 10 × 2 grid in our panel:

```
infoPanel.setLayout(new GridLayout(10,2));
```

Next we add all of our components to our panel, in the order we want them to appear (left to right, top to bottom):

```
infoPanel.add(statusLabel);
infoPanel.add(blankLabel);
infoPanel.add(lotLabel);
infoPanel.add(lotText);
...
infoPanel.add(find);
```

Finally, we add the panel to the frame's content pane, and show the frame:

```
contentPane.add(infoPanel);
displayFrame.show();
```

Summary

In this chapter, we have created two abstract data types that represent lists. The Unsorted List ADT assumes that the list elements are not sorted by key; the Sorted List ADT assumes that the list elements are sorted by key. We have viewed each from three perspectives: the logical level, the application level, and the implementation level. The extended Case Study uses the Sorted List ADT to help solve a problem. Figure 3.18 shows the relationships among the three views of the list data in the Case Study.

As we progressed through the chapter we expanded our use of Java constructs to support the list abstractions. In the first part of the chapter, we worked through the following variations of lists:

- `UnsortedStringList`—an unsorted list of strings
- `StringList`—an abstract string list specification; valid for both sorted and unsorted lists
- `UnsortedStringList2`—an extension of `StringList`
- `SortedStringList`—another extension of `StringList`

In order to make the software as reusable as possible, we learned how to use the Java *interface* mechanism to create generic ADTs. The user of the ADT must prepare a class that defines the objects to be in each container class. In the case of the list abstraction, objects to be contained on a list must implement the `Listable` interface; therefore, they must have an appropriate `compareTo` and a `copy` method associated with them. By requiring the user to meet this standard for the objects on the list, the code of the ADTs is very general.

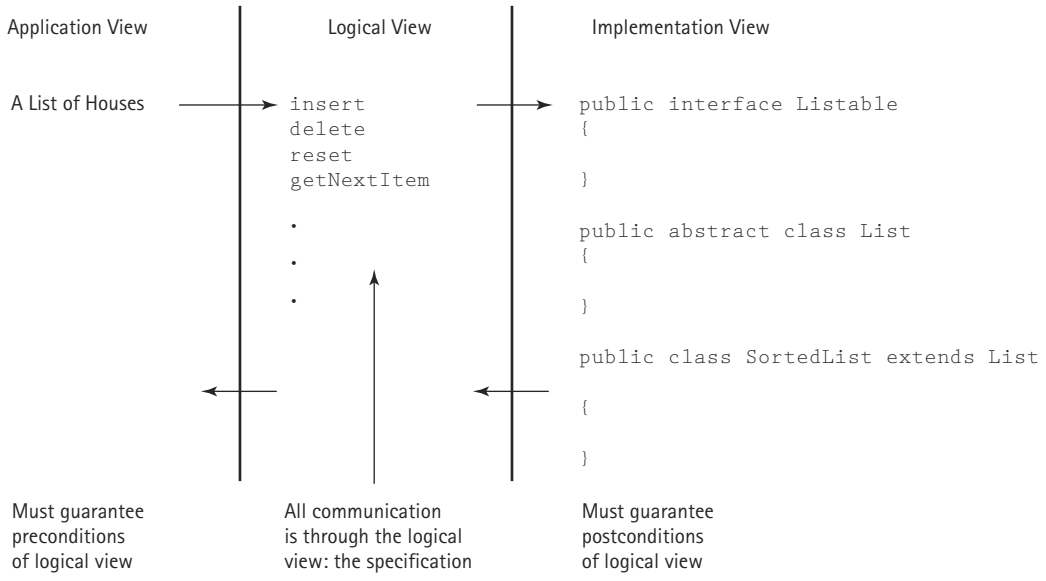


Figure 3.18 *Relationships among the views of data*

The Unsorted List or Sorted List ADT can process items of any kind; they are completely context independent. Within the chapter we saw examples of how to create lists of circles, strings, and houses. The ability to create generic structures led to two more list variations:

- `List`—an abstract list specification, no longer tied to strings; includes a `retrieve` operation
- `SortedList`—an extension of `List`

We compared the operations on the two ADTs using Big-O notation. Insertion into an unsorted list is O(1); insertion into a sorted list is O(N). Deletions from both are O(N). Searching in the unsorted list is O(N); searching in a sorted list is order O($\log_2 N$) if a binary search is used.

We have also seen how to write test plans for ADTs.

Summary of Classes and Support Files

The classes and files are listed in the order in which they appear in the text. Inner classes are not included. The package a class belongs to, if any, is listed in parentheses under Notes. The class and support files are available on our web site. They can be found in the `ch03` subdirectory of the `bookFiles` directory.

Classes, Interfaces, and Support Files Defined in Chapter 3

File	1st Ref.	Notes
`UnsortedStringList.java`	page 150	(`ch03.stringLists`) Array-based implementation of an unsorted string list ADT
`TDUnsortedStringList.java`	page 160	Test driver for `UnsortedStringList.java`
`StringList.java`	page 165	(`ch03.stringLists`) Abstract class—defines all the constructs for an array based list of strings that do not depend on whether or not the list is sorted
`UnsortedStringList2.java`	page 166	(`ch03.stringLists`) Extends `StringList` under the assumption that the list is *not* kept sorted
`SortedStringList.java`	page 181	(`ch03.stringLists`) Extends `StringList` under the assumption that the list *is* kept sorted
`Listable.java`	page 194	(`ch03.genericLists`) Interface—objects used with the following list classes must be derived from classes that implement this interface
`ListCircle.java`	page 195	(`ch03.genericLists`) Example of a class that implements `Listable`

(continued on next page)

File	1st Ref.	Notes
`List.java`	page 197	(`ch03.genericLists`) Abstract class—defines all the constructs for an array-based generic list that do not depend on whether or not the list is sorted; the list stores objects derived from a class that implements `Listable`; includes a `retrieve` method, that was not part of the previous lists
`SortedList.java`	page 200	(`ch03.genericLists`) Extends `List` under the assumption that the list *is* kept sorted
`ListString.java`	page 204	(`ch03.genericLists`) Another example of a class that implements `Listable`
`TDSortedList.java`	page 206	Test driver for `SortedList.java`
`ListHouse.java`	page 215	(`ch03.houses`) Implements `Listable`; provides information about a house that can be stored on a list
`HouseFile.java`	page 218	(`ch03.houses`) Manages the `houses.dat` file
`RealEstate.java`	page 224	The real estate application
`testlist1.dat`	page 162	Test data for the `TDUnsortedStringList` program
`testout1.dat`	page 162	Results of using `testlist1.dat` as input to the Unsorted String List test driver
`testlist2.dat`	page 206	Test data for the `TDSortedList` program
`testout2.dat`	page 206	Results of using `testlist2.dat` as input to the Sorted List test driver

The diagrams in Figure 3.19 show the relationships among the classes listed above. Abstract classes are shown in (*Italics*) within parentheses, interfaces are shown within <brackets>, and applications are boxed. Relationships are shown by arrows using UML standard representations (solid arrow with hollow arrowhead represents the inheritance relationship "extends," dotted arrow with hollow arrowhead represents the implements relationship between a class and an interface, and dotted arrow with open arrowhead represents a "uses" relationship—the latter relationships are also labeled "uses.") Finally, the package groupings are indicated by "blue rectangles."

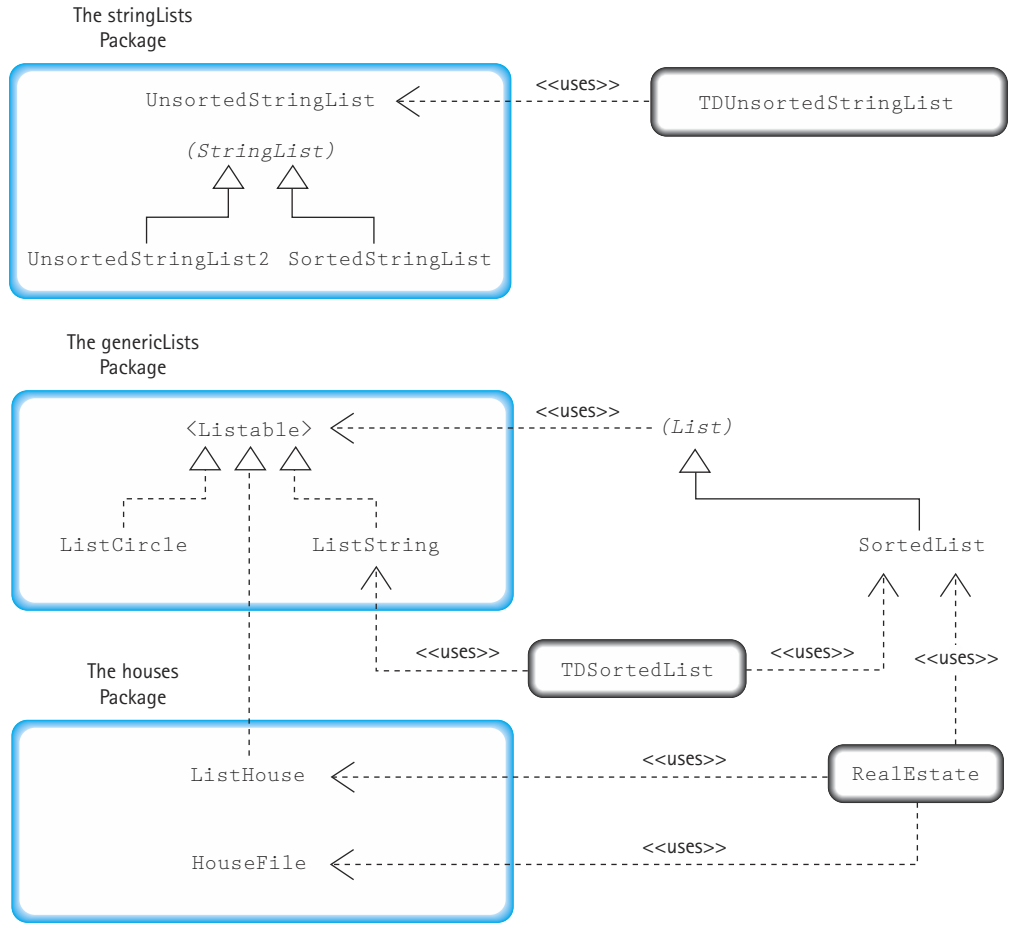


Figure 3.19 *Chapter 3 classes and their relationships*

On page 241 is a list of the Java Library Classes that were used in this chapter for the first time in the textbook. The classes are listed in the order in which they are first used. Note that in some classes the methods listed might not be defined directly in the class; they might be defined in one of its superclasses. With the methods we also list constructors, if appropriate. For more information about the library classes and methods the reader can check Sun's Java documentation.

Library Classes Used in Chapter 3 for the First Time

Class Name	Package	Overview	Methods Used	Where Used
`JTextField`	`swing`	Provides a container for a single line of user text	`getText`, `JTextField`, `setText`	`RealEstate`
`ActionListener`	`awt.event`	An interface for classes that listen for and handle action events		`RealEstate`
`ActionEvent`	`awt.event`	Provides objects for passing event information between event sources and event listeners	`getActionCommand`	`RealEstate`
`JButton`	`swing`	Provides a container for an interface button	`ActionListener`, `JButton`,	`RealEstate`
`LineBorder`	`swing`	Sets a border for the display of a component	`LineBorder`	`RealEstate`
`Color`	`lang`	Provides color constants		`RealEstate`

Exercises

3.1 Lists

1. Give examples from the "real world" of unsorted lists, sorted lists, lists that permit duplicate keys, and lists that do not permit duplicate keys.
2. Describe how the individuals in each of the following groups of people could be uniquely identified; that is, what would make a good key value for each of the groups.
   a. Citizens of a country who are eligible to vote
   b. Members of a sports team
   c. Students in a school
   d. E-mail users
   e. Automobile drivers
   f. Actors/actresses in a play

3.2 Abstract Data Type Unsorted List

3. Classify each of the Unsorted List ADT operations (`UnsortedStringList`, `isFull`, `lengthIs`, `isThere`, `insert`, `delete`, `reset`, `getNextItem`) according to operation type (Constructor, Iterator, Observer, Transformer).
4. The chapter specifies and implements an Unsorted List ADT (for strings).
   a. Design an algorithm for an application-level routine `printLast` that accepts a list as a parameter and returns a `boolean`. If the list is empty, the routine prints "List is empty" and returns `false`. Otherwise, it prints the last item of the list and returns `true`. The signature for the routine should be

```
boolean printLast(PrintWriter outfile, UnsortedStringList list)
```

   b. Devise a test plan for your algorithm.
   c. Implement and test your algorithm.
5. The chapter specifies and implements an Unsorted List ADT (for strings).
   a. Design an algorithm for an application level routine that accepts two lists as parameters, and returns a count of how many items from the first list are also on the second list. The signature for the routine should be

```
int compareLists(UnsortedStringList list1, UnsortedStringList list2)
```

   b. Devise a test plan for your algorithm.
   c. Implement and test your algorithm.
6. What happens if the constructor for `UnsortedStringList` is passed a negative parameter? How could this situation be handled by redesigning the constructor?
7. A friend suggests that since the `delete` operation of the Unsorted List ADT assumes that the parameter element is already on the list, the designers may as well assume the same thing for other operations since it would simplify things. Your friend wants to add the assumption to both the `isThere` and the `insert` operations! What do you think?
8. Describe the ramifications of each of the following changes to the chapter's code for the indicated `UnsortedStringList` methods.
   a. `isFull` change "return (list.length == numItems);" to "return (list.length = numItems);"
   b. `lengthIs` change "return numItems;" to "return list.length;"
   c. `isThere` change the second "moreToSearch = (location < numItems);" to "moreToSearch = (location <= numItems);"
   d. `insert` remove "numItems++;"
   e. `delete` remove "numItems--;"
9. The test plan on page 161 for the `UnsortedStringList` class was not complete.
   a. Complete the test plan.
   b. Create a set of test input files that represents the completed test plan.

c. Use the `TDUnsortedStringList` program, available with the rest of the textbook's programs, to run and verify your tests.

10. The Unsorted List ADT (for `UnsortedStringList`) is to be extended with a `boolean` operation, `isEmpty`, which determines whether or not the list is empty.
    a. Write the specifications for this operation.
    b. Write a method to implement the operation.
11. The Unsorted List ADT (for `UnsortedStringList`) is to be extended with an operation, `smallest`, which returns a copy of the "smallest" list element. It is assumed that the operation will not be invoked if the list is empty.
    a. Write the specifications for this operation.
    b. Write a method to implement the operation.
12. Rather than enhancing the Unsorted List ADT by adding a `smallest` operation, you decide to write a client method to do the same task.
    a. Write the specifications for this method.
    b. Write the code for the method, using the operations provided by the Unsorted List ADT
    c. Write a paragraph comparing the client method and the ADT method (Exercise 11) for the same task.
13. The specifications for the Unsorted List ADT `delete` operation state that the item to be deleted is in the list.
    a. Create a specification for a new form of delete, called `tryDelete`, that leaves the list unchanged if the item to be deleted is not in the list. The new delete operation should return a `boolean` value `true` if the item was found and deleted, `false` if the item was not on the list.
    b. Implement `tryDelete` as specified in (a).
14. The specifications for the Unsorted List ADT state that the list contains unique items. Suppose this assumption is dropped, and the list is allowed to contain duplicate items.
    a. How would the specification have to be changed?
    b. Create a specification for a new form of delete for this new ADT, called `deleteAll`, that deletes all list elements that match the parameter item's key. You should still assume that at least one matching item is on the list.
    c. Implement `deleteAll` as specified in (b).
15. The text's implementation of the `delete` operation for the Unsorted List ADT (`UnsortedStringList`) does not maintain the order of insertions because the algorithm swaps the last item into the position of the one being deleted and then decrements length.
    a. Would there be any advantage to having `delete` maintain the insertion order? Justify your answer.

b. Modify `delete` so that the insertion order is maintained. Code your algorithm, and test it.

16. Change the specifications for the Unsorted List ADT so that `insert` throws an exception if the list is full. Implement the revised specification.

17. Create a new implementation of the Unsorted List ADT (`UnsortedStringList`) using the Java Library's `ArrayList` class instead of plain arrays.

3.3 Abstract Classes

18. The abstract class `StringList` contains both abstract and concrete methods.
    a. List the abstract methods.
    b. List the concrete methods.
    c. Explain the difference between an abstract method and a concrete method.

19. Suppose you wanted to add the operation `isEmpty`, as defined in Exercise 10, to the `StringList` class. Would you make it an abstract method or a concrete method? Justify your answer.

20. Suppose you wanted to add the operation `smallest`, as defined in Exercise 11, to the `StringList` class. Would you make it an abstract method or a concrete method? Justify your answer.

21. Consider the UML diagram in Figure 3.5.
    a. What does the "+" symbol represent?
    b. What does the "#" symbol represent?
    c. What does the arrow represent?
    d. Why are some of the method names italicized?
    e. Why is the variables section of the class diagram for the `UnsortedStringList2` class empty?

3.4 Abstract Data Type Sorted List

22. The Sorted List ADT (for `SortedStringList`) is to be extended with an operation, `smallest`, which returns a copy of the "smallest" list element. It is assumed that the operation will not be invoked if the list is empty.
    a. Write the specifications for this operation.
    b. Write a method to implement the operation.

23. Rather than enhancing the Sorted List ADT by adding a `smallest` operation, you decide to write a client method to do the same task.
    a. Write the specifications for this method.
    b. Write the code for the method, using the operations provided by the Sorted List ADT.
    c. Write a paragraph comparing the client method and the ADT method (Exercise 22) for the same task.

24. The algorithm for the Sorted List ADT insert operation starts at the beginning of the list and looks at each item, to determine where the insertion should take place. Once the insertion location is determined, the algorithm moves each list item between that location and the end of the list, starting at the end of the list, over to the next position. This creates space for the new item to be inserted. Another approach to this algorithm is just to start at the last location, examine the item there to see if the new item should be placed before it or after it, and shift the item in that location to the next location if the answer is "before." Repeating this procedure with the next to last item, then the one next to that, and so on, will eventually move all the items that need to be moved, so that when the answer is finally "after" (or the beginning of the list is reached) the needed location is available for the new item.
    a. Formalize this new algorithm with a pseudocode description, such as the algorithms presented in the text.
    b. Rewrite the `insert` method of the `SortedStringList` class to use the new algorithm.
    c. Test the new method.
25. The specifications for the Sorted List ADT `delete` operation state that the item to be deleted is on the list.
    a. Create a specification for a new form of delete, called `tryDelete`, that leaves the list unchanged if the item to be deleted is not in the list. The new delete operation should return a `boolean` value `true` if the item was found and deleted, `false` if the item was not on the list.
    b. Implement `tryDelete` as specified in (a).
26. The Sorted List ADT (for `SortedStringList`) is to be extended with an operation `merge`, which adds the contents of a list parameter to the current list.
    a. Write the specifications for this operation. The signature for the routine should be

```
void merge(SortedStringList list)
```

    b. Design an algorithm for this operation.
    c. Devise a test plan for your algorithm.
    d. Implement and test your algorithm.
27. A String List ADT is to be extended by the addition of method `trimList`, which has the following specifications:

trimList(String lower, String upper)

Effect:	Removes all elements from the list that are less than `lower` and greater than `upper`
Postconditions:	This list contains only items that are between `lower` and `upper` inclusive

a. Implement `trimList` as a method of `UnsortedStringList`.
b. Implement `trimList` as a member method of `SortedStringList`.
c. Compare the algorithms used in (a) and (b).
d. Implement `trimList` as a client method of `UnsortedStringList`.
e. Implement `trimList` as a client method of `SortedStringList`.

3.5 Comparison of Algorithms

28. Describe the order of magnitude of each of the following functions using Big-O notation:
    a. $N^2 + 3N$
    b. $3N^2 + N$
    c. $N^5 + 100N^3 + 245$
    d. $3N\log_2 N + N^2$
    e. $1 + N + N^2 + N^3 + N^4$
    f. $(N * (N - 1)) / 2$
29. Give an example of an algorithm (other than the examples discussed in the chapter) that is
    a. O(1)
    b. O(N)
    c. O(N^2)
30. Describe the order of magnitude of each of the following code sections using Big-O notation:
    a.
```
count = 0;
for (i = 1; i <= N; i++)
  count++;
```
    b.
```
count = 0;
for (i = 1; i <= N; i++)
  for (j = 1; j <= N; j++)
    count++;
```
    c.
```
value = N;
count = 0;
while (value > 1)
{
  value = value / 2;
  count++;
}
```
31. Algorithm 1 does a particular task in a "time" of N^3, where N is the number of elements processed. Algorithm 2 does the same task in a "time" of $3N + 1000$.
    a. What are the Big-O requirements of each algorithm?

b. Which algorithm is more efficient by Big-O standards?

c. Under what conditions, if any, would the "less efficient" algorithm execute more quickly than the "more efficient" algorithm?

3.6 Comparison of Unsorted and Sorted List ADT Algorithms

32. Assume that for each of the listed exercises an optimal algorithm was written (optimal means that it is not possible under the circumstances to write a faster algorithm). Give a Big-O estimate of the run time for the corresponding algorithms. Unless otherwise stated, let *N* represent the size of the list.

    a. Exercise 4a: `printList` for `UnsortedStringList`

    b. Exercise 5a: `compareLists` for `UnsortedStringList` (*N* = size of the larger list)

    c. Exercise 10: `isEmpty` for `UnsortedStringList`

    d. Exercise 11: `smallest` for `UnsortedStringList`

    e. Exercise 12: `smallest` for `UnsortedStringList` client

    f. Exercise 13: `tryDelete` for `UnsortedStringList`

    g. Exercise 22: `smallest` for `SortedStringList`

    h. Exercise 23: `smallest` for `SortedStringList` client

3.7 Generic ADTs

33. We did not devise a test plan for the `SortedList` class.

    a. Create an appropriate test plan using the `ListString` class to provide objects for storing on the list. Remember to include tests of the `retrieve` operation.

    b. Create a set of test input files that represents the completed test plan.

    c. Use the `TDSortedList` program to run and verify your tests.

34. Create a new concrete class, `UnsortedList`, that extends the `List` class, as discussed at the end of the section, A Generic Sorted List ADT.

35. Consider a `ListNumber` class that implements the `Listable` interface. The class defines two instance variables, one of primitive type `int` and the other of type `String`. The former acts as the key. The idea is that objects of the class can hold an integer value, for example, 5, and the corresponding string, "five". The class exports a constructor that accepts two parameters that are used to initialize the hidden instance variables, two observer methods that return the values of the hidden instance variables, a `toString` method that returns `value`, and of course the required `compareTo` and `copy` methods.

    a. Create the `ListNumber` class.

    b. Test your `ListNumber` class by using it with the `SortedList` class.

Case Study: Real Estate Listings

36. Devise and perform a thorough test of the Real Estate application program.
37. Explain how you would have to change the Real Estate program to handle each of the following specification changes. For each case, indicate which program units need to be changed and a general description of how the change could be implemented.
    a. The `houses.dat` file is redesigned to include the owner's first name first, and last name second, instead of vice versa.
    b. In the interface "Lot numbers" are to be referred to as "Locations".
    c. The information for each house is augmented by a "Number of bathrooms" attribute.
    d. In a surprising and unconventional move, the company decides that each house will have a unique price, and that houses should be listed in order of price instead of lot numbers.
38. Expand the Real Estate program so that the "blank label" field of the interface is used to always show the total number of houses on the list.
39. Expand the Real Estate program to include two more user interface buttons: largest and smallest. If the list of houses is empty and the user clicks on either of the new buttons, the message "List is empty" should appear in the status label area. Otherwise, when the user clicks on the "largest" button, the program should display the house information for the largest house in terms of square feet; and when the user clicks on the "smallest" button, the program should display the house information for the smallest house in terms of square feet.

Chapter 4

ADTs Stack and Queue

Goals

Measurable goals for this chapter include that you should be able to

- provide a formal specification of an ADT using a Java interface
- explain the benefits of using a Java interface for a formal specification
- describe a stack and its operations at a logical level
- demonstrate the effect of stack operations using a particular implementation of a stack
- implement the Stack ADT in an array-based implementation
- implement the Stack ADT using the Java Library `ArrayList` class
- use the Java *exception* mechanism within an ADT
- describe the strengths and drawbacks of both the store "by reference" and store "by copy" approaches to implementing data structures
- explain the difference between the Java Library classes `Vector` and `ArrayList`.
- define stack, queue, concrete class, subinterface
- identify the Java Library class that most closely resembles the ADTs List, Stack, and Queue defined in the textbook
- describe a queue and its operations at a logical level
- demonstrate the effect of queue operations using a particular implementation of a queue
- implement the Queue ADT using an array-based implementation
- use a Stack or Queue ADT as a component of a solution to an application problem
- evaluate a Postfix expression "by hand"
- describe an algorithm for evaluating Postfix expressions using a stack

In this chapter your toolkit of data structures is expanded to include two important new ones, the stack and the queue. As with lists, we study these data structures as ADTs and look at them from the logical, application and implementation levels. The case studies, and several smaller examples, help you learn how to use the stack and the queue to solve problems.

While learning about the stack and the queue, you continue to build your practical knowledge of the Java language. You learn how to use the Java *interface* construct to specify an ADT and you explore the differences between storing information "by copy" and "by reference." Finally, an overview of the Java Class Library's collections framework introduces you to the wealth of resources available in the library.

4.1 Formal ADT Specifications

In Chapter 3 we developed a specification for an Unsorted List ADT. The specification describes the logical structure and the exported operations of the ADT. For each operation we listed its interface, plus a description of the effect of the operation and any preconditions and postconditions. The specification acts as a contract created by the designer of the ADT, relied upon by the application programmer who uses the ADT, and fulfilled by the programmer who implements the ADT.

The Java language provides a construct, the *interface*, for formally capturing such a specification. Recall that an *interface* may contain only constant values and abstract methods. An abstract method consists of the method's interface description only—no method body.

Our Unsorted List ADT Specification included a description of the interfaces of the public methods of the ADT. In the specification, these are presented as the method headers of the Java code that is used to implement the operations. The Java *interface* construct lets us collect together these method interfaces into a syntactic unit. Therefore, from this point on we formalize the specification of our ADTs by using a Java *interface*. The method interfaces of our specification are listed as Java code. All other parts of the specification are presented as comments. Using the Java *interface* construct for our ADT specifications produces several benefits:

1. We can formally check the syntax of our specification. When we compile the interface, the compiler uncovers any syntactical errors in the method interface definitions.
2. We can formally verify the interface "contract." As mentioned above, a specification acts as a contract between the designer and the implementer of the ADT. The code for the ADT should begin with the statement that it "implements the interface." When we compile the ADT implementation, the compiler enforces the contract, at least as far as the information about method names, parameters, and return types.
3. We can assume a consistent interface among alternate implementations of the ADT. We sometimes create alternate implementations for an ADT, perhaps to emphasize differing design criteria. Some implementations may optimize the use of memory space; others may emphasize the efficiency of a specific subset of the ADT operations. If all of the ADT implementations "implement" the same interface, then we

are assured that they provide a consistent view to the client programs. We can substitute one implementation for another, without having to be concerned about the syntax of their interfaces. Of course, syntactic correctness does not imply that the functionality of an ADT implementation is correct.

For an example, we return to the List ADT developed in Chapter 3. Let's first review the way our approach to the List ADT evolved in that chapter:

- We developed the specification of an Unsorted List ADT and created an array based implementation of a list of strings (Section 3.2, Abstract Data Type Unsorted List).
- We developed an abstract class `StringList` (concrete methods `StringList`, `isFull`, `lengthIs`, `reset`, and `getNextItem`; abstract methods `isThere`, `insert`, `delete`) and an `UnsortedStringList` class that extended the abstract class, providing implementations for the abstract methods under the assumption that the list was not kept sorted (Section 3.3, Abstract Classes).
- We developed a `SortedStringList` class that extended the abstract class `StringList`, providing implementations for the abstract methods under the assumption that the list was kept sorted (Section 3.4, Abstract Data Type Sorted List).
- We developed the `Listable` interface that defines the kinds of objects we could henceforth use with our lists, a generic abstract list class, `List`, that used items of type `Listable` instead of type `String` and that included a `retrieve` method, and a `SortedList` class that extended `List` (Section 3.7, Generic ADTs).

Below we define an interface that captures our list model as it stood at the end of Chapter 3. Therefore, our specification does not assume that the list is sorted or unsorted; the list manipulates items of type `Listable`; and a `retrieve` operation is included. When using an interface to specify an ADT we include the effect, precondition, postcondition, and exception information. Following the convention established in Chapter 3, we do not repeat all of this information in each of the classes that implement the interface, since the repetition would be monotonous in a textbook setting, although, in a professional programming situation, where the interface and implementation may be kept separate, it is common to repeat it.

We call the interface `ListInterface`. Note that it does not include any constructors. This is because, just as for an abstract class, you cannot instantiate objects of an interface. You implement the interface with a class and instantiate objects of that class. The class that implements the interface provides appropriate constructors.

```
//--------------------------------------------------------------------------
// ListInterface.java            by Dale/Joyce/Weems                Chapter 4
//
// Interface for a class that implements a list of unique elements, i.e.,
// no duplicate elements as defined by the key of the list.
// The list has a special property called the current position -- the position
// of the next element to be accessed by getNextItem during an iteration
// through the list. Only reset and getNextItem affect the current position.
//--------------------------------------------------------------------------

package ch04.genericLists;
```

```
public interface ListInterface
{
  public boolean isFull();
  // Effect:        Determines whether this list is full
  // Postcondition: Return value = (this list is full)

  public int lengthIs();
  // Effect:        Determines the number of elements on this list
  // Postcondition: Return value = number of elements on this list

  public boolean isThere (Listable item);
  // Effect:        Determines if element matching item is on this list
  // Postcondition: Return value = (element with the same key as item is on
  //                this list)

  public Listable retrieve(Listable item);
  // Effect:        Returns a copy of the list element with the same key as
  //                item
  // Preconditions: Item is on this list
  // Postcondition: Return value = (list element that matches item)

  public void insert (Listable item);
  // Effect:        Adds a copy of item to this list
  // Preconditions: This list is not full
  //                Element matching item is not on this list
  // Postcondition: Copy of item is on this list

  public void delete (Listable item);
  // Effect:        Deletes the element of this list whose key matches item's
  //                key
  // Preconditions: One and only one element on list has a key matching item's
  //                key
  // Postcondition: No element on list has a key matching the argument item's
  //                key

  public void reset();
  // Effect:        Initializes current position for an iteration through this
  //                list
  // Postcondition: Current position is first element on this list

  public Listable getNextItem ();
  // Effect:        Returns a copy of the element at the current position on
  //                this list and advances the value of the current position
```

```
  // Preconditions:  Current position is defined.
  //                 There exists a list element at current position.
  //                 No list transformers called since most recent call to
  //                 reset
  // Postconditions: Return value = (a copy of element at current position)
  //                 If current position is the last element then current
  //                 position is set to the beginning of this list, otherwise
  //                 it is updated to the next position
}
```

We have created a new package, `ch04.genericLists`, that includes all the files from the `ch03.genericLists` package, plus the `ListInterface.java` file. In this package, the abstract class `List` (Section 3.7) is redefined so that its header reads

```
public abstract class List implements ListInterface
```

That associates the abstract `List` class with our new `List` interface, and therefore ties together all of the files in the package.

The UML diagram in Figure 4.1 captures the relationships among the various code units that implement our list approach. We say that this diagram models our list framework. In the diagram, we show that the `ListInterface` interface "uses" the `Listable` interface. Actually, the `List`, `UnsortedList`, and `SortedList` classes also all use the `Listable` interface, but to keep the diagram from becoming too complicated, we display only the relationship for the component at the highest level of abstraction.

When client programs use an ADT, it is good practice for them to declare the ADT at as abstract a level as possible. This makes it easier to change the choice of implementation at a later time. For example, if a program uses a variable called `theList` of type `SortedList`, the variable should be declared as type `ListInterface` but instantiated as type `SortedList` as follows:

```
ListInterface theList;
theList = new SortedList();
```

Alternately, these could be combined into the single statement

```
ListInterface theList = new SortedList();
```

In this way the instantiated type of `theList` can be changed to a different implementation of `ListInterface` by changing only one line of the program, the line where the instantiation occurs.

If the programmer creates routines that are passed `theList` as an argument, the routines should be written to accept parameters of type `ListInterface`, rather than of type `SortedList`. For example, a routine to print the contents of the list might begin

```
public static void Print(ListInterface aList)
```

Figure 4.1 *UML class diagram for list approach*

In this way, the `Print` routine will work for any kind of list, as long as the list class implements `ListInterface`.

In the next section we use a Java interface to specify a Stack ADT.

4.2 Stacks

In Chapter 2, we looked at the built-in structures of Java from the logical view, the application view, and the implementation view. We saw that at the language level, the logical view is the syntax of the construct itself, and the implementation view is hidden within the run-time environment. In Chapter 3, we defined the ADTs Unsorted List and Sorted List. For these user-defined ADTs, the logical view is the class definition where the exported methods become the interface between the client program and the ADT. In Section 4.1 we saw how to formally capture this view using the Java *interface* construct. We now examine two very useful data structures as ADTs: the stack and the Queue. We place the files associated with the Stack ADT in a package named `ch04.stacks`, and the ones for the Queue ADT in a package named `ch04.queues`. We begin with a discussion of the stack.

Logical Level

Stack A structure in which elements are added and removed from only one end; a "last in, first out" (LIFO) structure

Consider the items pictured in Figure 4.2. Although the objects are all different, each illustrates a common concept—the stack. At the logical level, a stack is an ordered group of homogeneous items or elements. The removal of existing items and the addition of new ones can take place only at the top of the stack. For instance, if your favorite blue shirt is underneath a faded, old, red one in a stack of shirts, you must first remove the red shirt (the top item) from the stack. Only then can you remove the desired blue shirt, which is now the top item in the stack. The red shirt may then be replaced on the top of the stack or thrown away.

The stack may be considered an "ordered" group of items because elements occur in sequence according to how long they've been in the stack. The items that have been in

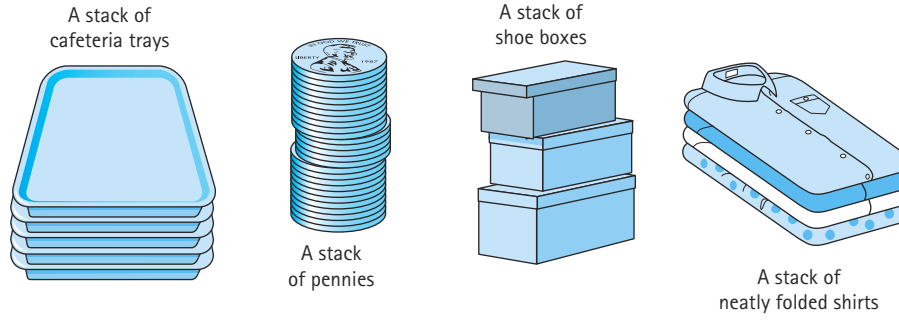


Figure 4.2 *Real-life stacks*

the stack the longest are at the bottom; the most recent are at the top. At any time, given any two elements in a stack, one is higher than the other. (For instance, the red shirt was higher in the stack than the blue shirt.)

Because items are added and removed only from the top of the stack, the last element to be added is the first to be removed. There is a handy mnemonic to help you remember this rule of stack behavior: A stack is a LIFO (Last In, First Out) structure.

The accessing protocol for a stack is summarized as follows: Both to retrieve elements and to store new elements, access only the top of the stack.

Operations on Stacks

The logical picture of the structure is only half the definition of an abstract data type. The other half is a set of operations that allows the user to access and manipulate the elements stored in the structure. Given the logical view of a stack, what kinds of operations do we need in order to use a stack?

The operation that adds an element to the top of a stack is usually called `push`, and the operation that removes the top element off the stack is referred to as `pop`. (Since we are about to implement these operations as Java methods of the same names, we show them in keyword font.) Classically the `pop` operation has both removed the top element of the stack, and returned the top element to the client program that invoked `pop`. More recently, programmers have been defining two separate operations to perform these actions. Software engineers have discovered that implementing operations that do more than one action can result in confusing programs. You should avoid creating methods with side effects when possible. We follow the modern conventions and define an operation `pop` that removes the top element from a stack, and an operation `top` that returns the top element of a stack.[1] Our `push` and `pop` operations are transformers, and our `top` operation is an observer. Figure 4.3 shows how a stack, envisioned as a stack of building blocks, is modified by several `push` and `pop` operations.

When we begin using a stack, it should be empty. In fact, this is true for all of the ADTs that we define in this text. So from here on, we assume that each ADT we define is implemented with at least one class constructor that sets it to the empty state.

Exceptional Situations

Now we ask if there are any exceptional situations with respect to using a stack that require handling. We've identified the operations `push`, `pop`, and `top` and we also need a constructor. The constructor simply initializes a new stack—there are no situations where this action, in itself, causes an error.

The remaining operations, on the other hand, all present potential problem situations. The descriptions of the `pop` and `top` operations refer to manipulating the "top element of the stack." But what if the stack is empty? Then there is no top element to manipulate. We handle this situation in two ways. First, we provide an additional stack

---

[1] Another common approach is to define a `pop` operation in the classical way, i.e., it removes and returns the top element, and to define another operation, often called `peek`, that simply returns the top element.

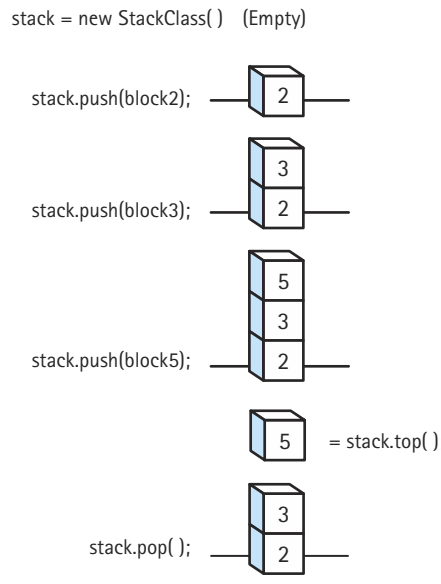


Figure 4.3 *The effects of push and pop operations*

observer operation called `isEmpty`, which returns a `boolean` value indicating whether or not the stack is empty. The application program can use this operation to prevent misuse of the `pop` and `top` operations:

```
if !myStack.isEmpty()
  value = myStack.top();
```

We also define an exception, the `StackUnderflowException`, to be thrown by both the `pop` and the `top` operations if they are called when the stack is empty. We define this new exception to extend the standard Java `RuntimeException`, since it represents a situation that a programmer can avoid by properly using the stack. The `RuntimeException` class is typically used in such situations. `RuntimeException` exceptions are unchecked, i.e., they do not have to be explicitly caught by a program. Here is the code for the `StackUnderflowException` class:

```
package ch04.stacks;

public class StackUnderflowException extends RuntimeException
{
  public StackUnderflowException()
  {
  }
```

```
  public StackUnderflowException(String message)
  {
    super(message);
  }
}
```

Now the application programmer can decide to prevent popping or accessing empty stacks by using the `isEmpty` operation as a guard, or to blindly "try" the operations on stacks and "catch and handle" the raised exception in the case of an empty stack. As noted, since the `StackUndeflowException` extends `RuntimeException`, it is an unchecked exception. If it is raised and not caught, it is eventually thrown out to the run-time environment and the program displays an error message and halts.

A consideration of the `push` operation leads to a similar conclusion. As a logical data structure, a stack is never conceptually "full," but for a particular implementation the space allocated for the stack may be completely used up, making the `push` operation problematic. Again, we handle this situation in two ways. First, we provide an additional stack observer operation called `isFull`, which returns a `boolean` value indicating whether or not the stack is full. The application program can use this operation to prevent misuse of the `push` operation. We also define an exception, the `StackOverflowException`, to be thrown by the `push` operation if it is called when the stack is full. Here is the code for the `StackOverflowException` class:

```
package ch04.stacks;

public class StackOverflowException extends RuntimeException
{
  public StackOverflowException()
  {
  }

  public StackOverflowException(String message)
  {
    super(message);
  }
}
```

As with the underflow situation, the application programmer can decide to prevent pushing information onto a full stack through use of the `isFull` operation, or to "try" the operation on a stack and "catch and handle" the raised exception in the case of a full stack. The `StackOverflowException` is also an unchecked exception.

Stack Contents

We now have a logical picture of a stack and are almost ready to use the stack in a program. The part of the program that uses the stack, of course, won't be concerned with how the stack is actually implemented—we want the implementation level to be hidden,

or encapsulated. The accessing operations such as a `push`, `pop`, and `top` are the windows into the stack encapsulation, through which the stack's data are passed. But the program does have to "know" what kinds of things can be put on the stack. Is it a stack of integers, a stack of strings, a stack of circles?

Recall the discussion from Section 3.7, Generic ADTs. In order to implement a generic list ADT we first considered if we could define a list of objects, i.e., of items of class `Object`. Our analysis showed that this approach was not viable, because we needed to be sure that we could compare and copy the items on our lists. Therefore, we created the `Listable` interface, which specifies the methods (`compareTo` and `copy`) that objects used with our lists must provide. This allows us to implement lists of `Listable` objects, knowing that these objects can be manipulated by the list methods successfully.

What methods must be available for objects that we wish to store and retrieve using a stack? Must those objects support a `compareTo` operation? No, since we do not intend to "order" the objects based on their values, and we do not need to support operations such as `isThere` or `retrieve`, that depend upon matching a key value. Must those objects support a `copy` operation? Yes, if we want to keep only copies of objects on the stack, rather than the objects themselves. This is what we did with lists in Chapter 3. However, the alternate approach is also viable—implement a stack of objects by storing and retrieving references to the objects, rather than references to copies of the objects. Since we believe it is important for you to see and understand both approaches, we use the latter one in this chapter. The ramifications of this choice are discussed in the feature section: Implementing ADTs "by Copy" or "by Reference."

Implementing ADTs "by Copy" or "by Reference"

When designing an ADT, such as for a list or a stack structure, we have a choice about how to handle the elements—"by copy" or "by reference". This feature section describes each approach and discusses the ramifications of the design decision.

By Copy

With this approach, the ADT manipulates copies of the data used in the client program. When the ADT is presented with a data element to store, it makes a copy of the element and stores the copy. For example, code for a list `insert` operation might be

```
public void insert (Listable item)
// Adds a copy of item to this list
{
  list[numItems] = item.copy();
  numItems++;
}
```

In Java, of course, if the list elements are objects, then it is really a reference to a copy of the element that is stored—since all Java objects are manipulated by reference. The key distinction here is that it is a reference **to a copy** of the element, and not a reference to the element itself, that is stored.

Similarly, when an ADT returns an element, it actually returns a reference to a copy of the element. For example, code for a list `getNextItem` operation:

```
public Listable getNextItem ()
// Returns copy of the next element on this list
{
  Listable next = list[currentPos];
  if (currentPos == numItems---1)
    currentPos = 0;
  else
    currentPos++;
  return next.copy();
}
```

This approach provides strong information hiding. In effect, the ADT is providing a separate repository for a copy of the client's data. We used this approach for the list ADTs in Chapter 3.

By Reference

With this approach the ADT manipulates references to the actual elements passed to it by the client program. For example, code for a list `insert` operation might be

```
public void insert (Listable item)
// Adds item to this list
{
  list[numItems] = item;
  numItems++;
}
```

Since the client program retains a reference to the element, whatever it passed as an argument to the method, we say we have exposed the contents of the ADT to the client program. The ADT still hides the way the data is organized—for example, that an array of objects is used—but it allows direct access to the individual elements of the ADT by the client program through the client program's own references.

This approach is used by most Java textbooks and for the structures in the Java Class Library. It has the benefit that it takes less time and space than the "by copy" method. Copying objects takes time, especially if the objects are large with complicated deep-copying methods. Storing extra copies of objects also requires extra memory. So, the "by reference" approach is an attractive approach—in fact it is the approach usually used in industry. In effect, the ADT is providing an organization for the original client data. We use this approach in this chapter.

Remember, when you use the "by reference" approach, you create aliases of your elements, and so you must deal with the potential problems associated with aliases. If your data elements are immutable, then there are no problems. However, if the elements can be changed, problems can arise. If an element is accessed and changed through one alias, it could disrupt the status of the element when accessed through the other alias. This situation is especially dangerous if the client program can use an alias to change an attribute of an element that is used by the ADT to determine the underlying organization of the data elements. For example, if the client directly changes the value of the key attribute in an object that's stored on a sorted list, then the object may no longer be in the proper place within the list. Because the client's access to the object was not through a method of the sorted list class, the class has no way of knowing that it should correct this situation. A subsequent `retrieve` operation on this unsorted list would likely fail.

An Example

The diagrams in Figure 4.4 show ramifications of both approaches. Suppose we have objects that hold a person's `name` and `weight`. Further suppose that we have a list of these objects sorted by the "key" variable `weight` (not the best choice for a list key, but it helps us make our point in this example). We insert three objects into the list, and then transform one of the objects with a `diet` method, that changes the `weight` of the object. The left-hand side of the figure models the approach of storing references to copies of the objects: the "by copy" approach. The right-hand side models the approach of storing references to the original objects; the "by reference" approach.

The state of the models after the objects have been inserted into the lists, shown in the middle of the page, clearly shows the differences in the underlying implementations. The "by copy" approach creates copies and the list elements reference them; these copies take up space that is not required in the "by reference" approach. It is also clear from the right side of the figure that with the "by reference" approach we are creating aliases for the objects, as we can see more than one reference to an object. Note that in both approaches the list elements are sorted by weight.

The situation becomes more interesting as we modify one of the objects. When the person represented by the `S1` object loses some weight, the `diet` method is invoked to decrease the `weight` of the object. In this scenario, both approaches display problems. In the "by copy" approach we see that the `S1` object has been updated. The copy of the `S1` object maintained on the list is clearly out-of-date. It holds the old weight value. A programmer must remember that such a list stores only the values of objects as they existed at the time of the `insert` operation and changes to those objects are not reflected in the objects stored on the list. The programmer must design the programs to update the list, if appropriate.

In the "by reference" approach, the object referred to by the list contains the up-to-date weight information, since it is the same object referred to by the `S1` variable. However, the list is no longer sorted by the `weight` attribute. Since the update to the `weight` took place without any list activity, the list objects remain in the same order as before. The list structure is now corrupt, and calls to the list methods may behave unpredictably. Instead of directly updating the `S1` object, the program should have removed the object from the list, updated the object, and then reinserted the object onto the list.

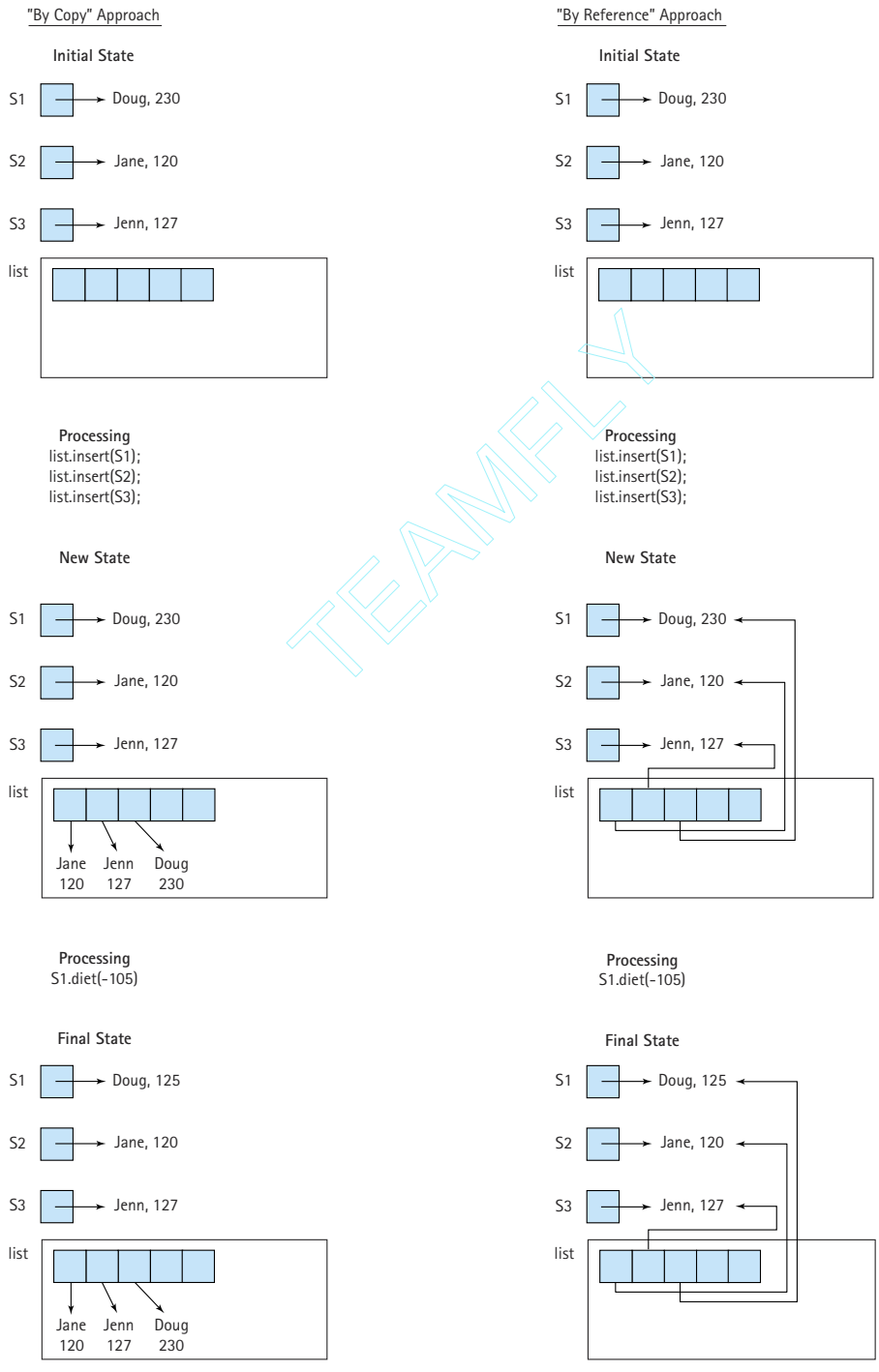


Figure 4.4 *Store "by copy" versus store "by reference"*

Summation

So, which approach is better?

That depends. If processing time and space is an issue, and if you are comfortable counting on the application programs to behave properly, then the "by reference" approach is probably best. If you are not too concerned about time and space (maybe your list objects are not too large), but you are concerned with maintaining careful control over the access to and integrity of your lists, then the "by copy" approach is probably best. The suitability of either approach depends on what the list is used for.

In Chapter 3 we used the "by copy" approach to implement the list ADT. We wanted to emphasize the importance of information hiding. In this chapter, we use the "by reference" approach to implement stack and queue ADTs. We want you to explore both approaches.

Are there any other methods that must be supported by the objects to be stored on the stack? The answer is no. Think about the stack operations we wish to support: `push`, `pop`, `top`, `isEmpty`, and `isFull`. None of them require any specific operation support from the objects being stored. If this is not clear at this point in the discussion, it becomes clear as you work through the implementation of the Stack ADT. Sometimes logical requirements are uncovered during the practical implementation stage, and you have to "work backwards" to revise your logical requirements before again "moving forwards" to work some more on the implementation. In this case, we find that we can successfully implement the stack operations without having to perform any backtracking—they do not require any special operations. Therefore, our stack elements are of the class Object.

The Stack ADT Specification

The interfaces to the accessing operations of our Stack ADT are described in the following specification.

```
//--------------------------------------------------------------------------
// StackInterface.java           by Dale/Joyce/Weems                Chapter 4
//
// Interface for a class that implements a stack of Objects.
// A stack is a last-in, first-out structure
//--------------------------------------------------------------------------

package ch04.stacks;

public interface StackInterface
```

```
{
  public void push(Object item) throws StackOverflowException;
  // Effect:         Adds item to the top of this stack
  // Postconditions: If (this stack is full)
  //                    an unchecked exception that communicates
  //                    'push on stack full' is thrown
  //                 else
  //                    item is at the top of this stack

  public void pop() throws StackUnderflowException;
  // Effect:         Removes top item from this stack
  // Postconditions: If (this stack is empty)
  //                   an unchecked exception that communicates
  //                   'pop on stack empty' is thrown
  //                 else
  //                   top element has been removed from this stack

  public Object top() throws StackUnderflowException;
  // Effect:         Returns a reference to the element on top of this stack
  // Postconditions: If (this stack is empty)
  //                    an unchecked exception that communicates
  //                   'top on stack empty' is thrown
  //                 else
  //                   return value = (top element of this stack)

  public boolean isEmpty();
  // Effect:         Determines whether this stack is empty
  // Postcondition:  Return value = (this stack is empty)

  public boolean isFull();
  // Effect:         Determines whether this stack is full
  // Postcondition:  Return value = (stack is full)
}
```

This specification can be compiled to uncover any syntax errors in the method interfaces. Note that the *interface* documents effects, preconditions, and postconditions as comments. Also note that some of the methods throw unchecked exceptions.

Application Level

Stacks are very useful ADTs. They are often used in situations in which we must process nested components.

For example, programming language systems usually use a stack to keep track of operation calls. The main program calls operation A, which in turn calls operation B,

which in turn calls operation C. When C finishes, control returns to B; when B finishes, control returns to A, and so on. The call and return sequence is essentially a last-in, first-out sequence, so a stack is the perfect structure for tracking it. Recall that when an exception is thrown all the way out to the Java run-time environment that an error message is printed that includes "a system stack trace." This trace shows the nested sequence of method calls that ultimately led to the exception being thrown. These calls were saved on the "system stack."

Compilers often use stacks to perform syntax analysis of language statements. The definition of a programming language usually consists of nested components—for example, *for* loops can contain *if-then* statements that contain *while* loops that contain *for* loops. As a compiler is working through such nested constructs, it "saves" information about what it is currently working on in a stack; then when it finishes its work on the innermost construct, it can "retrieve" its previous status from the stack and pick up where it left off. Similarly, an operating system sometimes saves information about the current executing process on a stack, so that it can work on a higher-priority interrupting process. And if that process is interrupted by an even higher priority process, its information can also be pushed on the process stack. When the OS finishes its work on the highest-priority process, it pops the information about the most recently stacked process, and continues working on it.

Let's look at a simpler problem, related to nested components—the problem of determining if a set of grouping symbols is well formed. This is a classic problem for which a stack is an appropriate data structure. The general problem is to determine if a set of paired symbols is used appropriately. The specific problem is: Given a set of different types of grouping symbols, determine if the open and close versions of each type are used correctly. For our example we'll consider pairs (), [], and {}. Any number of other characters may appear in the input, but a closing symbol must match the last unmatched opening symbol and all symbols must be matched when the input is finished. Figure 4.5 shows examples of both well-formed and ill-formed expressions.

For this problem, we follow the same input/output model we have been using for our test drivers. An input file holds a separate expression on each line. A corresponding output file is created, repeating each of the input lines and stating whether or not it is a well-formed expression. If it is not a well-formed expression, an appropriate error message should be written. The names of the input and output files are

Well-Formed Expressions	Ill-Formed Expressions
(xx (xx ( )) xx)	(xx (xx ( )) xxx) xxx)
[ ] ( ) { }	] [
([ ] { xxx } xxx ( ) xxx)	(xx [xxx) xx]
([{ [(([{ x }]) x)] } x])	([{ [(([{ x }]) x)] } x })
xxxxxxxxxxxxxxxxxxxxxxxxxx	xxxxxxxxxxxxxxxxxxxxxx {

Figure 4.5 *Well-formed and ill-formed expressions*

passed to the program on the command line. Summary statistics are written to an output frame.

The program reads a line of input and processes its characters one at a time. For each character, it does one of three tasks, depending on whether the character is an open special symbol, a closing special symbol, or not a special symbol.

This last case is the easiest. If the character is not a special symbol, it is discarded and another character is read. If the character is an open special symbol, it is saved on the stack. If the character is a closing special symbol, it must be checked against the last open special symbol. If they match, the character and the last open special symbol are discarded and the next character is processed. If the closing special symbol does not match the top of the stack, or if the stack is empty, then the expression is ill-formed. The stack is the appropriate data structure in which to save the open special symbols because we always need to examine the most recently saved one. When all of the characters have been processed, the stack should be empty—otherwise, there were open special symbols left over.

Now we are ready to write the main algorithm, assuming an instance of a Stack ADT as defined by `StackInterface`. The basic flow of the algorithm is to continuously read and handle the input lines. Handling an input line means to look at it character by character following the actions described in the previous paragraph, until we reach the end of the line or until we determine that the expression is ill-formed. To indicate that we have discovered the expression is ill-formed, we set the boolean variable `balancedString` to `false`. If we reach the end of the line without discovering an imbalance, and at that time the stack is empty, then we have a well-formed expression. Otherwise we have an ill-formed expression.

Main Algorithm

```
Initialize expression counts
Read first input line
while there are still lines to process
   Increment the total number of expressions
   Echo print the current expression to the output file
   Create a new stack
   Set balancedString to true
   Get the first character from the current input line
   while (there are still more characters to process AND the expression is still balanced)
      Process the current character
      Get the next character
   if (!balancedString)
      Increment the number of ill-formed expressions
      Write an appropriate message to the output file
```

```
   else
      Increment the number of well-formed expressions
      Write an appropriate message to the output file
   Read the next input line
Write summary information to the output frame
```

The above algorithm follows the basic pattern of:

```
"Read" the first piece of information
while not finished processing information
   Handle the current information
   "Read" the next piece of information
```

It uses this processing pattern for both the lines of expressions and for the characters within each line. It helps increase your programming proficiency to recognize such patterns and reuse them when appropriate.

The only part of the above algorithm that may require expansion before moving on to the coding stage is the "Process the current character" command. We previously described how to handle each type of character. Here are those steps in algorithmic form:

```
if (the character is an open symbol)
   Push the character onto the stack
else if (the character is a closed symbol)
   if the stack is empty
      Set balancedString to false
   else
      Set openSymbol to the character at the top of the stack
      Pop the stack
      if character does not "match" with openSymbol
         Set balancedString to false
else
   Skip the character
```

The code for the program `Balanced` is listed next. It assumes that a class `ArrayStack` implements the `StackInterface` interface. There are several things to note about the program. The code lines related to each of these issues are highlighted in the program listing. We have also highlighted the calls to the stack operations.

1. We declare our stack to be of type `StackInterface`, but instantiate it as type `ArrayStack`, following the convention suggested at the end of Section 4.1.
2. To organize our processing, we create methods `openSet` and `closeSet` that accept a character argument and return a boolean value indicating whether or not the character argument is in the corresponding set of characters.
3. We use the `Character` class as a wrapper. Since our Stack ADT handles objects, we need to wrap the string characters as `Character` objects, before pushing them onto the stack. We then use the `Character` class's `charValue` method to turn them back into primitive characters upon receiving them from the stack with the `top` operation.
4. We use the library's `DecimalFormat` class to ensure consistency of the output lines. The relevant lines are emphasized. First we import the `DecimalFormat` class so that it is available for use:

   ```
   import java.text.DecimalFormat;
   ```

   Next we instantiate an object of the class called `fmt`, initializing it with the string "00". This string indicates that any number formatted with the `fmt` object must provide a digit for both the ten's location and the unit location. If no significant digit is needed, then a 0 should be used.

   ```
   DecimalFormat fmt = new DecimalFormat("00");
   ```

   Finally, on the line where the code echo prints the input lines, numbering them for easy reference, we use the `fmt` object to format the string number:

   ```
   outFile.println("String " + fmt.format(numStrings) + ": " + line);
   ```

   As you can see in the example output file, this ensures that the number uses up two output spaces.

Why do we bother with this? For each input expression we have two output lines. This approach lets us know exactly how many spaces are used for the first output line, and therefore allows us to line up the second output line, through judicious use of blank spaces, exactly with the relevant part of the first line. Note that this works only for the first 99 expressions. If we typically expect more expressions than that, we could initialize `fmt` with the string "000" instead of "00". See the Java Class Library documentation for more information about the `DecimalFormat` class.

```
//--------------------------------------------------------------------------
// Balanced.java              by Dale/Joyce/Weems                  Chapter 4
//
// Checks for balanced special symbols
// Input file consists of a sequence of expressions, one per line
// Special symbol types (), [], and {}
// Output file contains, for each expression:
```

```
//    A copy of the expression
//    Whether or not the expression is balanced
// Input and output file names are supplied by user through command-line
// parameters
// Output frame supplies summary statistics
//--------------------------------------------------------------------------

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import ch04.stacks.*;
import java.text.DecimalFormat;

public class Balanced
{
  public static boolean openSet(char ch)
  // Returns true if ch is one of (, [, {
  {
    return (   (ch == '(')
            || (ch == '[')
            || (ch == '{'));
  }

  public static boolean closeSet(char ch)
  // Returns true if ch is one of ), ], }
  {
    return (   (ch == ')')
            || (ch == ']')
            || (ch == '}'));
  }

  public static void main(String[] args) throws IOException
  {
    int numStrings = 0;              // Total number of strings processed
    int wellFormed = 0;              // Number of well-formed strings found
    int illFormed = 0;               // Number of ill-formed strings found

    char currChar;                   // Current string character being studied
    int  currCharIndex;              // Index of current character
    int  lastCharIndex;              // Index of last character in the string

    char openSymbol;                 // Open symbol character popped from stack

    boolean balancedString;          // True as long as string is balanced
```

```
StackInterface stack;              // Holds unmatched open symbols

String line = null;                // Input line
String dataFileName = args[0];     // Name of input file
String outFileName = args[1];      // Name of output file

BufferedReader dataFile = new BufferedReader(new FileReader(dataFileName));
PrintWriter outFile = new PrintWriter(new FileWriter(outFileName));
DecimalFormat fmt = new DecimalFormat("00");

line = dataFile.readLine();        // Read the first input line.

while(line!=null)                  // While haven't read all of the input
                                   // lines
{
  numStrings++;
  outFile.println("String " + fmt.format(numStrings) + ": " + line);
  outFile.print("            ");

  balancedString = true;
  stack = new ArrayStack();

  currCharIndex = 0;
  lastCharIndex = line.length() - 1;

  while (balancedString && (currCharIndex <= lastCharIndex))
  {
    currChar = line.charAt(currCharIndex);
    outFile.print(currChar);

    if(openSet(currChar))      // If the current character is one of [, {, (
    {
      // Wrap the character and push it onto the stack
      Character openSymbolObject = new Character(currChar);
      stack.push(openSymbolObject);
    }
    else
    {
      if(closeSet(currChar))   // If the current character is one of ], }, )
      {
        try                    // Try to pop a character off the stack
        {
          openSymbol = ((Character)stack.top()).charValue();
          stack.pop();
```

```
            // If the popped character doesn't match
            if ( !(  ((currChar == ')') && (openSymbol == '('))
                   ||((currChar == ']') && (openSymbol == '['))
                   ||((currChar == '}') && (openSymbol == '{'))))
              balancedString = false;
          }
          catch(StackUnderflowException e)     // Stack was empty
          {
            balancedString = false;
          }
        }
      }
      currCharIndex++;              // Set up processing of next character
    }

    if (!balancedString)
    {
      illFormed++;
      outFile.println(" Unbalanced symbols ");
    }
    else
    if (!stack.isEmpty())
    {
      illFormed++;
      outFile.println(" Premature end of string");
    }
    else
    {
      wellFormed++;
      outFile.println(" The string is balanced.");
    }

    outFile.println();
    line = dataFile.readLine();    // Set up processing of next line
  }
  dataFile.close();
  outFile.close();

  //Set up output frame
  JFrame outputFrame = new JFrame();
  outputFrame.setTitle("Balanced Parenthesis");
  outputFrame.setSize(300,200);
  outputFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
        // Instantiate content pane and information panel.
        Container contentPane = outputFrame.getContentPane();
        JPanel infoPanel      = new JPanel();

        // Set layout.
        infoPanel.setLayout(new GridLayout(4,1));

        infoPanel.add(new JLabel("Total Number Of Strings "+ numStrings));
        infoPanel.add(new JLabel("Total Number Of Well Formed Strings "+
                                 wellFormed));
        infoPanel.add(new JLabel("Total Number Of Ill Formed Strings "+
                                 illFormed));
        infoPanel.add(new JLabel("Program completed. Close window to exit"));
        contentPane.add(infoPanel);

        // Show information.
        outputFrame.show();
    }
}
```


Figure 4.6 shows a sample input file, with an associated output file and information frame. The input file contains alternating well-formed and ill-formed expressions from the examples listed previously in Figure 4.5.

Implementation Level

We now consider the implementation of our Stack ADT. After all, methods such as `push`, `pop`, and `top` are not magically available to the Java application programmer. We need to write these routines in order to include them in a program.

An Array-Based Implementation

Because all the elements of a stack are of the same class, `Object`, an array seems like a reasonable structure to contain them. We can put elements into sequential slots in the array, placing the first element pushed onto the stack into the first array position, the second element pushed into the second array position, and so on. The floating "high-water" mark is the top element in the stack. Why, this sounds just like our Unsorted List ADT implementation!

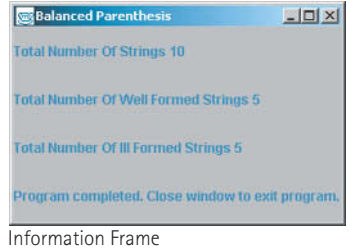
Be careful: We are not saying that a stack is an unsorted list. A stack and an unsorted list are two entirely different data structures. What we are saying is that we can use the same implementation strategy.

```
( xx ( xx ( ) ) xx )
( xx ( xx ( ) ) xxx ) xxx)
[ ] ( ) { }
] [
( [ ] { xxx } xxx ( ) xxx )
( xx [ xxx ) xx ]
( [ { [ ( ( [ { x } ] ) x ) ] } x ] )
( [ { [ ( ( [ { x } ] ) x ) ] } x } )
xxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxx {
```

test1.in file

```
String 01:  ( xx ( xx ( ) ) xx )
            ( xx ( xx ( ) ) xx ) The string is balanced.

String 02:  ( xx ( xx ( ) ) xxx ) xxx)
            ( xx ( xx ( ) ) xxx ) xxx) Unbalanced symbols

String 03:  [ ] ( ) { }
            [ ] ( ) { } The string is balanced.

String 04:  ] [
            ] Unbalanced symbols

String 05:  ( [ ] { xxx } xxx ( ) xxx )
            ( [ ] { xxx } xxx ( ) xxx ) The string is balanced.

String 06:  ( xx [ xxx ) xx ]
            ( xx [ xxx ) Unbalanced symbols

String 07:  ( [ { [ ( ( [ { x } ] ) x ) ] } x ] )
            ( [ { [ ( ( [ { x } ] ) x ) ] } x ] ) The string is balanced.

String 08:  ( [ { [ ( ( [ { x } ] ) x ) ] } x } )
            ( [ { [ ( ( [ { x } ] ) x ) ] } x } Unbalanced symbols

String 09:  xxxxxxxxxxxxxxxxxxxxxxxxx
            xxxxxxxxxxxxxxxxxxxxxxxxx The string is balanced.

String 10:  xxxxxxxxxxxxxxxxxxxxx {
            xxxxxxxxxxxxxxxxxxxxx { Premature end of string
```

test1.out file



Information Frame

Command: java Balanced test1.in test1.out

Figure 4.6 *Sample run of the* `Balanced` *program*

Definition of Stack Class We implement our Stack ADT as a Java class that implements the `StackInterface` interface. We call this class `ArrayStack`, to differentiate it from another implementation approach we present in the next chapter.

What data members does our Stack ADT need? We need the stack items themselves and a variable indicating the top of the stack (which behaves similar to `numItems` in the List ADT). We hold the stack in a private array of `Objects` called `stack` and we hold the index of the top of the stack in a private integer variable called `topIndex`. In the Unsorted List ADT, `numItems` indicated how many items were on the list. In the Stack ADT, `topIndex` indicates which is the top element. So our analogy to the List ADT is off by one. The `topIndex` instance variable is initialized to –1 rather than 0, to indicate that nothing is stored on the stack. (That is, there is no top item in `stack[0]`.)

As was done for the list implementations, we provide two constructors for use by clients of the `Stack` class. One allows the client to specify the maximum expected size of the stack, and the other assumes a default maximum size of 100 elements.

The beginning of the `ArrayStack.java` file is:

```
//-------------------------------------------------------------------------
// ArrayStack.java             by Dale/Joyce/Weems                  Chapter 4
//
// Implements StackInterface using an array to hold the stack items
//-------------------------------------------------------------------------

package ch04.stacks;

public class ArrayStack implements StackInterface
{
  private Object[] stack;               // Array that holds stack elements
  private int topIndex = -1;            // Index of top element in stack

  // Constructors
  public ArrayStack()
  {
    stack = new Object[100];
  }

  public ArrayStack(int maxSize)
  {
    stack = new Object[maxSize];
  }
```

Definitions of Stack Operations The stack observer methods `isEmpty` and `isFull` are very similar to their counterparts in the array list implementations. The only difference relates to the difference between the stack's `topIndex` and the list's `numItems`. The

`isEmpty` method should compare `topIndex` to −1 and `isFull` should compare `topIndex` with one less than the size of the underlying array.

```
public boolean isEmpty()
// Checks if the stack is empty
{
  if (topIndex == -1)
    return true;
  else
    return false;
}

public boolean isFull()
// Checks if the stack is full
{
  if (topIndex == (stack.length - 1))
    return true;
  else
    return false;
}
```

Now, we have to write the method to push an item onto the top of the stack. If the stack is already full when we invoke `push`, there is nowhere to put the item. This condition is called stack overflow. The specifications state that the method should throw the `StackOverflowException` exception in this case. We include a pertinent error message when the exception is thrown. If the stack is not full, `push` must increment `topIndex` and store the new item into `stack[topIndex]`. The implementation of this method is straightforward.

Stack overflow The condition resulting from trying to push an element onto a full stack

```
public void push(Object item)
// Adds an element to the top of this stack
{
  if (!isFull())
  {
    topIndex++;
    stack[topIndex] = item;
  }
  else
    throw new StackOverflowException("Push attempted on a full
stack.");
}
```

Since the `StackOverFlow` exception extends Java's `RunTimeException` class, it is an unchecked exception; so calls to `push` do not have to handle the exception. But they can. Here is an example showing what a client code might do with the exception.

```
try
{
  // Code
  myStack.push(item);
  // More code
}
catch (StackOverFlowException error )
{
  System.out.println(error.getMessage());
  System.exit(1);
}
```

In this case, the client chooses to write the error message to `System.out` and then discontinue processing. By convention, the nonzero status code passed to the `System.exit` method indicates abnormal termination.

The `pop` method is essentially the reverse of `push`: Instead of putting an item onto the top of the stack, we remove the top item from the stack. This can be accomplished by simply decrementing `topIndex`. However, it is good practice to also "null out" the object reference of the array location associated with the current top of stack. Remember that the Java run-time engine reclaims space from objects that no longer have any "live" references to them. Decrementing the `topIndex` removes the stack's logical reference to the object; setting the array value to `null` removes the physical reference. It is the physical reference that is important to the garbage collector. After all, it knows nothing about the logical view of our stack. When all physical references to an object by a program are removed, its space can be reused. Note that we did not follow this convention in our List ADT implementations—we felt that there was already enough language-related complexity being introduced during the study of lists without adding this subtle consideration.

Stack underflow The condition resulting from trying to remove or return an element from an empty stack

If the stack is empty when we invoke `pop`, there is no top element to remove. This condition is called stack underflow. As with the `push` method, the specifications for the operation say to throw an exception.

```
public void pop()
// Removes an element from the top of this stack
{
  if (!isEmpty())
  {
    stack[topIndex] = null;
    topIndex--;
  }
```

```
  else
    throw new StackUnderflowException("Pop attempted on an empty stack.");
}
```

Finally, the `top` operation simply returns the top element of the stack, the element indexed by `topIndex`. As with the `pop` operation, if we attempt to perform the `top` operation on an empty stack, a stack underflow results.

```
public Object top()
// Returns the element on top of this stack
{
  Object topOfStack = null;
  if (!isEmpty())
    topOfStack = stack[topIndex];
  else
    throw new StackUnderflowException("Top attempted on an empty stack.");
  return topOfStack;
}
```

An ArrayList-Based Implementation

There are often many ways to implement an ADT. In this section, we present an alternate implementation for the Stack ADT based on the `ArrayList` class of the Java Class Library.

The `ArrayList` class was introduced in Section 2.3, Class-Based Types. The defining feature of the `ArrayList` class is that it can grow and shrink in response to the program's needs. This means our constructor no longer needs to declare a maximum stack size. Additionally, in this stack implementation, the method `isFull` always returns the value `false`. We do not have to handle stack overflows. One could argue that if a program runs completely out of memory then the stack could be considered full, and should throw the `StackOverflowException`. However, in that case the runtime environment throws an "out of memory" exception anyway; we do not have to worry about the exception going unnoticed. Furthermore, running out of system memory is a serious problem (hopefully a rare event), and cannot be handled the same as a simple stack overflow.

The `ArrayList` class is a good choice for implementing our Stack ADT. Since stacks only grow and shrink from one end, we do not have to worry about the execution overhead associated with inserting an element into the middle of an array list (which requires shifting of multiple elements). And we no longer need to worry about stack overflow. Finally, the array list's `size` method can be used to keep track of the top of our stack. The index of the top of the stack is always the `size` minus 1.

Review the set of `ArrayList` operations described in Chapter 2 and study the following code. Compare this implementation to the previous implementation. They are similar: They both implement the `StackInterface`; yet they are different. One is based directly on arrays and the other uses arrays indirectly through the `ArrayList` class.

```
//-------------------------------------------------------------------------
// ArrayListStack.java           by Dale/Joyce/Weems                Chapter 4
//
// Implements StackInterface using an ArrayList to hold the stack items
//-------------------------------------------------------------------------

package ch04.stacks;

import java.util.*;
import java.util.ArrayList;

public class ArrayListStack implements StackInterface
{
  private ArrayList stack;        // ArrayList that holds stack elements

  // Constructor
  public ArrayListStack()
  {
    stack = new ArrayList();
  }

  public void push(Object item)
  // Adds an element to the top of this stack
  {
    stack.add(item);
  }

  public void pop()
  // Removes an element from the top of this stack
  {
    if (!isEmpty())
    {
      stack.remove(stack.size() - 1);
    }
    else
      throw new StackUnderflowException("Pop attempted on an empty stack.");
  }

  public Object top()
  // Returns the top element from this stack
  {
    Object topOfStack = null;
    if (!isEmpty())
      topOfStack = stack.get(stack.size() - 1);
```

```
    else
      throw new StackUnderflowException("Top attempted on an empty stack.");
    return topOfStack;
  }

  public boolean isEmpty()
  // Checks if this stack is empty
  {
    if (stack.size() == 0)
      return true;
    else
      return false;
  }

  public boolean isFull()
  // Checks if this stack is full
  // Assumes stack is never full since ArrayList implementation can grow as
  // needed
  {
    return false;
  }
}
```

Test Plan

The test plan for the Stack ADT is much like the test plan for the Unsorted List ADT. Because we are testing the implementation of an abstract data type that we have just written, we use a clear-box strategy, checking each operation. Unlike the Unsorted List ADT, we do not have an iterator that allows us to cycle through the items and print them. We must `push` items onto the stack, retrieve them with `top` and print them, and then remove them with `pop`, rather than printing the contents of the stack. Note that the items are output in reverse order.

We also must test the situation in which stack overflow and underflow occur. For purposes of this test, we assume that the exception is handled so that the message is displayed and then the program terminates. Of course, we cannot test for stack overflow in the `ArrayList` based implementation (remember why?). The Expected Output column of our test plan below assumes the array-based implementation.

Because the type of data stored in the stack has no effect on the operations that manipulate the stack, we can use objects of class `Integer`. We set the stack size to 5, to keep our test cases manageable.

Operation to be Tested and Description of Action	Input Values	Expected Output
`ArrayStack`	5	
apply isEmpty immediately		Stack is empty
`push`, `pop`, and `top`		
push 4 items, top/pop and print	5,7,6,9	9,6,7,5
push with duplicates and top/pop and print	2,3,3,4	4,3,3,2
interlace operations		
push	5	
pop		
push	3	
push	7	
pop		
top and print		3
`isEmpty`		
invoke when empty		Stack is empty
push and invoke		Stack is not empty
pop and invoke		Stack is empty
`isFull`		
push 4 items and invoke		Stack is not full
push 1 item and invoke		Stack is full
throw `StackOverflowException`		Outputs string:
push 5 items then		"Push attempted on a full stack."
push another item		Program terminates
throw `StackUnderflowException`		Outputs string:
when stack is empty		"Pop attempted on an empty stack."
attempt to pop		Program terminates
when stack is empty		Outputs string:
attempt to top		"Top attempted on an empty stack."
		Program terminates

4.3 The Java Collections Framework

If you look in the documentation of the Java Class Library, you find a `Stack` class. It is similar to the Stack ADT we developed in this chapter. The Java Library provides classes that implement ADTs that are based on common data structures—stacks, lists, maps, sets, and more.

Another term for a data structure is collection. The Java developers refer to the set of library classes that support data structures, and the rules for using them, as the "collections framework." This framework includes interfaces, abstract classes, concrete classes, and protocols for their use. Recall that an abstract class is one that contains some abstract methods, i.e., methods without method bodies. By concrete class we mean a class in which all of the methods have bodies.

Collection A collection is an object that holds other objects. Typically we are interested in inserting, removing, and iterating through the contents of a collection

Concrete class A concrete class is a class that can have instantiated objects. It does not contain any abstract methods

In this section we introduce you to the *collections framework*. We briefly overview the architecture of the framework and look at a few of the collection classes that are related to the ADTs that you have already studied. As you progress through the text and study additional ADTs, we point out the related library classes, as appropriate.

The collections framework is an extensive set of tools. It does more than just provide abstractions of data structures; it provides a foundation for more advanced programming techniques. It is not our goal to cover all of the intricacies of the framework. This textbook is about fundamental data structures and how we implement them, not about how to use Java's specific implementations of similar structures. Still, it is instructive to see how other programmers have approached similar problems.

Thus, we'll occasionally stop to explain how the library contents relate to the data structures we are studying. We also want you to be able to use library classes, when appropriate, to solve problems. In fact we have already used one of the library collection classes in this chapter. We used the `ArrayList` class to implement a Stack ADT in Section 4.2.

Properties of Collections Framework Classes

Because the collection classes share a common framework, there are some properties that they all have in common. All Java library collection implementation classes

- Allow a collection's size to increase dynamically as needed.
- Are defined to hold items of class `Object`, to ensure the collections can use any object as a component.
- Hold references to the actual objects provided by the client programs, rather than references to copies of the objects.

- Provide at least two constructors—one that creates an empty collection and one that accepts as a parameter any other collection object and creates a new collection object that holds the same information.

Although all collection classes behave the same in some ways, there are also ways in which they differ:

- The logical structure of the elements they contain
- The underlying implementation
- The efficiency of the various operations
- The set of supported operations
- Whether or not they allow duplicate elements
- Whether or not they support element "keys"
- Whether or not their elements are sorted in any way

The most appropriate collections framework class to be used to help solve any specific problem depends upon the qualities of the problem. In fact, it might be that no appropriate collections framework class exists, and you have to define a new ADT to best fit the situation.

The Legacy Classes

Prior to the release of Java 1.2 (also called the Java 2 platform), the standard class library supported only a handful of collection classes: `BitSet`, `HashTable`, `Stack`, and `Vector`. These classes continue to be supported in the more extensive Java 2 collections framework, although some of them have improved counterparts in Java 2. We look at each of these *legacy classes* briefly.

The `Vector` class provides a dynamically sized array. The `Vector` class's methods are designed to allow access by different program threads that are running at the same time to a vector object (an advanced feature of Java, called concurrency). This is great for concurrent programming; however, for all of our simple single-threaded programs, it is not so good. The overhead involved in supporting concurrency makes access to vectors very slow. In Java 2, the `Vector` class has been supplanted by the `ArrayList` class, which provides the same functionality but without the concurrency support. Until you learn about concurrent programming, you should not use the `Vector` class.

The `BitSet` class provides an array of bits. We usually think of a bit as something that can store either the value 0 or the value 1. Since we can view a 0 as representing the Boolean value `false`, and a 1 as representing the Boolean value `true`, we can think of a bit set as being an array of Boolean values. In fact, each component of a library `BitSet` has a Boolean value associated with it. The individual bits can be manipulated (set, cleared, examined) and `BitSet` objects can be combined with various Boolean operations (and, or, not). The `BitSet` class is useful if you need to store and manipulate an array of Boolean values—for example, an array that indicates whether or not student *n* attended class on a particular day.

Just like the `Vector` class, the `HashTable` class supports synchronization. For nonconcurrent programs you should use Java 2's `HashMap` class instead. We look at the topic of hashing in Chapter 10.

The `Stack` class is very similar to our Stack ADT. It provides operations for pushing, popping, and testing if the stack is empty. The pop operation is defined in the classic tradition, that is, it both removes the top element from the stack and returns a reference to the object that was in the top position. Recall that for our Stack ADT we separated this functionality into two methods, `pop` that removed the top item without returning it, and `top` that returned the top item without removing it. The library stack provides a method analogous to our `top` method—it is called `peek`. The library stack does not provide an operation to test when the stack is full. This is because the class is implemented using the previously discussed `Vector` class, which provides dynamically sized arrays. In fact, the library `Stack` class extends the `Vector` class. The performance problems of the `Vector` class also plague the library's `Stack` class. For this reason, our implementation of a stack is probably more efficient than the library's implementation.

Java 2 Collections Framework Interfaces

In Section 4.1 you saw how to use a Java interface to define a contract for all List ADT implementations to fulfill. The Java library designers used the same approach when creating the collections framework. There are three fundamental interfaces for the collections framework: `Collection`, `Map`, and `Iterator`.

The Collection Interface

The `Collection` interface is used by collection classes that do not support a unique key value. Examples include unkeyed lists and sets. The following table describes some of the interesting operations listed in the `Collection` interface.

Method Name	Parameter Type	Returns	Operation Performed
`add`	`Object`	`boolean`	Ensures that this collection holds the parameter element. If the collection was changed as a result of the operation, `true` is returned. Otherwise, `false` is returned.
`addAll`	`Collection`	`boolean`	Adds all of the elements of the parameter collection to this collection. If the collection was changed as a result of the operation, `true` is returned. Otherwise, `false` is returned.
`contains`	`Object`	`boolean`	Returns `true` if this collection contains the parameter element. Otherwise, returns `false`.
`isEmpty`	(none)	`boolean`	Returns `true` if this collection is empty; `false` otherwise

Subinterface A subinterface is an interface that extends another interface. It must list all of the abstract methods in the interface it extends, plus it can add more abstract methods of its own.

The library does not provide any direct implementations of this interface. It does provide implementations of subinterfaces of this interface. What is a subinterface? A subinterface is an interface that extends another interface. Just as one Java class can extend another Java class, creating a superclass–subclass inheritance relationship—so can one interface extend another interface. In fact, unlike with classes that can only extend a single class, there is no limit to the number of interfaces that can be extended by a particular interface. We say that Java supports multiple inheritance of interfaces.

The library provides several subinterfaces of the `Collection` interface, including the `List` interface. It is interesting to note that the List ADT we developed in Chapter 3 does not match well with the library structures related to the `List` interface. Since our List ADT supported unique keys, it actually matches the library "map" structures more closely than the "list" structures.

The Map Interface

The `Map` interface is used by library collection classes that map unique keys to values. Like our List ADT, it defines a method to retrieve an object based on its key value. This operation is not supported by the classes that implement the `Collection` interface. Classes that implement the `Map` interface include `AbstractMap`, `HashMap`, and `Hashtable`.

The Iterator Interface

The `Iterator` interface helps guarantee that client programs are able to iterate through collections. It defines methods similar to those we defined for iterating through our lists in Chapter 3. Rather than supplying a reset operation, `Iterator` objects are "reset" when they are instantiated. They do provide a `next` operation that is analogous to our list's `getNextItem` operation. Unlike our list iterations, which "wrap around" if we try to access past the end of the list, `Iterator` objects raise an exception in this case. Therefore, `Iterator` objects support a boolean `hasNext` operation that the client can use to see if there are any more objects available.

An `Iterator` object also allows a client program to remove an element from a collection. The convention is that during an iteration, if the client calls the iterator's `remove` method, the element of the collection that was just visited by the iteration is removed. This allows the client program to obtain an element during an iteration, examine it, determine whether or not it should be removed, and if so, immediately have it removed by the iterator.

A well-defined collection class includes a method that returns a new `Iterator` object for its collections.

The AbstractCollection Class

The library's `AbstractCollection` class is an abstract class that implements the `Collection` interface. The `Collection` interface defines 15 operations. Some of these are

fundamental operations, that is, they depend upon the particular implementation of the collection. Others however, can be implemented in terms of the fundamental operations. This is exactly what the `AbstractCollection` class does.

For example, the `size` method depends upon the underlying implementation, so in the `AbstractCollection` class it is defined as an abstract method. But the `isEmpty` method can be defined in terms of the `size` method:

```
public boolean isEmpty()
{
  return (this.size() == 0);
}
```

By defining as many methods as possible in terms of the fundamental methods, the `AbstractCollection` class reduces the work needed to implement a new collection class. A programmer can extend the `AbstractCollection` class, provide the missing method bodies for the fundamental methods, and the other methods are automatically taken care of.

What Next?

The abstraction and reuse built into the collections framework doesn't stop there. Similar to the `Collection` interface, the `AbstractCollection` class has no direct implementations; in other words, no concrete classes extend it. Instead, it is extended by two other abstract classes: `AbstractSet` and `AbstractList`. Finally, these classes are extended by concrete classes. For example, the `ArrayList` class extends the `AbstractList` class. A look at the documentation for the `ArrayList` class shows that it also implements the library's `List` interface; which in turn, extends the `Collection` interface.

Whew!

We are not going to delve into any more details of the collections framework. Our intention was to provide an overview, and to give you a taste of the complexity involved in putting together a coherent group of generically reusable resources. The Java library designers did a good job. The Java Class Library provides a great deal of functionality, and compared to some of the program libraries of other popular languages, is relatively uncomplicated.

Remember that we can use the library without understanding all of its internal details. We were able to use the `ArrayList` class without knowing any details about the collections framework. We simply needed a logical view of the functionality of the `ArrayList` class and a description of its operation interfaces.

If you are interested in learning more about the Java Library Classes, you can study the extensive documentation available at the Sun Microsystems' site. In particular, the lists you studied in Chapter 3 are most closely related to the library's `Map` and `SortedMap` interfaces. Although the library's `List` interface provides a similar structure, it allows duplicate elements, does not support keys, and many of its implementations support additional features. The library's maps, on the other hand, support keys and do not allow elements with duplicate keys.

The stacks studied in this chapter are very closely related to the library's legacy `Stack` class.

In the next section we look at queues. They have no counterpart in the library, although they can be easily implemented on top of some of the library's classes.

4.4 Queues

Logical Level

A stack is a structure with the special property that elements are always added to and removed from the top. We know from experience that many collections of data elements operate in the reverse manner: Elements are added at one end and removed from the other. This data structure, called a FIFO (First In, First Out) queue, has many uses in computer programs. As before, we consider the FIFO queue data structure as an ADT at three levels: logical, implementation, and application. In the rest of this chapter, "queue" refers to a FIFO queue. (Another queue-type data structure, the priority queue, is discussed in Chapter 9.)

Queue A structure in which elements are added to the rear and removed from the front; a "first in, first out" (FIFO) structure

A queue (pronounced like the letter Q) is an ordered, homogeneous group of elements in which new elements are added at one end (the "rear") and elements are removed from the other end (the "front"). As an example of a queue, consider a line of students waiting to pay for their textbooks at a university bookstore (see Figure 4.7). In theory, if not in practice, each new student gets in line at

Figure 4.7 *A FIFO queue*

the rear. When the cashier is ready for a new customer, the student at the front of the line is served.

To add elements to a queue, we access the rear of the queue; to remove elements we access the front. The middle elements are logically inaccessible, even if, at the implementation level, we store the queue elements in a random-access structure such as an array. It is convenient to picture the queue as a linear structure with the front at one end and the rear at the other end. However, we must stress that the "ends" of the queue are abstractions; they may or may not correspond to any physical characteristics of the queue's implementation. The essential property of the queue is its FIFO access.

Like the stack, the queue is a holding structure for data that we use later. We put a data item onto the queue, and then when we need it, we remove it from the queue. If we want to change the value of an element, we should take that element off the queue, change its value, and then return it to the queue. We usually do not directly manipulate the values of items that are currently in the queue.

Operations on Queues

The bookstore example suggests two operations that can be applied to a queue. First, new elements can be added to the rear of the queue, an operation that we call `enqueue`. We can also take elements off the front of the queue, an operation that we call `dequeue`. Unlike the stack operations `push` and `pop`, the adding and removing operations on a queue do not have standard names. The `enqueue` operation is sometimes called enq, enque, add, or insert; `dequeue` is also called deq, deque, remove, or serve.

Another useful queue operation is checking whether the queue is empty. The `isEmpty` method returns `true` if the queue is empty and `false` otherwise. We can only `dequeue` when the queue is not empty. Theoretically, we can always `enqueue`, for in principle a queue is not limited in size. We know from our experience with stacks, however, that certain implementations (an array representation, for instance) require that we test whether the structure is full before we add another element. This real-world consideration applies to queues as well, so we define an `isFull` method. Figure 4.8 shows how a series of these operations would affect a queue.

The Queue ADT Specification

As we did with stacks, we define our queues to hold items of class `Object`. Also, as with stacks, we do not worry about manipulating copies of the objects; the references stored and returned from the `Queue` implementation are references to the original objects. However, unlike with stacks, with queues we revert to the programming by contract approach; the caller of `enqueue` and `dequeue` is responsible for checking for overflow and underflow before calling the methods. These methods do not raise any exceptions. Below we capture our queue specification in a Java *interface*.

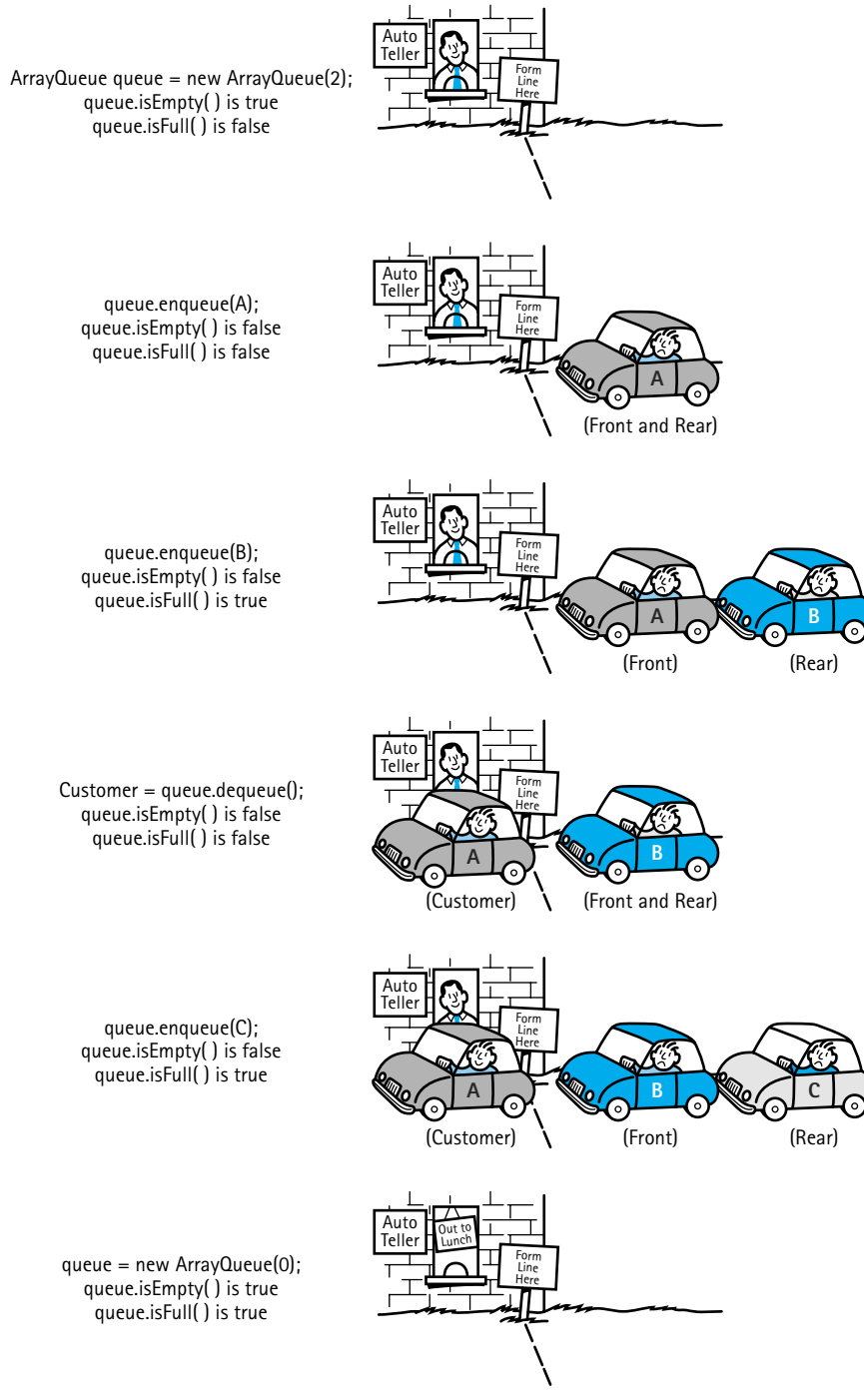


Figure 4.8 *The effects of queue operations*

```
//--------------------------------------------------------------------------
// QueueInterface.java           by Dale/Joyce/Weems                Chapter 4
//
// Interface for a class that implements a queue of Objects.
// A queue is a first-in, first-out structure
//--------------------------------------------------------------------------

package ch04.queues;

public interface QueueInterface

{
  public void enqueue(Object item);
  // Effect:         Adds item to the rear of this queue
  // Precondition:   This queue is not full
  // Postcondition:  item is at the rear of this queue

  public Object dequeue();
  // Effect:         Removes front element from this queue and returns it
  // Precondition:   This queue is not empty
  // Postconditions: Front element has been removed from this queue
  //                 Return value = (the removed element)

  public boolean isEmpty();
  // Effect:         Determines whether this queue is empty
  // Postcondition:  Return value = (this queue is empty)

  public boolean isFull();
  // Effect:         Determines whether this queue is full
  // Postcondition:  Return value = (queue is full)
}
```

Note that the `dequeue` operation has a side effect; it both removes and returns the element at the front of the queue. In our discussion of the Stack ADT, we argued that operations with side effects should be avoided. However, in this case, the "remove and return" approach is so common for queue implementations that we decided to stick with the classic approach.

Application Level

We discussed how stacks can be used by operating systems and compilers. Similarly, queues are often used for system programming purposes. For example, an operating system often maintains a FIFO queue of processes that are ready to execute or that are waiting for a particular event to occur. The programmer who creates the operating system can use a Queue ADT to implement this.

Computer systems must often provide a "holding area" for messages between two processes, two programs, or even two systems. This holding area is usually called a "buffer" and is often implemented as a FIFO queue. For example, if a large number of mail messages arrive at a mail server at about the same time, the messages are held in a buffer until the mail server can get around to processing them. It processes them in the order they arrived—in "first in, first out" order. (Some mail servers may be designed to handle the messages based on a priority system. In that case, the priority queue of Chapter 9 would be a more appropriate data structure.)

To demonstrate the use of queues, we look at a simpler problem: identifying palindromes. A *palindrome* is a string that reads the same forwards as backwards. While we are not sure of their general usefulness, identifying them provides us with a good example for the use of both queues and stacks. Besides, palindromes can be entertaining. Some famous palindromes are:

- A tribute to Teddy Roosevelt, who orchestrated the creation of the Panama Canal: "A man, a plan, a canal—Panama!"
- Allegedly muttered by Napoleon Bonaparte upon his exile to the island of Elba (although this is hard to believe since Napoleon mostly spoke French!): "Able was I ere, I saw Elba."
- Overheard in a Chinese restaurant: "Won ton? Not now!"
- And possibly the world's first palindrome: "Madam, I'm Adam."
- Followed immediately by one of the world's shorted palindromes: "Eve."

As you can see, the rules for what is a palindrome are somewhat lenient. Typically, we do not worry about punctuation, spaces, or matching the case of letters.

We again follow the input/output model we have established for our test drivers—the same model used for the balanced parentheses example in the section on stacks. An input file holds a separate string on each line. A corresponding output file is created, repeating each of the input lines and stating whether or not it is a palindrome. This program assumes that no input line is more than 180 characters in length. If an input line is longer than that, it is skipped. The names of the input and output files are passed to the program on the command line. Summary statistics are written to an output frame.

The program reads a line of input and checks to see how long it is. If it is too long, it moves on to the next line of input. Otherwise, it creates a new stack and a new queue, and it repeatedly pushes each letter from the input line onto the stack, and also enqueues it onto the queue. To simplify comparison later, the actual characters pushed and enqueued are the lowercase versions of the characters in the string. When all of the characters of the line have been processed, the program repeatedly pops a letter from the stack and dequeues a letter from the queue. As long as these letters match each other for the entire way through this process, we have a palindrome. Can you see why? Since the queue is a "first in, first out" structure, the letters are returned from the queue in the same order they appear in the string. But the letters taken from the stack, a "last in, first out" structure, are returned in the opposite order from the way they appear in the string. So, we are comparing the letters from the forward view of the string to the letters from the backward view of the string.

Now we are ready to write the main algorithm assuming an instance of a Stack ADT as defined by `StackInterface` and an instance of the Queue ADT as defined by `QueueInterface`. The basic flow of the algorithm is to continuously read and handle the input lines. Handling an input line means to treat it following the actions described in the previous paragraph. For each line we repeatedly compare the characters from the stack and the queue to each other until the structures become empty or we determine that the string is not a palindrome. To indicate that we have discovered that the string is not a palindrome we set the boolean variable `stillPalindrome` to `false`.

Main Algorithm

```
Initialize expression counts
Read first input line
while there are still lines to process
   Increment the total number of strings
   Echo print the current string to the output file
   if the string is too long
      Increment the number of overlong strings
      Write "String too long" to the output file
   else
      Process the current string
      if (!stillPalindrome)
         Increment the count of non palindromes
         Write "Not a palindrome" to the output file
      else
         Increment the count of palindromes
         Write "Is a palindrome" to the output file
Write summary information to the output frame
```

This top-level algorithm shows the basic flow of control and manipulation of the lines of strings. But the details of how it is determined (whether or not the current string is a palindrome) are hidden in the phrase "Process the current string". Here is a description of that algorithm:

Process the current string

```
Create a new stack
Create a new queue
For each character in the string
   if the character is a letter
      Change the character to lowercase
      Push the character onto the stack
      enqueue the character onto the queue
Set stillPalindrome to true
while (there are still more characters in the structures && the string can still be a palindrome)
   Pop character1 from the stack
   dequeue a character2 from the queue
   if (character1 != character2)
      Set stillPalindrome to false
```


The only part of the above algorithm that may require expansion before moving on to the coding stage is the determination of when "there are still more characters in the structures." This could be accomplished by using the `isEmpty` methods of the `stack` and `queue` classes. Another approach also works, however. We can track the number of letters we find in the current string. Remember, that it is only these letters that are placed in the data structures. Then we can use a *for* loop, based on the number of letters we found, to control how many times we remove and compare characters. We use this second approach.

A few details require further explanation; the corresponding lines in the code are highlighted. Two static methods of the `Character` class, `isLetter` and `toLowerCase` are used. Since these are static methods they are invoked through the name of the class, and not through an object of the class. The `isLetter` method returns a `boolean` value indicating whether the character passed to it is a letter or not, in other words, if it is an alphabetic character. It is used to let us skip over punctuation marks and spaces. The `toLowerCase` method returns the lowercase version of its character argument; if the argument is already lowercase, it simply returns the character unchanged. Its use ensures that all stored characters are lowercase; therefore, when we compare the characters, case is not an issue.

Just as we did with stacks, we use the `Character` class as a wrapper. Since our queue ADT handles objects only, we need to wrap the characters as `Character` objects, before passing them to the ADTs. This wrapping also allows us to use the `Character` class's `equals` method to compare the two characters.

The code for the program `Palindrome` is listed next. It uses our `ArrayStack` and assumes a class `ArrayQueue` implements the `QueueInterface` interface.

```
//--------------------------------------------------------------------------
// Palindrome.java             by Dale/Joyce/Weems                  Chapter 4
//
// Checks for palindromes
// Input file consists of a sequence of strings, one per line
// Output file contains, for each string:
//    Whether or not the string is a palindrome ...  blanks are ignored
// Input and output file names are supplied by user through command line
// parameters
// Output frame supplies summary statistics
//--------------------------------------------------------------------------

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.text.DecimalFormat;
import ch04.stacks.*;
import ch04.queues.*;

public class Palindrome
{
  public static void main(String[] args) throws IOException
  {
    final int maxStringSize = 180;   // Maximum size of an input line
    int numStrings = 0;              // Total number of strings processed
    int palindromes = 0;             // Number of palindromes found
    int nonPalindromes = 0;          // Number of nonpalindromes found
    int tooLong = 0;                 // Number of strings too long to process

    char ch;                         // Current input string character being
                                     //   processed
    int numLetters;                  // Number of letter characters in current
                                     //   string
    int charCount;                   // Number of characters checked so far

    Character fromStack;             // Current Char object popped from stack
    Character fromQueue;             // Current Char object dequeued from queue
    boolean stillPalindrome;         // True as long as the string might still
                                     // be a palindrome

    StackInterface stack;            // Holds nonblank string characters
    QueueInterface queue;            // Also holds nonblank string characters
```

```
String line = null;                // Input line
String dataFileName = args[0];     // Name of input file
String outFileName = args[1];      // Name of output file

BufferedReader dataFile = new BufferedReader(new FileReader(dataFileName));
PrintWriter outFile = new PrintWriter(new FileWriter(outFileName));
DecimalFormat fmt = new DecimalFormat("000");

outFile.println();             // Print a blank line
line = dataFile.readLine();    // Read the first input line

while(line!=null)              // while haven't read all of the input lines
{
  numStrings = numStrings + 1;
  outFile.println("String " + fmt.format(numStrings) + ": " + line);

  if (line.length() > maxStringSize)
  {
    tooLong = tooLong + 1;
    outFile.println("String too long - processing skipped");
  }
  else
  {
    // Check if line is a palindrome
    stack = new ArrayStack(maxStringSize);
    queue = new ArrayQueue(maxStringSize);
    numLetters = 0;

    for (int i = 0; i < line.length(); i++)
    {
      ch = line.charAt(i);
      if (Character.isLetter(ch))
      {
        numLetters = numLetters + 1;
        ch = Character.toLowerCase(ch);
        stack.push(new Character(ch));
        queue.enqueue(new Character(ch));
      }
    }

    stillPalindrome = true;
    charCount = 0;

    while (stillPalindrome && (charCount < numLetters))
```

```
    {
      fromStack = (Character)stack.top();
      stack.pop();
      fromQueue = (Character)queue.dequeue();
      if (!fromStack.equals(fromQueue))
        stillPalindrome = false;
      charCount++;
    }

    if (!stillPalindrome)
    {
      nonPalindromes = nonPalindromes + 1;
      outFile.println("            Not a palindrome ");
    }
    else
    {
      palindromes = palindromes + 1;
      outFile.println("            Is a palindrome.");
    }
  }
  outFile.println();
  line = dataFile.readLine();    // Set up processing of next line
}
dataFile.close();
outFile.close();

// Set up output frame
JFrame outputFrame = new JFrame();
outputFrame.setTitle("Palindromes");
outputFrame.setSize(300,200);
outputFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Instantiate content pane and information panel
Container contentPane = outputFrame.getContentPane();
JPanel infoPanel      = new JPanel();

// Set layout
infoPanel.setLayout(new GridLayout(5,1));

infoPanel.add(new JLabel("Total Number Of Strings "+ numStrings));
infoPanel.add(new JLabel("Number Of Strings too long for Processing " + tooLong));
infoPanel.add(new JLabel("Number Of Palindromes "+ palindromes));
infoPanel.add(new JLabel("Number Of Non Palindromes "+ nonPalindromes));
```

```
        infoPanel.add(new JLabel("Program completed. Close window to exit."));
        contentPane.add(infoPanel);

        // Show information.
        outputFrame.show();
    }
}
```

Figure 4.9 shows a sample input file, with an associated output file and information frame.

```
A man, a plan, a canal, Panama
amanaplanacanalpanama
This is not a palindrone!
aaaaaaaaaa
a
aaaaaaaaaaabaaaaaaaaaaaaaaaaaaaaaaaaa
aAaaaaaaaabaaaaaAaaa
This string is too long
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxx
xxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxx
bob
dan
Madam, I'm Adam.
eve
Eve
```

testP1.in

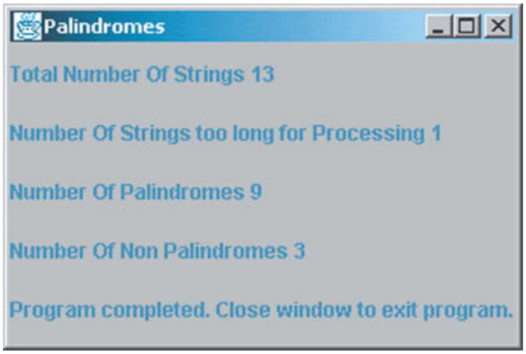
```
String 001:  A man, a plan, a canal, Panama
             Is a palindrome.

String 002:  amanaplanacanalpanama
             Is a palindrome.

String 003:  This is not a palindrone!
             Not a palindrome

String 004:  aaaaaaaaaa
             Is a palindrome.

String 005:  a
             Is a palindrome.

String 006:  aaaaaaaaaaabaaaaaaaaaaaaaaaaaaaaaaaaa
             Not a palindrome

String 007:  aAaaaaaaaabaaaaaAaaa
             Is a palindrome.
```

testP1.out beginning

Figure 4.9 *Sample run of the* `Palindrome` *program*

```
String 008: This string is too long
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxx
xxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxxxxxxxxx
String too long - processing skipped

String 009:  bob
             Is a palindrome.

String 010:  dan
             Not a palindrome

String 011:  Madam, I'm Adam.
             Is a palindrome.

String 012:  eve
             Is a palindrome.

String 013:  Eve
             Is a palindrome.
```

testP1.out continued



Information Frame

Command: java Palindrome testP1.in testP1.out

Figure 4.9 *(Continued)*

Implementation Level

Now that we've had the opportunity to be queue users, let's look at how a queue might be implemented in Java. As with a stack, the queue can be stored in a static array with its size fixed at compile time or in a dynamically allocated array (the `ArrayList` version) with its size determined at run time. We look at the static implementation here.

Definition of Queue Class

We implement our Queue ADT as a Java class. We call this class `ArrayQueue`, to differentiate it from another implementation approach we present in the next chapter.

What data members does our Queue ADT need? We need the queue items themselves; they are the elements of the underlying array. To facilitate the `isEmpty` and `isFull` operations we decide to use two instance variables, `capacity` and `numItems`. The `capacity` variable holds the maximum number of items the queue can hold, and the `numItems` variable holds the current number of items on the queue.

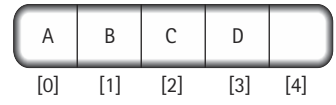
We still need some way of determining the front and rear items of the queue. There are several alternatives possible. This design decision is also interrelated with the approaches we use for implementing the queue operations.

We need to determine the relationship between the location of an item in the queue, and the location of the item in the underlying array.

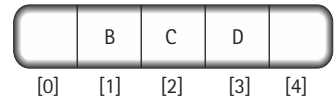
Fixed-Front Design Approach

In implementing the stack, we began by inserting an element into the first array position and then we let the top float with subsequent `push` and `pop` operations. The bottom of the stack, however, was fixed at the first slot in the array. Can we use a similar solution for a queue, keeping the front of the queue fixed in the first array slot and letting the rear move down as we add new elements?

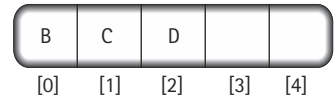
Let's see what happens after a few enqueues and dequeues if we insert the first element into the first array position, the second element into the second position, and so on. After four calls to `enqueue` with arguments '`A`', '`B`', '`C`', and '`D`', the queue would look like this:



Remember that the front of the queue is fixed at the first slot in the array, whereas the rear of the queue moves down with each enqueue. Now we dequeue the front element in the queue:



This operation deletes the element in the first array slot and leaves a hole. To keep the front of the queue fixed at the top of the array, we need to move every element in the queue up one slot:



Let's summarize the queue operations corresponding to this queue design. The `enqueue` operation would be the same as `push`. The `dequeue` operation would be more

complicated than `pop`, because all the remaining elements of the queue would have to be shifted up in the array to move the new front of the queue up to the first array slot.

Before we go any further, we want to stress that this design would work. It may not be the best design for a queue, but it could be successfully implemented. There are multiple functionally correct ways to implement the same abstract data type. One design may not be as good as another (because it uses more space in memory or takes longer to execute) and yet may still be correct.

Now let's evaluate this design. Its strengths are its simplicity and ease of coding; it is almost exactly as the stack implementation. Though the queue is accessed from both ends rather than just one (as in the stack), we just have to keep track of the rear, because the front is fixed. Only the dequeue operation is more complicated. What is the weakness of the design? It's the need to move all the elements up every time we remove an element from the queue, which increases the amount of work needed to dequeue.

How serious is this weakness? To make this judgment, we have to know something about how the queue is to be used. If this queue is used for storing large numbers of elements at one time, the processing required to move up all the elements after the front element has been removed makes this solution a poor one. On the other hand, if the queue generally contains only a few elements, all this data movement may not amount to much processing. Further, we need to consider whether performance—how fast the program executes—is of importance in the application that uses the queue. Thus, the complete evaluation of the design depends on the requirements of the client program.

In the real programming world, however, you don't always know the exact uses or complete requirements of client programs. For instance, you may be working on a very large project with a hundred other programmers. Other programmers may be writing the specific application programs for the project while you are producing some utility programs that are used by all the different applications. If you don't know the requirements of the various users of your queue operations, you must design general-purpose utilities. In this situation, the design described here is not the best one possible.

Floating-Front Design Approach

The need to move the elements in the array was created by our decision to keep the front of the queue fixed in the first array slot. If we keep track of the index of the front as well as the rear, we can let both ends of the queue float in the array.

Figure 4.10 shows how several `enqueue` and `dequeue` operations would affect the queue. The `enqueue` operations have the same effect as before; they add elements to subsequent slots in the array and increment the index of the rear indicator. The `dequeue` operation is simpler, however. Instead of moving elements up to the beginning of the array, it merely increments the front indicator to the next slot.

Letting the queue elements float in the array creates a new problem when the rear indicator gets to the end of the array. In our first design, this situation told us that the

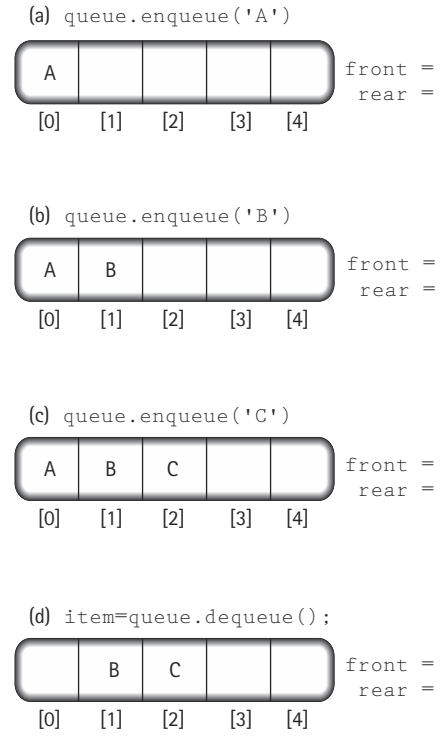


Figure 4.10 *The effect of enqueue and dequeue*

queue was full. Now, however, it is possible for the rear of the queue to reach the end of the (physical) array when the (logical) queue is not yet full (Figure 4.11a).

Because there may still be space available at the beginning of the array, the obvious solution is to let the queue elements "wrap around" the end of the array. In other words, the array can be treated as a circular structure in which the last slot is followed by the first slot (Figure 4.11b). To get the next position for the rear indicator, for instance, we can use an *if* statement:

```
if (rear == (capacity - 1))
  rear = 0;
else
  rear = rear + 1;
```

Another way to reset `rear` is to use the modulo (`%`) operator:

```
rear = (rear + 1) % capacity;
```

We use this floating-front design approach with the wrap around.

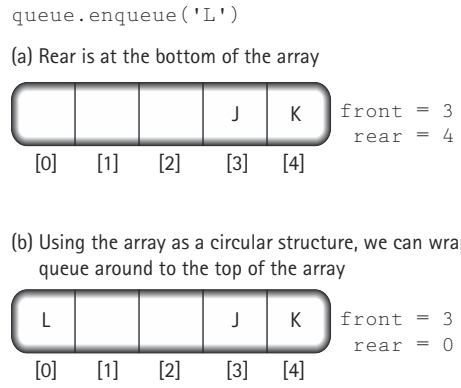


Figure 4.11 *Wrapping the queue elements around*

The Instance Variables and Constructors

From our analysis we see that we must add two instance variables to the class: `front` and `rear`. The beginning of the `ArrayQueue.java` file is:

```
//--------------------------------------------------------------------------
// ArrayQueue.java             by Dale/Joyce/Weems                  Chapter 4
//
// Implements QueueInterface using an array to hold the queue items
//--------------------------------------------------------------------------

package ch04.queues;

public class ArrayQueue implements QueueInterface
{
  private Object[] queue;          // Array that holds queue elements
  private int capacity;            // Size of the array (capacity of the queue)
  private int numItems  = 0;       // Number of items on the queue
  private int front = -1;          // Index of front of queue
  private int rear = 0;            // Index of rear of queue

  // Constructors
  public ArrayQueue()
  {
    queue = new Object[100];
    capacity = 100;
  }

  public ArrayQueue(int maxSize)
  {
    queue = new Object[maxSize];
    capacity = maxSize;
  }
```

As you can see, we have included the two standard constructors: one for which the client program specifies a maximum size for the queue, and one that defaults to a maximum size of 100 elements.

Definitions of Queue Operations

Given the discussion of the previous subsection, the implementation of our queue operations is straightforward. The `enqueue` method increments the `front` variable, "wrapping it around" if necessary, inserts the parameter element into the `front` location, and increments the `numItems` variable.

```
public void enqueue(Object item)
// Adds an element to the front of this queue
{
  front = (front + 1) % capacity;
  queue[front] = item;
  numItems = numItems + 1;
}
```


The `dequeue` method is essentially the reverse of this—it returns the element indicated by the `rear` variable, increments `rear`, also wrapping if necessary, and decrements `numItems`. Note that this method starts by making a copy of the reference to the object it eventually returns. It does this because during its next few steps, it removes the reference to the object from the array, and if it did not first make a copy, it would not be able to return the required information.

```
public Object dequeue()
// Removes an element from the rear of this queue
{
  Object toReturn = queue[rear];
  queue[rear] = null;
  rear = (rear + 1) % capacity;
  numItems = numItems - 1;
  return toReturn;
}
```

Note that `dequeue`, like the stack `pop` operation, actually sets the value of the array location associated with the removed element to `null`. As explained before, this allows the Java garbage collection process to work with up-to-date information.

The observer methods are very simple.

```
public boolean isEmpty()
// Checks if this queue is empty
{
  return (numItems == 0);
}

public boolean isFull()
// Checks if this queue is full
{
  return (numItems == capacity);
}
```

The entire `ArrayQueue.java` program is contained on the web site.

Test Plan

To make sure that you have tested all the necessary cases, make a test plan, listing all the queue operations and what tests are needed for each, as we did for stacks. (For example, to test the method `isEmpty`, you must call it at least twice, once when the queue is empty and once when it is not.)

You want to `enqueue` elements until the queue is full and then to call methods `isEmpty` and `isFull` to see whether they correctly judge the state of the queue. You can then `dequeue` all the elements in the queue, printing them out as you go, to make sure that they are correctly removed. At this point you can call the queue status methods again to see whether the empty condition is correctly detected. You also want to test out the "tricky" part of the array-based algorithm: You `enqueue` until the queue is full, `dequeue` an element, then `enqueue` again, forcing the operation to circle back to the beginning of the array.

Comparing Array Implementations

The circular array solution is not nearly as simple or intuitive as our first queue design. What did we gain by adding some amount of complexity to our design? By using a more efficient `dequeue` algorithm, we achieved better performance. To find out how much better, let's analyze the first design. Because the amount of work needed to move all the remaining elements is proportional to the number of elements, this version of `dequeue` is a O(N) operation. The second array-based queue design only requires `dequeue` to perform a few simple operations. The amount of work never exceeds some fixed constant, no matter how many elements are in the queue, so the algorithm is O(1).

All the other operations are O(1). No matter how many items are in the queue, they do (essentially) a constant amount of work.

Case Study

Postfix Expression Evaluator

Problem *Postfix notation*[2] is a notation for writing arithmetic expressions in which the operators appear after their operands. There are no precedence rules to learn, and parentheses are never needed. Because of this simplicity, some popular hand-held calculators use postfix notation to avoid the complications of the multiple parentheses required in nontrivial infix expressions (Figure 4.12); it is also used by compilers for generating nonambiguous expressions. You are to write a computer program that evaluates postfix expressions. In addition to evaluating the expressions, your program must display some statistics about the evaluation activity.

Discussion In elementary school you learned how to evaluate simple expressions that involve the basic binary operators: addition, subtraction, multiplication, and division. (These are called *binary operators* because they each operate on two operands.) It is easy to see how a child would solve the following problem:

$$2 + 5 = ?$$

Figure 4.12 *A calculator that evaluates postfix expressions*

[2] Postfix notation is also known as reverse Polish notation (RPN), so named after the Polish logician Jan Lukasiewicz (1875–1956) who developed it.

As expressions become more complicated, the pencil and paper solutions require a little more work. A number of tasks must be performed to solve the following problem:

$$(((13 - 1) / 2) * (3 + 5)) = ?$$

These expressions are written using a format known as *infix* notation. This same notation is used for writing arithmetic expressions in Java. The operator in an infix expression is written in between its operands. When an expression contains multiple operators such as the one shown here, we need to use a set of rules to determine which operation to carry out first. You learned in your mathematics classes that multiplication is done before addition. You learned Java's operator-precedence rules in your first Java programming course. In both situations, we use parentheses to override the normal ordering rules. It is easy to make a mistake writing or interpreting an infix expression containing multiple nested sets of parentheses.

Evaluating Postfix Expressions Postfix notation is another format for writing arithmetic expressions. In this notation, the operator is written after the two operands. Here are some simple postfix expressions and their results.

Postfix Expression	Result
4 5 +	9
9 3 /	3
17 8 −	9

The rules for evaluating postfix expressions with multiple operators are much simpler than those for evaluating infix expressions; simply evaluate the operations from left to right. Now, let's look at a postfix expression containing two operators.

$$6\ 2\ /\ 5\ +$$

We evaluate the expression by scanning from left to right. The first item, 6, is an operand, so we go on. The second item, 2, is also an operand, so again we continue. The third item is the division operator. We now apply this operator to the two previous operands. Which of the two saved operands is the divisor? The one we saw most recently. We divide 6 by 2 and substitute 3 back into the expression, replacing 6 2 /. Our expression now looks like this:

$$3\ 5\ +$$

We continue our scanning. The next item is an operand, 5, so we go on. The next (and last) item is the operator +. We apply this operator to the two previous operands, obtaining a result of 8.

Here's another example.

$$4\ 5\ +\ 7\ 2\ -\ *$$

If we scan from left to right, the first operator we encounter is +. Applying this to the two preceding operands, we obtain the expression

9 7 2 − *

The next operator we encounter is −, so we subtract 2 from 7, obtaining

9 5 *

Finally, we apply the last operator, *, to its two preceding operands and obtain our final answer, 45.

Here are some more examples of postfix expressions containing multiple operators and the results of evaluating them. See if you get the same results when you evaluate them.

Postfix Expression	Result
4 5 7 2 + − *	−16
3 4 + 2 * 7 /	2
5 7 + 6 2 − *	48
4 2 3 5 1 − + * + *	not enough operands
4 2 + 3 5 1 − * +	18

Our task is to write a program that evaluates postfix expressions entered interactively from the keyboard through a graphical user interface. In addition to computing and displaying the value of an expression, our program must display error messages when appropriate ("not enough operands," "too many operands," and "illegal symbol") and display statistics about the evaluation approach.

Before we specify our input and output exactly, let's look at the data structure and algorithm involved in the problem solution.

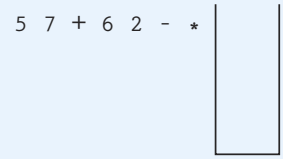
Postfix Expression Evaluation Algorithm As so often happens, our by-hand algorithm can be used as a guideline for our computer algorithm. From the previous discussion, we see that there are two basic items in a postfix expression: operands (numbers) and operators. We access items (an operand or an operator) one at a time from the interface. When the item we get is an operator, we apply it to the last two operands. Therefore, we must save previously scanned operands in a container object of some kind. A stack is the ideal place to store the previous operands, because the top item is always the most recent operand and the next item on the stack is always the second most recent operand—just the two operands required when we find an operator. The following algorithm uses a stack in this manner to evaluate a postfix expression:

EvaluateExpression

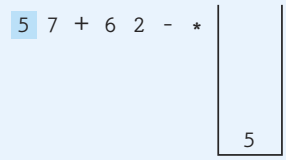
```
while more items exist
   Get an item
   if item is an operand
      stack.push(item)
   else
      operand2 = stack.top()
      stack.pop()
      operand1 = stack.top()
      stack.pop()
      Set result to operand1 item operand2
      stack.push(result)
result = stack.top()
stack.pop()
return result
```

Each iteration of the *while* loop processes one operator or one operand from the expression. When an operand is found, there is nothing to do with it (we haven't yet found the operator to apply to it), so we save it on the stack until later. When an operator is found, we get the two topmost operands from the stack, do the operation, and put the result back on the stack; the result may be an operand for a future operator.

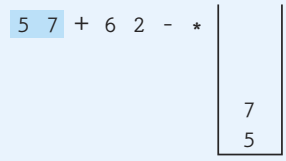
Let's trace this algorithm. Before we enter the loop, the input remaining to be processed and the stack look like this:



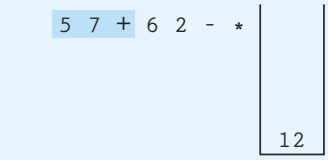
After one iteration of the loop, we have processed the first operand and pushed it onto the stack.



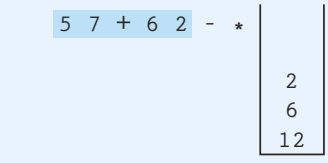
After the second iteration of the loop, the stack contains two operands.



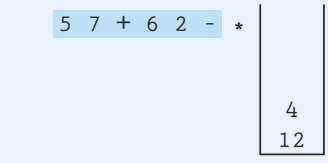
We encounter the + operator in the third iteration. We remove the two operands from the stack, perform the operation, and push the result onto the stack.



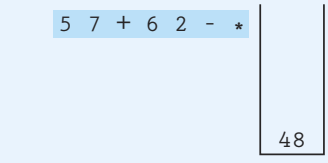
In the next two iterations of the loop, we push two operands onto the stack.



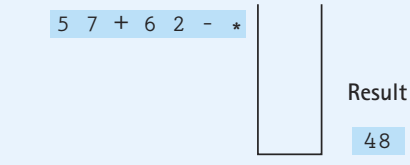
When we find the − operator, we remove the top two operands, subtract, and push the result onto the stack.



When we find the * operator, we remove the top two operands, multiply, and push the result onto the stack.



Now that we have processed all of the items on the input line, we exit the loop. We remove the result, 48, from the stack.



Of course, we have glossed over a few "minor" details, such as how we recognize an operator and how we know when we are finished. All the input values in the example were one-digit numbers. Clearly, this is too restrictive. We also need to handle invalid input. We discuss these challenges when we develop the individual module algorithms.

Brainstorming and Filtering The specifications of our problem are listed in Figure 4.13.

Specification: Program Postfix Evaluation

Function

The program evaluates postfix arithmetic expressions containing integers and the operators +, −, *, and /. In addition to displaying the result of the expression, the program displays some statistics about the evaluation phase.

Interface

The program presents a graphical user interface that includes a text box where the user can enter a postfix expression. When the user presses an Evalulate button, the current expression is evaluated and the result is displayed. Additionally, statistics about the evaluation phase are displayed. The interface includes a Clear button. When it is pressed, the text box and previous results and statistics, if any, are cleared.

Input

The input is a series of arithmetic expressions entered interactively from the keyboard into a text box in postfix notation. The expression is made up of operators (the characters '+', '−', '*', and '/') and integers (the operands). Operators and operands must be separated by at least one blank. Users can also press two buttons, Evaluate and Clear. And, of course, they can close the application.

Data

All numbers input, manipulated, and output by the program are integers.

Output

After the evaluation of each expression, the results are displayed on the interface:

"Result = value"

Also displayed are the following statistics related to the integers pushed onto the stack during the processing of the expression: count (number of pushes), minimum (the smallest value pushed), maximum (the largest value pushed), and average (the average value pushed). For example, if the expression is 2 6 * then the numbers pushed during processing would be 2, 6, and 12, and the statistics would be `count = 3`, `minimum = 2`, `maximum = 12`, and `average = 6`. Note that the average is rounded down.

Figure 4.13 *Specification for the postfix evaluator*

Error Processing

The program should recognize illegal postfix expressions, and instead of displaying a value when the Evaluate button is pressed, it should display error messages as follows:

Illegal symbol	If an expression contains a symbol that is not an integer or not one of "+", "-", "*", and "/"
Too many operands—stack overflow	If an expression requires more than 50 stack items
Too many operands—operands left over	If there is more than one operand left on the stack after the expression is processed; for example, the expression 5 6 7 +
Not enough operands—stack underflow	If there are not enough operands on the stack when it is time to perform an operation; for example, 5 6 7 + + +; and, for example, 5 + 5

Assumptions

1. The operations in expressions are valid at run time. This means that we do not try to divide by zero. Also, we do not try to generate numbers outside of the range of the primitive Java `int` type.
2. The stack used to process postfix expressions has a maximum size of 50.

Figure 4.13 *(Continued)*

A study of them provides the following list of nouns that appear to be possibilities for classes: postfix arithmetic expressions, operators, result, statistics about the evaluation phase, operands, the stack, and error messages. Let's look at each in turn.

- We already know that the "postfix arithmetic expressions" are entered into a text box; therefore, an expression should be represented by a string.
- This means we can probably represent "operators" as strings also. Another possibility is to hold the operators in an ADT that provides a "set" of characters. However, upon reflection, we realize that all we really have to do is recognize the operator characters, and the built-in string and character operations we already have at our disposal should be sufficient.
- The "result" of an evaluation is an interesting case. Where does the result come from? We could evaluate it in our main program, but we already know that we are using a graphical user interface—and following our standard approach, providing the interface is the primary role of the main program. Therefore, we propose the creation of a separate class `PostFixEvaluator` that provides a method that accepts a postfix expression as a string and returns the value of the expression. And we propose the creation of the main program, `PostFix`, that provides the graphical interface and uses the other classes to solve the problem.

- Considering "statistics about the evaluation phase" a little more carefully, we realize it is a perfect candidate for a separate class. Essentially, we have a set of numbers (the numbers that are pushed onto the stack) that we generate while processing the postfix expression. When we are finished evaluating the expression, we need to know some information about these numbers. We propose an `IntSetStats` class (statistics about a set of integers) to provide this service.
- The "operands" are integers.
- The "stack" is easy. We decide to use our `ArrayStack` class.
- The "error messages" we need to generate are all related to the evaluation of the postfix expression. Since we have proposed that the `PostFixEvaluator` class handles the postfix expression as a string and evaluates it, we realize that the errors are discovered within `PostFixEvaluator`. However, the error messages must be communicated through the graphical user interface. Therefore, to facilitate the communication of the error messages between `PostFixEvaluator` and the main program, we propose the creation of an exception class called `PostFixException`.

We leave it to you to create some CRC cards for the five classes we just identified (`PostFix`, `PostFixEvaluator`, `IntSetStats`, `ArrayStack`, and `PostFixException`) and to model some scenarios involving the use of the postfix expression evaluator.

We now move on to the design, implementation, and testing of the classes. Note that we assume the `ArrayStack` class has already been thoroughly tested. Also note that we can test the other three classes altogether, once they have all been created, by trying out a number of postfix expressions (both legal and illegal).

Evolving a Program

We present our case study in an idealized fashion. We make a general problem statement; discuss it; define formal specifications; identify classes; design and code the classes; and then test the system. In reality, however, an application like this would probably evolve gradually, with small unit tests performed along the way. Especially during design and coding, it is sometimes helpful to take smaller steps, and to evolve your program rather than trying to create the entire thing at once. For example, for this case study you could:

1. Build a prototype of the graphical interface (`PostFix`), just to see if the layout is OK. The prototype would just present the interface components—it would not support any processing.
2. Build a small part of `PostFixEvaluator` and see if you can pass it a string from the interface when the Evaluate button is pressed.
3. Next, see if you can pass back some information, any information, about the string, from `PostFixEvaluator` to `PostFix` and have it displayed on the interface. (For example, you could display the number of tokens in the string.)
4. Upgrade `PostFixEvaluator` so that it recognizes operands and transforms them into integers. Have it obtain an operand from the expression string, transform it into an integer, push the integer onto a stack, retrieve it, and pass it back to `PostFix` for display. Test this by entering a single operand expression into the text box and pressing the Evaluate button.

5. Upgrade `PostFixEvaluator` to recognize operators and process expressions that are more complicated. Test some legal expressions.
6. Add the "push statistics" portion and test again using legal expressions.
7. Add the error trapping and reporting portion and test using illegal expressions.

Devising a good program evolution plan is often the key to successful programming.

The IntSetStats Class We start with the simplest class. The key insight here is that we do not have to store the integers. The required statistics can be calculated whenever needed, if we simply keep track of the count, the sum, the minimum, and the maximum. We create a `register` method that is used by the client to pass in integers. Its code simply updates the four instance variables `count`, `min`, `max`, and `total`. We use a simple trick by initializing `max` to the smallest integer possible, and initializing `min` to the largest integer possible. This guarantees that the first integer "registered" with the set is recorded as being both the current maximum and the current minimum. The observers return strings, since the information is intended for display. The code is very straightforward:

```
//-------------------------------------------------------------------------
// IntSetStats.java          by Dale/Joyce/Weems                  Chapter 4
//
// Keeps track of some statistics about a set of integers passed to it
//-------------------------------------------------------------------------

package ch04.stacks;

public class IntSetStats
{
  private int count = 0;
  private int max = Integer.MIN_VALUE;
  private int min = Integer.MAX_VALUE;
  private int total = 0;

  public void register(int value)
  {
    count = count + 1;
    if (value > max)
      max = value;
    if (value < min)
      min = value;
    total = total + value;
  }
```

```
  public String getCount()
  {
    return Integer.toString(count);
  }

  public String getMax()
  {
    return Integer.toString(max);
  }

  public String getMin()
  {
    return Integer.toString(min);
  }

  public String getAverage()
  {
    if (count != 0)
      return Integer.toString((total / count));
    else
      return "none";
  }
}
```

The PostFixEvaluator Class The only purpose of this class is to provide a method that accepts a postfix expression as a string and returns the value of the expression. We do not need any objects of the class, just an `evaluate` method. Therefore, we implement the public `evaluate` method as a `static` method. This means it is not associated with objects of the class and is invoked through the class itself.

The `evaluate` method must take a postfix expression as a string parameter and return the value of the expression. We decide to have the method return the value of the expression as a string also; this makes for a balanced interface. Besides, we know that the client in this example needs to have the result as a string in order to display it anyway.

The code for the class is listed on page 315. It follows the basic postfix expression algorithm that we developed earlier. A few interesting features of the code are described next.

Push Statistics Recall that the client also needs to display information about the "push statistics." To provide this information to the client, we add a second parameter to the `evaluate` method, a parameter of type `IntSetStats`. The client instantiates the `IntSetStats` object and passes a reference to it, to the `evaluate` method. The `evaluate` method uses that reference to "register" the various integers that are pushed onto the stack during the processing of the expression. Since the client retains its own reference to the `IntSetStats` object, the client is able to "observe" the desired statistics when needed.

String Tokenizing The `evaluate` method needs to parse the expression string into operands and operators. The Java library provides a useful tool, the `StringTokenizer` class, to make this job easier. Since some readers may not be familiar with it, we briefly describe it in the String Tokenizing feature section below.

String Tokenizing

The `evaluate` method of the `PostFixEvaluator` class parses the user's input expression string into operands and operators. We use the Java library `StringTokenizer` class to facilitate this processing. The corresponding lines of code in the `PostFixEvaluator` class are emphasized.

A program instantiates a `StringTokenizer` object by passing its constructor a string, for example:

```
StringTokenizer tokenizer = new StringTokenizer(expression);
```

In this case, expression is a string, for example "15 8 * 13 +".

The `StringTokenizer` object `tokenize` can then be used to observe information about the tokens of the string, and to obtain the tokens themselves. A token is a substring that is delimited by special characters. A program can define its own special characters for its purposes, or use the default characters (typical "white space" characters such as blanks and tabs). In our program, we use the default delimiters. So for the sample string shown above, the tokenizer breaks the string into 5 tokens: "15", "8", "*", "13", and "+". A call to `tokenizer.countTokens` would return the value 5.

The `StringTokenizer` class includes a pair of methods (`hasMoreTokens` and `nextToken`) that can be used to iterate through all the tokens of a string. The `nextToken` method returns a string equal to the next token—so the first time it is called in our example it returns "15"; the second time it is called it returns "8"; etc. The `hasMoreTokens` method returns a boolean indicating whether `nextToken` still has more tokens to return. See the emphasized code in the `PostFixEvaluator` class for an example of how these methods are used to step through the tokens of the postfix expression.

Error Message Generation Look through the code for the lines that throw `PostFixException` exceptions. You should be able to see that we cover all of the error conditions required by the problem specification. As would be expected, the error messages directly related to the stack processing are all protected by *if*-statements that check to see if the stack is empty (not enough operands) or full (too many operands). The only other error trapping occurs when attempting to transform a token (already determined not to be an operator) into an integer. If the

```
value = Integer.parseInt(token);
```

statement raises an exception because the `token` cannot be transformed into an integer, we catch it and then throw our own customized `PostFixException` exception to the client program with the message "Illegal symbol".

```
//--------------------------------------------------------------------------
// PostFixEvaluator.java          by Dale/Joyce/Weems                Chapter 4
//
// Provides a postfix expression evaluation
//--------------------------------------------------------------------------

package ch04.postfix;

import ch04.stacks.*;
import java.util.StringTokenizer;

public class PostFixEvaluator
{
  public static String evaluate(String expression, IntSetStats pushInts)
  {
    ArrayStack stack = new ArrayStack(50);

    int value;
    Integer wrapValue;

    int operand1;
    int operand2;

    int result = 0;
    Integer wrapResult;

    String token;

    StringTokenizer tokenizer = new StringTokenizer(expression);

    while (tokenizer.hasMoreTokens())
    {
      token = tokenizer.nextToken();
      if (isNotOperator(token))
      {
        // Process operand
        try
        {
          value = Integer.parseInt(token);
          wrapValue = new Integer(value);
        }
```

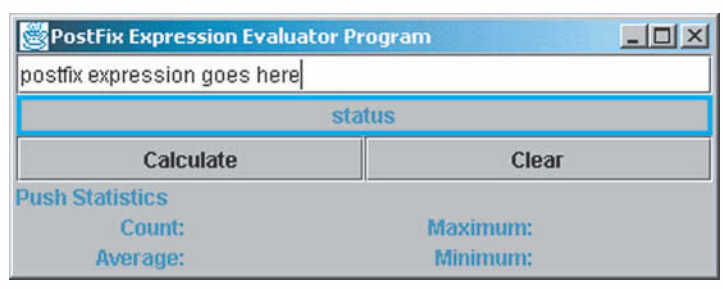
```
        catch (NumberFormatException badData)
        {
          throw new PostFixException("Illegal symbol: " +
                                     badData.getMessage());
        }

        if (stack.isFull())
          throw new PostFixException("Too many operands - stack overflow");
        stack.push(wrapValue);
        pushInts.register(value);
      }
      else
      {
        // Process operator
        if (stack.isEmpty())
          throw new PostFixException("Not enough operands - stack underflow");
        operand2 = ((Integer)stack.top()).intValue();
        stack.pop();

        if (stack.isEmpty())
          throw new PostFixException("Not enough operands - stack underflow");
        operand1 = ((Integer)stack.top()).intValue();
        stack.pop();

        if (token.equals("/"))
          result = operand1 / operand2;
        else
        if(token.equals("*"))
          result = operand1 * operand2;
        else
        if(token.equals("+"))
          result = operand1 + operand2;
        else
        if(token.equals("-"))
          result = operand1 - operand2;

        wrapResult = new Integer(result);
        stack.push(wrapResult);
        pushInts.register(result);
      }
    }

    if (stack.isEmpty())
      throw new PostFixException("Not enough operands - stack underflow");
    result = ((Integer)stack.top()).intValue();
    stack.pop();
```

```
    // Stack should now be empty
    if (!stack.isEmpty())
      throw new PostFixException("Too many operands - operands left over");

    return Integer.toString(result);
  }

  private static boolean isNotOperator(String testThis)
  {
    if(testThis.equals("/") || testThis.equals("*") ||
       testThis.equals("-") || testThis.equals("+"))
      return false;
    else
      return true;
  }
}
```

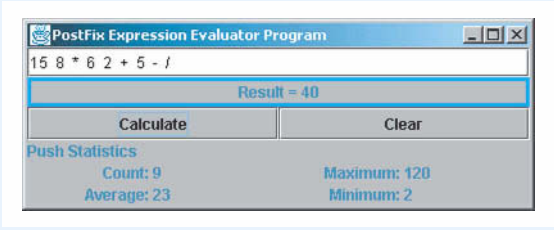
The PostFix Class This class is the main driver for our application. It consists mostly of user interface-related statements.

The main processing defined in this class takes place in the `ActionPerformed` method of the `ActionHandler` class when the event `getActionCommand` returns a string indicating that the Evaluate button has been pressed. In this case, the program instantiates a new `IntSetStats` object `pushInts`, and passes it and the current postfix expression (from the text field) to the `PostFixEvaluator`'s `evaluate` method. It obtains the `result` of the expression directly from the method as a return value. It obtains the required "push statistics" by invoking the observers of `pushInts`. In the case that an exception is caught, `result` is set to the exception's message. In either case, `result` is displayed on the interface.

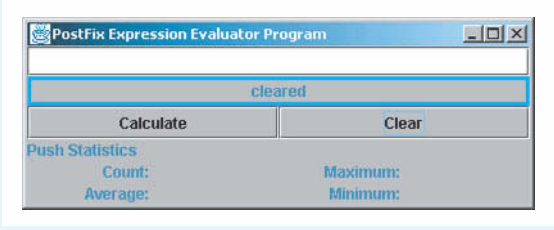
The rest of the code is concerned with displaying the interface. In this program we use a new approach to laying out the interface, which is described in the Java Input/Output III feature section below. Here are a few screenshots from the running program. The first shows the interface as originally presented to the user:



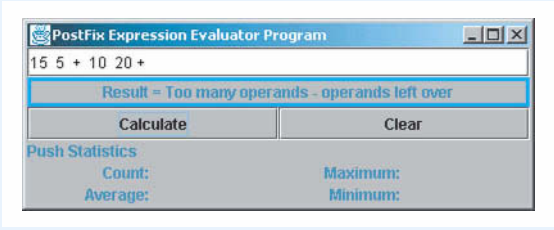
Here's the result of a successful evaluation:



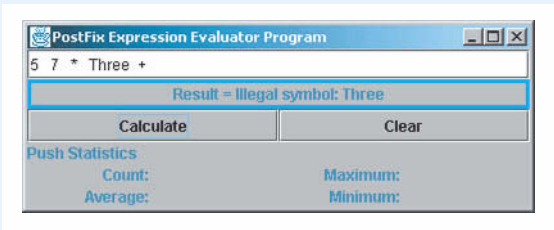
Next, the Clear button is pushed:



Here's what happens when the user enters an expression with too many operands:



And finally, what happens when an illegal operand is used:



As stated above, most of the code of `PostFix` deals with the user interface. To help you sort things out, we have emphasized the code related to working with the other classes created for this application: `PostFixEvaluator`, `PostFixException`, and `IntSetStats`. Note that

the main driver does not directly use `ArrayStack`—it is used strictly by the `PostFixEvaluator` class when evaluating an expression.

```
//--------------------------------------------------------------------------
// PostFix.java               by Dale/Joyce/Weems                  Chapter 4
//
// Evaluates postfix expressions
//--------------------------------------------------------------------------

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
import java.io.*;
import ch04.postfix.*;

public class PostFix
{
  // Text field
  private static JTextField expressionText;  // Text field for postfix
                                             // expression

  // Status Label
  private static JLabel statusLabel;         // Label for status/result info

  // push statistics labels
  private static JLabel countValue;
  private static JLabel minimumValue;
  private static JLabel maximumValue;
  private static JLabel averageValue;

  // To track "push" statistics
  private static IntSetStats pushInts;

  // Define a button listener
  private static class ActionHandler implements ActionListener
  {
    public void actionPerformed(ActionEvent event)
    // Listener for the button events
    {
      if (event.getActionCommand().equals("Evaluate"))
      { // Handles Evaluate event
        String result;
        try
```

```
        {
          pushInts = new IntSetStats();
          result = PostFixEvaluator.evaluate(expressionText.getText(),
                                             pushInts);
          countValue.setText(pushInts.getCount());
          minimumValue.setText(pushInts.getMin());
          maximumValue.setText(pushInts.getMax());
          averageValue.setText(pushInts.getAverage());
        }
        catch (PostFixException error)
        {
          result = error.getMessage();
        }
        statusLabel.setText("Result = " + result);
      }
      else
      if (event.getActionCommand().equals("Clear"))
      { // Handles Clear event
        statusLabel.setText("cleared");
        countValue.setText("");
        minimumValue.setText("");
        maximumValue.setText("");
        averageValue.setText("");
        expressionText.setText("");
      }
    }
  }

  public static void main(String args[]) throws IOException
  {
    // Declare/instantiate/initialize display frame
    JFrame displayFrame = new JFrame();
    displayFrame.setTitle("PostFix Expression Evaluator Program");
    displayFrame.setSize(400,150);
    displayFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Text box for expression
    expressionText = new JTextField("postfix expression goes here", 60);

    // Status/Result label
    statusLabel = new JLabel("status", JLabel.CENTER);
    statusLabel.setBorder(new LineBorder(Color.red,3));

    // Labels for "push" statistics output
    JLabel headerLabel   = new JLabel("Push Statistics", JLabel.LEFT);
```

```
JLabel blankLabel   = new JLabel("");
JLabel blankLabel2  = new JLabel("");
JLabel blankLabel3  = new JLabel("");
JLabel countLabel   = new JLabel("Count: ", JLabel.RIGHT);
JLabel minimumLabel = new JLabel("Minimum: ", JLabel.RIGHT);
JLabel maximumLabel = new JLabel("Maximum: ", JLabel.RIGHT);
JLabel averageLabel = new JLabel("Average: ", JLabel.RIGHT);
countValue   = new JLabel("", JLabel.LEFT);
minimumValue = new JLabel("", JLabel.LEFT);
maximumValue = new JLabel("", JLabel.LEFT);
averageValue = new JLabel("", JLabel.LEFT);

// Evaluate and clear buttons
JButton evaluate   = new JButton("Evaluate");
JButton clear       = new JButton("Clear");

// Button event listener
ActionHandler action = new ActionHandler();

// Register button listeners
evaluate.addActionListener(action);
clear.addActionListener(action);

// Instantiate content pane and information panels
Container contentPane = displayFrame.getContentPane();
JPanel expressionPanel = new JPanel();
JPanel buttonPanel = new JPanel();
JPanel labelPanel = new JPanel();

// Initialize expression panel
expressionPanel.setLayout(new GridLayout(2,1));
expressionPanel.add(expressionText);
expressionPanel.add(statusLabel);

// Initialize button panel
buttonPanel.setLayout(new GridLayout(1,2));
buttonPanel.add(evaluate);
buttonPanel.add(clear);

// Initialize label panel
labelPanel.setLayout(new GridLayout(3,4));
labelPanel.add(headerLabel);
labelPanel.add(blankLabel);
labelPanel.add(blankLabel2);
labelPanel.add(blankLabel3);
labelPanel.add(countLabel);
```

```
        labelPanel.add(countValue);
        labelPanel.add(maximumLabel);
        labelPanel.add(maximumValue);
        labelPanel.add(averageLabel);
        labelPanel.add(averageValue);
        labelPanel.add(minimumLabel);
        labelPanel.add(minimumValue);

        // Set up and show the frame
        contentPane.add(expressionPanel, "North");
        contentPane.add(buttonPanel, "Center");
        contentPane.add(labelPanel, "South");

        displayFrame.show();
    }
}
```

Testing Postfix Evaluator As mentioned before, we can test the classes created for our case study all together. We run the Postfix Evaluator program and enter a sequence of varied postfix expressions. We should test expressions that contain only additions, subtractions, multiplications, and divisions, and expressions that contain a mixture of operations. We should test expressions where the operators all come last, and other expressions where the operators are intermingled with the operands. Of course, we must evaluate all test expressions "by hand" in order to verify the correctness of the program's results. Finally, we must test that illegal expressions are correctly handled, as defined in the specifications. This includes a test of stack overflow.

Java Input/Output III

The input/output approaches used for the Postfix Evaluator case study build on the approaches used in the `IncDate` test driver of Chapter 1 and the Real Estate case study of Chapter 3. Remember that the Java I/O tools are extremely robust, and the approaches used in this text are not the only approaches possible.

Nested Containers

You have probably realized that the layout of the interface for the Postfix Evaluator is slightly more complicated than that of the Real Estate program of Chapter 3. Although its true that the

Real Estate program had more labels, text fields, and buttons, its layout was a single Grid layout, defined by the statement

```
infoPanel.setLayout(new GridLayout(10,2));
```

For the Real Estate program, the layout manager presented the components in a 10-by-2 grid (10 rows, 2 columns) of identically sized components.

For the PostFix Evaluator we do not want equal-sized components. At the top of our frame we want two components that extend the width of the frame, one a text field for the user to enter postfix expressions and the other a label used to report results and error messages. In the middle of our frame we want two buttons, each spanning half the frame. And at the bottom of our frame we want a 3-by-4 grid of labels for the push statistics. (We need four columns because we display two statistic name-value pairs across a row.) How can we organize the different areas of our interface with different layouts? By using nested containers.

First we organize each of our identified sub-areas into their own containers (panels), each following their own layout definition. We call these containers the `expressionPanel`, the `buttonPanel`, and the `labelPanel`. They are instantiated with the following code:

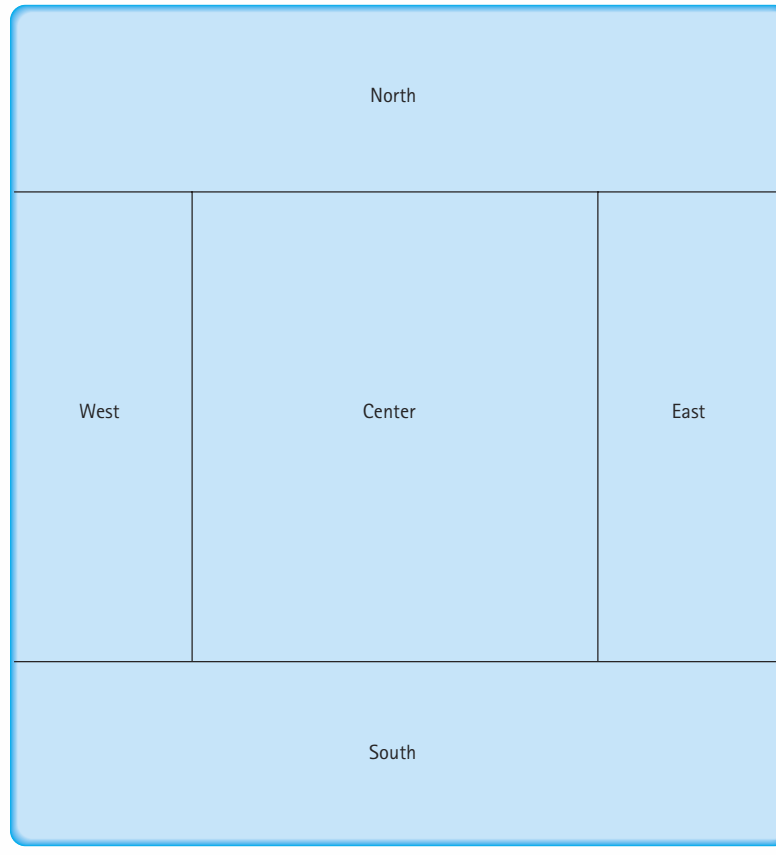
```
JPanel expressionPanel = new JPanel();
JPanel buttonPanel = new JPanel();
JPanel labelPanel = new JPanel();
```

A quick study of our three subareas reveals that they can each be organized with the grid layout approach we used in the Real Estate program. In the "top" area, the components span the frame—so they should be in a single column. In the "middle" area, the two buttons require a single row, but two columns. And in the "bottom" area, the labels require a 3-by-4 grid, as explained above. Here is the code that defines the layouts used for each panel:

```
expressionPanel.setLayout(new GridLayout(2,1));
buttonPanel.setLayout(new GridLayout(1,2));
labelPanel.setLayout(new GridLayout(3,4));
```

Once the subcontainers are defined, we add them to the "overall" container. In this case, we can add them directly to the content pane. Recall that the content pane is the part of the display frame where we add content for display. The content pane is itself a Java Swing container. We can define its layout, just as we define the layout of any container. We could define it as a 3-row-by-1-column grid layout to achieve our desired results. However, in this case, we are able to use its default layout, the Border layout, which gives a slightly better appearance to our interface.

The Border layout divides a container into five areas as follows:



With the Border layout, the programmer can choose which area to add specific components. The layout manager keeps the component in that part of the interface. If the user changes the frame size vertically, the West, Center, and East areas change in size proportionally; if the user changes the frame size horizontally, the North, Center, and South areas are adjusted.

We complete our interface by adding our three frames to the appropriate area of the content pane, and then displaying the frame:

```
contentPane.add(expressionPanel, "North");
contentPane.add(buttonPanel, "Center");
contentPane.add(labelPanel, "South");

displayFrame.show();
```

Nested Grid and Border layouts provide a wide range of interface design possibilities. These approaches are sufficient for the rest of the case studies developed in this text. For more sophisticated layout management approaches, the reader is encouraged to study the other Java layout managers, in particular the Grid Bag layout approach.

Summary

We have defined a stack at the logical level as an abstract data type, discussed its use in an application, and presented two implementations, each encapsulated in a class. Though our logical picture of a stack is a linear collection of data elements with the newest element (the top) at one end and the oldest element at the other end, the physical representation of the stack class does not have to recreate our mental image. The implementation of the stack class must support the last in, first out (LIFO) property; how this property is supported, however, is another matter. For instance, the `push` operation could "time stamp" the stack elements and put them into an array in any order. To `pop`, we would have to search the array, looking for the newest time stamp. This representation is very different from the stack implementations we developed in this chapter, but to the user of the stack class they are all functionally equivalent. The implementation is transparent to the program that uses the stack because the stack is encapsulated by the operations in the class that surrounds it.

We also examined the definition and operations of a queue. We discussed some of the design considerations encountered when an array is used to contain the elements of a queue. Though the array itself is a random-access structure, our logical view of the queue as a structure limits us to accessing only the elements in the front and rear positions of the queue stored in the array.

There usually is more than one functionally correct design for the same data structure. When multiple correct solutions exist, the requirements and specifications of the problem may determine which solution is the best design.

In the design of data structures and algorithms, you find that there are often tradeoffs. A more complex algorithm may result in more efficient execution; a solution that takes longer to execute may save memory space. A simple algorithm that is fast enough is usually better than a faster, more difficult algorithm. As always, we must base our design decisions on what we know about the problem's requirements.

Summary of Classes and Support Files

The classes and files are listed in the order in which they appear in the text. Inner classes are not included. The package a class belongs to, if any, is listed in parentheses under Notes. The class and support files are available on our web site. They can be found in the `ch04` subdirectory of the `bookFiles` directory.

Classes, Interfaces, and Support Files Defined in Chapter 4

File	1st Ref.	Notes
`ListInterface.java`	page 251	(`ch04.genericLists`) This interface specifies our List ADT
`List.java`	page 253	(`ch04.genericLists`) This abstract class developed in Chapter 3 now "implements" the `ListInterface`
`StackUnderflowException.java`	page 257	(`ch04.stacks`) Used by the stack classes developed in this chapter
`StackOverflowException.java`	page 258	(`ch04.stacks`) Used by the stack classes developed in this chapter
`StackInterface.java`	page 263	(`ch04.stacks`) This interface specifies our Stack ADT
`Balanced.java`	page 268	Uses a stack to determine whether a string of characters contains balanced parentheses
`ArrayStack.java`	page 274	(`ch04.stacks`) Implements `StackInterface` using an array
`ArrayListStack.java`	page 278	(`ch04.stacks`) Implements `StackInterface` using an `ArrayList`
`QueueInterface.java`	page 289	(`ch04.queues`) This interface specifies our Queue ADT
`Palindrome.java`	page 293	Uses a queue to determine whether a string of characters is a palindrome
`ArrayQueue.java`	page 301	(`ch04.queues`) Implements `QueueInterface` using an array
`IntSetStats.java`	page 312	(`ch04.postfix`) Used by the Postfix Expression Evaluator to maintain push statistics
`PostFixEvaluator.java`	page 315	(`ch04.postfix`) Used by the Postfix Expression Evaluator to evaluate expressions
`PostFix.java`	page 319	The Postfix Expression Evaluator program for the case study
`PostFixException.java`		(`ch04.postfix`) Used by the Postfix classes developed in this chapter
`test1.in`	page 273	Test data for the `Balanced` program
`test1.out`	page 273	Test output from the `Balanced` program
`testP1.in`	page 296	Test data for the `Palindrome` program
`testP1.out`	page 296	Test output from the `Palindrome` program

Below is a list of the Java Library Classes that were used in this chapter for the first time in the textbook. The classes are listed in the order in which they are first used. We list only classes used in our programs, not classes just mentioned in the text. Note that in some classes the methods listed might not be defined directly in the class; they might be defined in one of its superclasses. With the methods we also list constructors, if appropriate. For more information about the library classes and methods, the reader can check Sun's Java documentation.

Library Classes Used in Chapter 4 for the First Time:

Class Name	Package	Overview	Methods Used	Where Used
`DecimalFormat`	`text`	Allows us to format numbers	`DecimalFormat`, `format`	`Balanced`
`StringTokenizer`	`util`	Makes it easy to break a string into substrings of tokens	`StringTokenizer`, `hasMoreTokens`, `nextToken`	`PostFixEvaluator`

Exercises

4.1 Formal ADT Specifications

1. Describe the benefits of using the Java interface construct to specify ADTs.
2. What happens if a Java interface specifies a particular method signature, and a class that implements the interface provides a different signature for that method? For example, suppose interface `SampleInterface` is defined as:

```
public interface SampleInterface
{
  public int sampleMethod();
}
```

and the class `SampleClass` is

```
public class SampleClass implements SampleInterface
{
  public boolean sampleMethod()
  {
    return true;
  }
}
```

3. True or False? Explain your answers.
   a. You can define constructors for a Java interface.
   b. Classes implement interfaces.

c. Classes extend interfaces.

d. A class that implements an interface can include methods that are not required by the interface.

e. A class that implements an interface can leave out methods that are required by an interface.

f. You can instantiate objects of an interface.

g. An interface definition can include concrete methods.

4.2 Stacks

4. Indicate whether a stack would be a suitable data structure for each of the following applications.

   a. A program to evaluate arithmetic expressions according to a specific order of operators

   b. A bank simulation of its teller lines to see how waiting times would be affected by adding another teller

   c. A program to receive data that are to be saved and processed in reverse order

   d. An address book to be built and maintained

   e. A word processor to have a PF key that causes the preceding command to be redisplayed. Every time the PF key is pressed, the program is to show the command that preceded the one currently displayed

   f. A dictionary of words used by a spelling checker to be built and maintained

   g. A program to keep track of patients as they check into a medical clinic, assigning patients to doctors on a first-come, first-served basis

   h. A data structure used to keep track of the return addresses for nested functions while a program is running

5. Show what is written by the following segments of code, given that `item1`, `item2`, and `item3` are `int` variables, and `stack` is an object of the class `ArrayStack`. Assume that you can store and retrieve variables of type `int` on `stack`.

   a.
```
item1 = 1;
item2 = 0;
item3 = 4;
stack.push(item2);
stack.push(item1);
stack.push(item1 + item3);
item2 = stack.top();
stack.push (item3*item3);
stack.push(item2);
stack.push(3);
item1 = stack.top();
stack.pop();
System.out.println(item1 + " " + item2 + " " + item3);
```

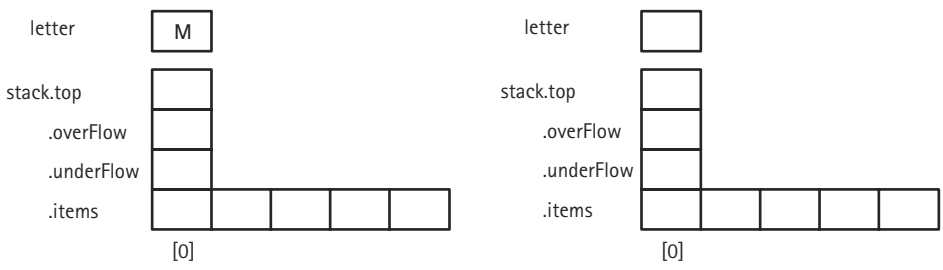
```
while (!stack.isEmpty())
{
  item1 = stack.top();
  stack.pop();
  System.out.println(item1);
}
```

b.

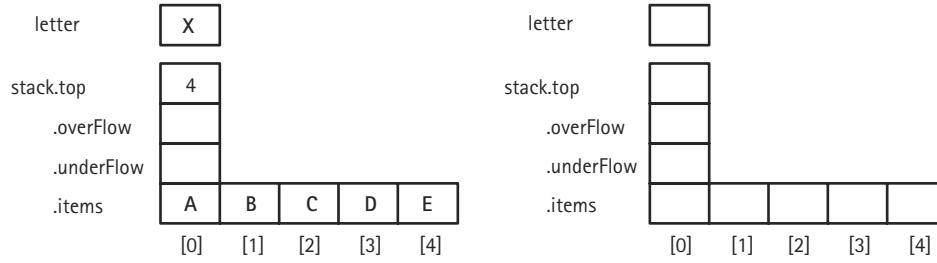
```
item1 = 4;
item3 = 0;
item2 = item1 + 1;
stack.push(item2);
stack.push(item2 + 1);
stack.push(item1);
item2 = stack.top();
stack.pop();
item1 = item2 + 1;
stack.push(item1);
stack.push(item3);
while (!stack.isEmpty())
{
  item3 = stack.top();
  stack.pop();
  System.out.println(item1);
}
System.out.println(item1 + " " + item2 + " " + item3);
```

6. A stack called `stack` is implemented as an object of a class that defines an array of items, an instance variable indicating the index of the last item put on the stack (`top`), and two `boolean` instance variables, `underFlow` and `overFlow`, that are set appropriately after each stack modification, to indicate whether the operation caused an overflow or underflow of the stack. The stack items are characters and `MAX_ITEM` is 5. For each part of the exercise, the left side of the figure represents the state of the `stack` before the specified operation. Show the result of the operation on the `stack` on the right side of the figure. Use '`T`' or '`F`' for true or false in the boolean instance variables.

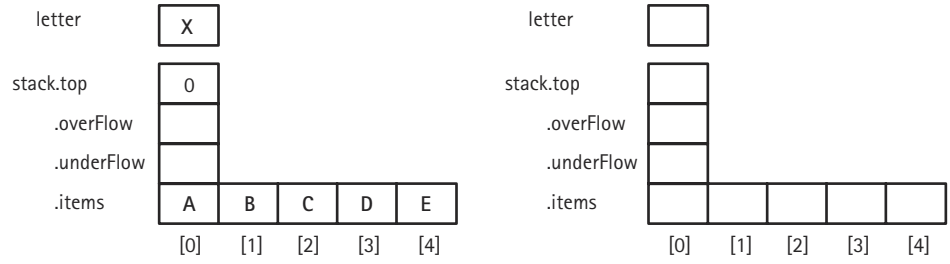
a. `stack.push(letter);`



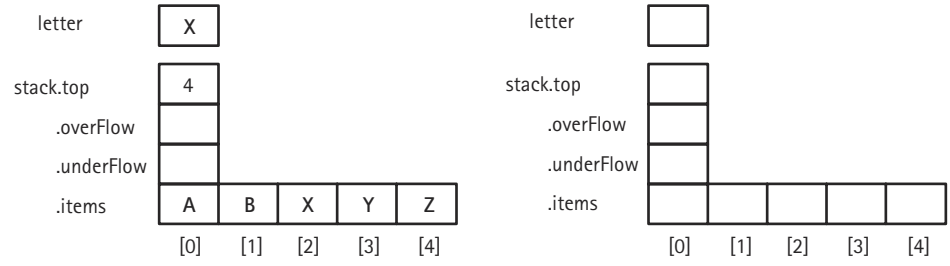
b. `stack.push(letter);`



c.
```
letter=stack.top();
stack.pop()
```



d.
```
letter=stack.top();
stack.pop()
```



7. Write a segment of code to perform each of the following operations. Assume `myStack` is an object of the class `ArrayStack`. You may call any of the public methods of `ArrayStack`. You may declare additional stack objects.

- **a.** Set `secondElement` to the second element in `myStack`, leaving `myStack` without its original top two elements.
- **b.** Set `bottom` equal to the bottom element in `myStack`, leaving `myStack` empty.
- **c.** Set `bottom` equal to the bottom element in `myStack`, leaving `myStack` unchanged.
- **d.** Make a copy of `myStack`, leaving `myStack` unchanged.

8. Two stacks of positive integers are needed, both containing integers with values less than or equal to 1000. One stack contains even integers; the other contains odd integers. The total number of elements in the combined stacks is never more than 200 at any time, but we cannot predict how many are in each stack. (All the elements could be in one stack, they could be evenly divided, both stacks could be empty, and so on.) Can you think of a way to implement both stacks in one array?
   a. Draw a diagram of how the stacks might look.
   b. Write the definitions for such a double-stack structure.
   c. Implement the push operation; it should store the new item into the correct stack according to its value (even or odd).
9. In each plastic container of Pez candy, the colors are stored in random order. Your little brother Phil only likes the yellow ones, so he painstakingly takes out all the candies, one by one, eats the yellow ones, and keeps the others in order, so that he can return them to the container in exactly the same order as before—minus the yellow candies, of course. Write the algorithm to simulate this process. (You may use any of the stack operations defined in the Stack ADT, but may not assume any knowledge of how the stack is implemented.)
10. In compiler construction, we need an inspector member for our stack so that we can examine stack elements based on their location in the stack (the top of the stack is considered location 1, the second element from the top is location 2, etc.) This is sometimes called (colloquially) a "glass stack" or (more formally) a "traversable stack." The definition of the stack is exactly as we specify in this chapter, except we add a public method named `inspector` that accepts a parameter of type `int` indicating the location we want to examine, and that returns the appropriate object.
    a. Write pre- and postconditions for `inspector`. Throw an exception if index is out of range.
    b. Code this method and test it thoroughly.
11. In compiler construction, we need to be able to pop more than one element at a time, discarding the items popped. To do this, we provide an `int` parameter `count` for the `pop` method and change the behavior to remove the top `count` items from the stack.
    a. Write this operation as client code, using operations from `StackInterface`.
    b. Write this operation as a new method of the `ArrayStack` class.
    c. Write this operation as a new method of the `ArrayListStack` class.
12. The following code segment is a count-controlled loop going from 1 through 5. At each iteration, the loop counter is either printed or put on a stack depending on the `boolean` result returned by the method `random`. (Assume that `random` randomly returns either a `true` or a `false`.) At the end of the loop, the items on the stack are removed and printed. Because of the logical properties of a stack, this code segment cannot print certain sequences of the values of the loop counter. You are given an output and asked to determine whether the code segment could generate the output. Assume that the stack is implemented as an `ArrayStack`.

```
for (count = 1; count <= 5; count++)
{
  if (random())
    System.out.println(count);
  else
    stack.push(count);
while (!stack.isEmpty())
{
  number = stack.top();
  stack.pop();
  System.out.println(number);
}
```

a. The following output is possible: 1 3 5 2 4

   i) True ii) False iii) Not enough information

b. The following output is possible: 1 3 5 4 2

   i) True ii) False iii) Not enough information

c. How would your answers to parts a and b change if you assume the stack is implemented as an `ArrayListStack`?

13. Describe the benefits and drawbacks of implementing a "container" ADT

    a. by copy.

    b. by reference.

14. Suppose you are "desk tracing" a program. You see that the program inserts a student record for a student with the ID of 55555 into a list. Later in the program the list is observed with an `isThere` method, to see if a student with the ID 55555 is on the list. Since no list methods have been invoked since the previous insert statement, you expect this `isThere` call to return `true`, but you know from a print-out of results that it returns `false`. What might explain this strange situation? What should you look for in the implementation of the list ADT and in the client program to verify your hypothesis?

4.3 The Java Collections Framework

(Note: The questions in this section may require "outside" research.)

15. Describe the major differences between the Java Library's `Vector` and `ArrayList` classes.

16. Explain how the iterators in the Java Collections Framework are used.

17. What is the defining feature of the Java Library `Set` class?

18. Which classes of the Java Library implement the `Collection` interface?

4.4 Queues

19. Show what is written by the following segments of code, given that `item1`, `item2`, and `item3` are `int` variables, and `queue` is an object of the class `ArrayQueue`. Assume that you can store and retrieve variables of type `int` in `queue`.

a.
```
item1 = 1;
item2 = 0;
item3 = 4
queue.enqueue(item2);
queue.enqueue(item1);
queue.enqueue(item1 + item3);
item2 = queue.dequeue()
queue.enqueue(item3*item3);
queue.enqueue(item2);
queue.enqueue(3);
item1 = queue.dequeue();
System.out.println(item1 + " " + item2 + " " + item3);
while (!queue.isEmpty())
{
  item1 = queue.dequeue();
  System.out.println(item1);
}
```

b.
```
item1 = 4;
item3 = 0;
item2 = item1 + 1;
queue.enqueue(item2);
queue.enqueue(item2 + 1);
queue.enqueue(item1);
item2 = queue.dequeue();
item1 = item2 + 1;
queue.enqueue(item1);
queue.enqueue(item3);
while (!queue.IsEmpty())
{
  item1 = queue.dequeue();
  System.out.println(item1);
}
System.out.println(item1 + " " + item2 + " " + item3);
```

20. The specifications for the Queue ADT have been changed. The class representing the queue must check for overflow and underflow and throw an exception. Rewrite the `QueueInterface` interface incorporating this change.

21. Write a segment of code to perform each of the following operations. Assume `myQueue` is an object of the class `ArrayQueue`. You may call any of the public methods of `ArrayQueue`. You may declare additional queue objects.

    a. Set `secondElement` to the second element in `myQueue`, leaving `myQueue` without its original front two elements.

    b. Set `last` equal to the rear element in `myQueue`, leaving `myQueue` empty.

c. Set `last` equal to the rear element in `myQueue`, leaving `myQueue` unchanged.

d. Make a copy of `myQueue`, leaving `myQueue` unchanged.

22. Indicate whether each of the following applications would be suitable for a queue.

a. An ailing company wants to evaluate employee records in order to lay off some workers on the basis of service time (the most recently hired employees are laid off first).

b. A program is to keep track of patients as they check into a clinic, assigning them to doctors on a first-come, first-served basis.

c. A program to solve a maze is to backtrack to an earlier position (the last place where a choice was made) when a dead-end position is reached.

d. An inventory of parts is to be processed by part number.

e. An operating system is to process requests for computer resources by allocating the resources in the order in which they are requested.

f. A grocery chain wants to run a simulation to see how average customer wait time would be affected by changing the number of checkout lines in the stores.

g. A dictionary of words used by a spelling checker is to be initialized.

h. Customers are to take numbers at a bakery and be served in order when their numbers come up.

i. Gamblers are to take numbers in the lottery and win if their numbers are picked.

23. Consider an operation on a queue that returns the number of items in the queue, without changing the queue itself. Describe how you would implement this operation

a. as a client method, where the client is using our `ArrayQueue` class for queues.

b. as a public method of the `ArrayQueue` class.

24. The following code segment is a count-controlled loop going from 1 through 5. At each iteration, the loop counter is either printed or put on a queue depending on the `boolean` result returned by the method `random`. (Assume that `random` randomly returns either a `true` or a `false`.) At the end of the loop, the items on the queue are removed and printed. Because of the logical properties of a queue, this code segment cannot print certain sequences of the values of the loop counter. You are given an output and asked to determine whether the code segment could generate the output. Assume that the queue is implemented as an `ArrayQueue`.

```
for (count = 1; count <= 5; count++)
{
  if (random())
    System.out.println(count);
  else
    queue.enqueue(count);
while (!queue.isEmpty())
```

```
{
  number = queue.dequeue();
  System.out.println(number);
}
```

a. The following output is possible: 1 2 3 4 5
   i) True ii) False iii) Not enough information

b. The following output is possible: 1 3 5 4 2
   i) True ii) False iii) Not enough information

c. The following output is possible: 1 3 5 2 4
   i) True ii) False iii) Not enough information

4.5 Case Study: Postfix Expression Evaluator

25. Evaluate the following postfix expressions:
    a. 5 7 8 * +
    b. 5 7 8 + *
    c. 5 7 + 8 *
    d. 1 2 + 3 4 + 5 6 * − *

26. Evaluate the following postfix expressions. Some of them may be ill-formed expressions—in that case, identify the appropriate error message (too many operands, too few operands).
    a. 1 2 3 4 5 + + +
    b. 1 2 + + 5
    c. 1 2 * 5 6 *
    d. / 23 * 87
    e. 4567 234 / 45372 231 * + 34526 342 / + 0 *

27. Revise and test the Postfix Expression Evaluator program to
    a. use Flow layout exclusively.
    b. use the `ArrayListStack` class instead of the `ArrayStack` class.
    c. output the range of numbers pushed onto the stack during the evaluation of an expression instead of the average of the numbers.
    d. catch and handle the "divide by zero" situation that was assumed not to happen; for example, if the input expression is 5 3 3 − / the result would be the message "illegal divide by zero."
    e. support a new operation indicated by "^" that returns the larger of its operands; for example, 5 7 ^ = 7.

Programming Project

28. This problem requires you to write a program to convert an infix expression to postfix format. The general form of your program should be similar to our test drivers; i.e., similar to the Palindrome and Balanced programs in this chapter. The main input is listed in a text file. Likewise, the results are sent to a text file. The file names are passed to the program as command line parameters. An output frame displays summary statistics (for example, the number of expressions converted) and allows the user to end the program.

    The evaluation of an infix expression such as `A + B * C` requires knowledge of which of the two operations, + or *, should be performed first. In general, `A + B * C` is to be interpreted as `A + (B * C)` unless otherwise specified. We say that multiplication takes `precedence` over addition. Suppose that we would now like to convert `A + B * C` to postfix. Applying the rules of precedence, we begin by converting the first portion of the expression that is evaluated, namely the multiplication operation. Doing this conversion in stages, we obtain

`A + B * C`	*Given infix form*
`A + B C *`	*Convert the multiplication*
`A B C * +`	*Convert the addition*

    (The part of the expression that has been converted is underlined.)

    The major rules to remember during the conversion process are that the operations with highest precedence are converted first and that after a portion of an expression has been converted to postfix, it is to be treated as a single operand. Let us now consider the same example with the precedence of operators reversed by the deliberate insertion of parentheses:

`( A + B ) * C`	*Given infix form*
`A B + * C`	*Convert the addition*
`A B + C *`	*Convert the multiplication*

    Note that in the conversion from "`A B + * C`" to "`A B + C *`", "`A B +`" was treated as a single operand. The rules for converting from infix to postfix are simple, provided that you know the order of precedence.

    We consider four binary operations: addition, subtraction, multiplication, and division. These operations are denoted by the usual operators, +, 2, *, and /, respectively. There are two levels of operator precedence. Both * and / have higher precedence than + and 2. Furthermore, when unparenthesized operators of the same precedence are scanned, the order is assumed to be left to right. Parentheses may be used in infix expressions to override the default precedence.

    As we discussed in this chapter, the postfix form requires no parentheses. The order of the operators in the postfix expressions determines the actual order of operations in evaluating the expression, making the use of parentheses unnecessary.

Input

The input file contains a collection of error-free infix arithmetic expressions, one expression per line. Expressions are terminated by semicolons, and the final expression is followed by a period. An arbitrary number of blanks and end-of-lines may occur between any two symbols in an expression. A symbol may be an operand (a single uppercase letter), an operator (+, 2, *, or /), a left parenthesis, or a right parenthesis.

Sample Input:

```
A + B – C ;
A + B * C ;
(( A + B ) / ( C – D ) + E ) / ( F + G ) .
```

Output

Your output file should consist of each input expression, followed by its corresponding postfix expression. All output (including the original infix expressions) must be clearly formatted (or reformatted) and also clearly labeled.

Sample Output:

```
Infix:    A + B – C ;
Postfix:  A B + C -
Infix:    A + B * C ;
Postfix:  A B C * +
Infix:    ( A + B ) / ( C – D ) ;
Postfix:  A B + C D – /
Infix:    ( ( A + B ) * ( C – D ) + E ) / ( F + G ) .
Postfix:  A B + C D – * E + F G + /
```

Your frame output can simply state how many expressions were converted and instruct the user to close the window to exit the program.

Discussion

In converting infix expressions to postfix notation, the following fact should be taken into consideration: In infix form the order of applying operators is governed by the possible appearance of parentheses and the operator precedence relations; however, in postfix form the order is simply the "natural" order—in other words, the order of appearance from left to right.

Accordingly, subexpressions within innermost parentheses must first be converted to postfix, so that they can then be treated as single operands. In this fashion, parentheses can be successively eliminated until the entire expression has been converted. The last pair of parentheses to be opened within a group of nested parentheses encloses the first subexpression within that group to be transformed.

This last-in, first-out behavior should immediately suggest the use of a stack. Your program may utilize any of the operations in the Stack ADT.

In addition, you must devise a Boolean method that takes two operators and tells you which has higher precedence. This is helpful because in Rule 3 below you need to compare the next input symbol to the top stack element. Question: What precedence do you assign to '('? You need to answer this question because '(' may be the value of the top element in the stack.

You should formulate the conversion algorithm using the following six rules:

Rule 1: Scan the input string (infix notation) from left to right. One pass is sufficient.

Rule 2: If the next symbol scanned is an operand, it may be immediately appended to the postfix string.

Rule 3: If the next symbol is an operator,

a. Pop and append to the postfix string every operator on the stack that
   i) is above the most recently scanned left parenthesis, and
   (ii) has precedence higher than or equal to that of the new operator symbol.
b. Then push the new operator symbol onto the stack.

Rule 4: When an opening (left) parenthesis is seen, it must be pushed onto the stack.

Rule 5: When a closing (right) parenthesis is seen, all operators down to the most recently scanned left parenthesis must be popped and appended to the postfix string. Furthermore, this pair of parentheses must be discarded.

Rule 6: When the infix string is completely scanned, the stack may still contain some operators. (No parentheses at this point. Why?) All these remaining operators should be popped and appended to the postfix string.

Data Structure

You may use either stack implementation from the chapter. Outside of the Stack ADT operations, your program may not assume knowledge of the stack implementation. If you need additional stack operations, you should specify and implement them using the operations for the Stack ADT.

Examples

Here are two examples to help you understand how the algorithm works. Each line on the following table demonstrates the state of the postfix string and the stack when the corresponding next infix symbol is scanned. The rightmost symbol of the stack is the top symbol. The rule number corresponding to each line demonstrates which of the six rules was used to reach the current state from that of the previous line.

Example 1: Input expression is `A + B * C / D - E`

Next Symbol	Postfix String	Stack	Rule
A	A		2
+	A	+	3
B	A B	+	2
*	A B	+ *	3
C	A B C	+ *	2
/	A B C *	+ /	3
D	A B C * D	+ /	2
-	A B C * D / +	−	3
E	A B C * D / + E	−	2
	A B C * D / + E −		6

Example 2: Input expression is `(A + B * (C - D) ) / E`

Next Symbol	Postfix String	Stack	Rule
(		(	4
A	A	(	2
+	A	(+	3
B	A B	(+	2
*	A B	(+ *	3
(	A B	(+ * (	4
C	A B C	(+ * (	2
-	A B C	(+ * (-	3
D	A B C D	(+ * (-	2
)	A B C D-	(+ *	5
)	A B C D - * +		5
/	A B C D - * +	/	3
E	A B C D - * + E	/	2
	A B C D - * + E /		6

Deliverables

- A listing of your source code files
- A listing of your implemented test plan

Chapter 5

Linked Structures

Goals

Measurable goals for this chapter include that you should be able to

- define and use a self-referential class to build a chain of objects (a linked structure)
- implement the Stack ADT as a linked structure
- implement the Queue ADT as a linked structure
- implement the Unsorted List ADT as a linked structure
- implement the Sorted List ADT as a linked structure
- compare alternative implementations of an abstract data type with respect to performance
- describe our list framework, identifying the interfaces, abstract classes, and concrete classes

In the last chapter, we implemented our Stack ADT and our Queue ADT using an array to hold the elements. Our array-based implementations allow the size of the array to be passed to the constructor at run time rather than requiring that it be known at compile time. However, these implementations still require that the maximum stack or queue size be known when the structure is instantiated.

We also implemented the Stack ADT using an object of the `ArrayList` class to hold the elements. The `ArrayList` class manages memory for the underlying array by allocating a larger array when necessary and copying the contents of the old array into the new one. In this chapter, we employ another technique: We allocate space on the stack or queue for each individual element as it is pushed or enqueued. We also apply the same approach to our unsorted and sorted lists.

5.1 Implementing a Stack as a Linked Structure

We have implemented lists, stacks, and queues using arrays. These implementations all suffer from the same drawback: the size of the structure must be determined when the object is instantiated. For example, when we use a variable of class `ArrayStack`, the maximum number of stack items is passed as an argument to the constructor, and an array of that size is allocated. If we use fewer elements, space is wasted; if we need to push more elements than the array can hold, we cannot. It would be nice if we could just get space for stack elements as we need it.

Self Referential Structures

Java does supply an operation for dynamically allocating space, an operation we have been using for all of our objects—the `new` operation. How can we use the `new` operation to dynamically build lists, stacks, and queues? We do not want to create a new array each time we add an element to one of these structures; we just want to create a new holder for the reference to the element. The question is, where can we hold this reference? So far, we have been collecting all of the references used in our structures in an array. What else can we do?

Let's just consider stacks.

Figure 5.1 shows the internal and abstract views of the results of the following sequence of stack operations using `ArrayStack`. (In the figures of this section we use the same identifiers that we use in our implementations; assume A, B, and C represent objects.)

```
StackInterface myStack;
myStack = new ArrayStack(4);
myStack.push(A);
myStack.push(B);
myStack.push(C);
```

We would like to remove the array from that figure, and from the stack implementation.

Suppose we just get rid of the array and have the instance variable `stack` of the `ArrayStack` object point to the object that is on top of the stack. That is the very object that we want to return when the `top` operation is invoked, so it makes sense to keep a reference to it. Figure 5.2 shows the result of the same code sequence using this new

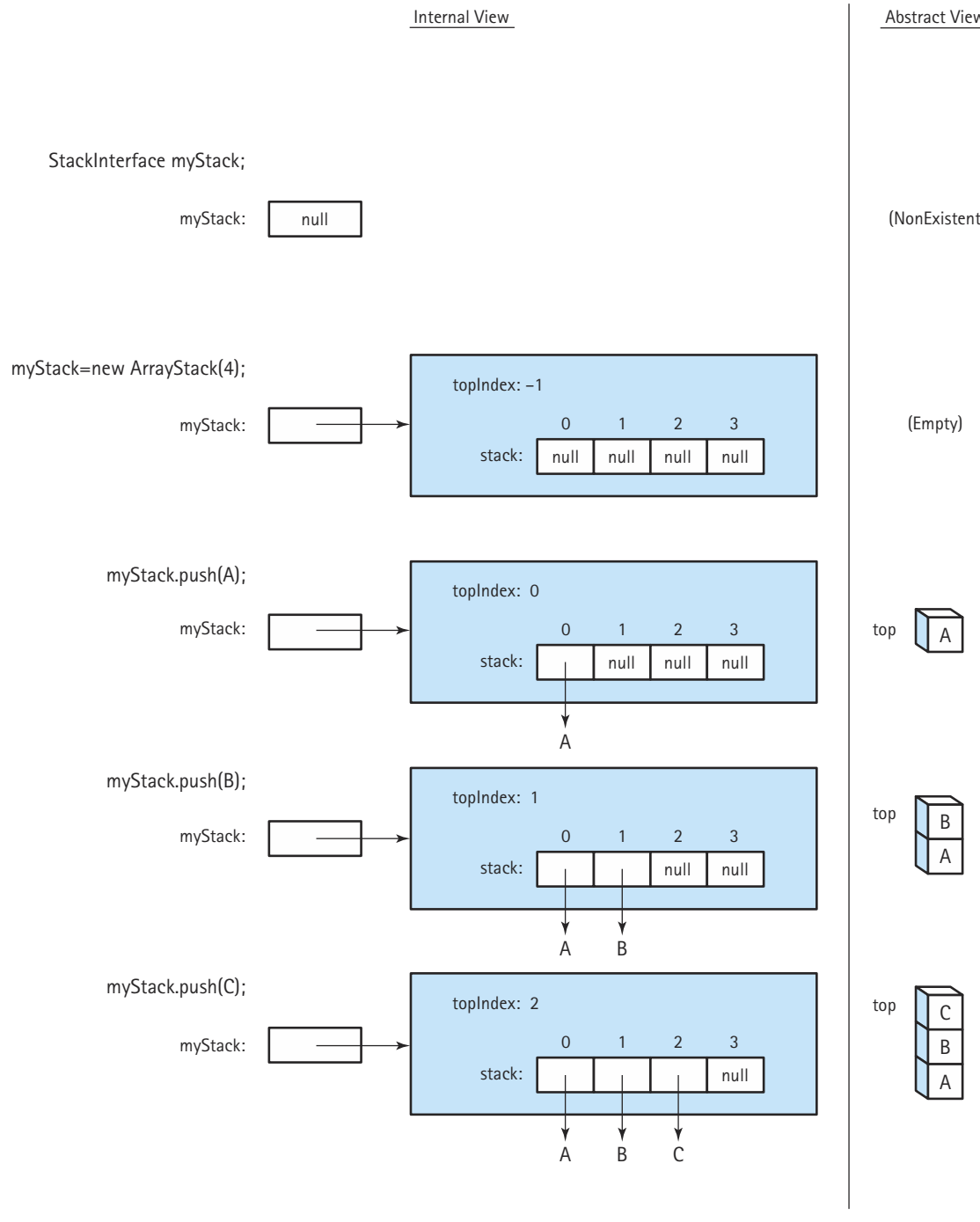


Figure 5.1 *Results of stack operations using an array*

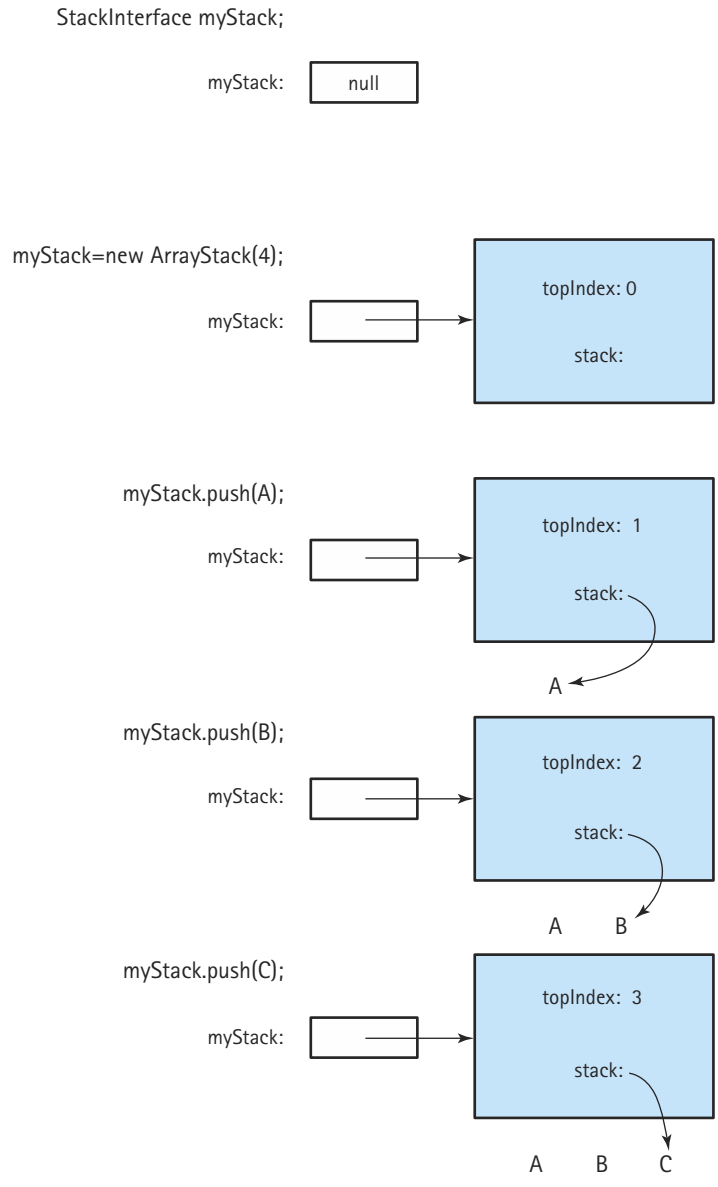


Figure 5.2 *Results of stack operations without using an array—losing references*

approach. Do you see any problems with this approach? It is true that a `top` operation could return a reference to object C, the current top of stack, but what happens if the `pop` operation is invoked? The element C on the top of the stack would be removed from the stack; but there is now no way to indicate the new top of stack. There is no longer any reference to the objects B and A.

What we need is a place to hold our references to A and B while C is on the top of the stack. But without an array where can we hold them? Suppose that every time we add a new element to the stack we create a reference to the previous top of the stack. If we do not have an array to store this reference, the only other place we can store it is as part of the new stack element. We do this by defining a class called `StackNode`. This class contains two instance variables: `info` is a reference to an object on the stack, and `link` is a reference to the `StackNode` that represents the next lower stack element. A single `StackNode` is pictured in Figure 5.3. Pushing an element onto the stack adds another link to the chain. Figure 5.4 shows the result of the previous code sequence using this approach. It graphically demonstrates the dynamic allocation of space for the references to the stack elements.

The `StackNode` class is defined as:

```
private class StackNode
// Used to hold references to stack nodes for the linked stack
// implementation
{
  private Object info;
  private StackNode link;
}
```

`StackNode` is an example of a self-referential class. We have emphasized the code related to the self-referential aspects in the declaration above. The `StackNode` class defines a variable to reference any object (the `info` variable) and a variable to reference another `StackNode` (the `link` variable). That next `StackNode` can hold a reference to any object and a reference to another `StackNode`. That next `StackNode` can hold a reference to any object and a reference to another `StackNode`. And so on until a particular `StackNode` holds the value `null` in its `link`.

Self-referential class A class that includes an instance variable or variables that can hold a reference to an object of the same class

Note: The terms reference and link are used interchangeably when discussing data structures. The ADT implementations presented in this chapter are therefore referred to as "reference-based" or as "linked structures."

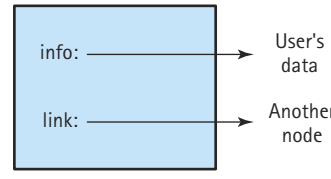


Figure 5.3 *A single node*

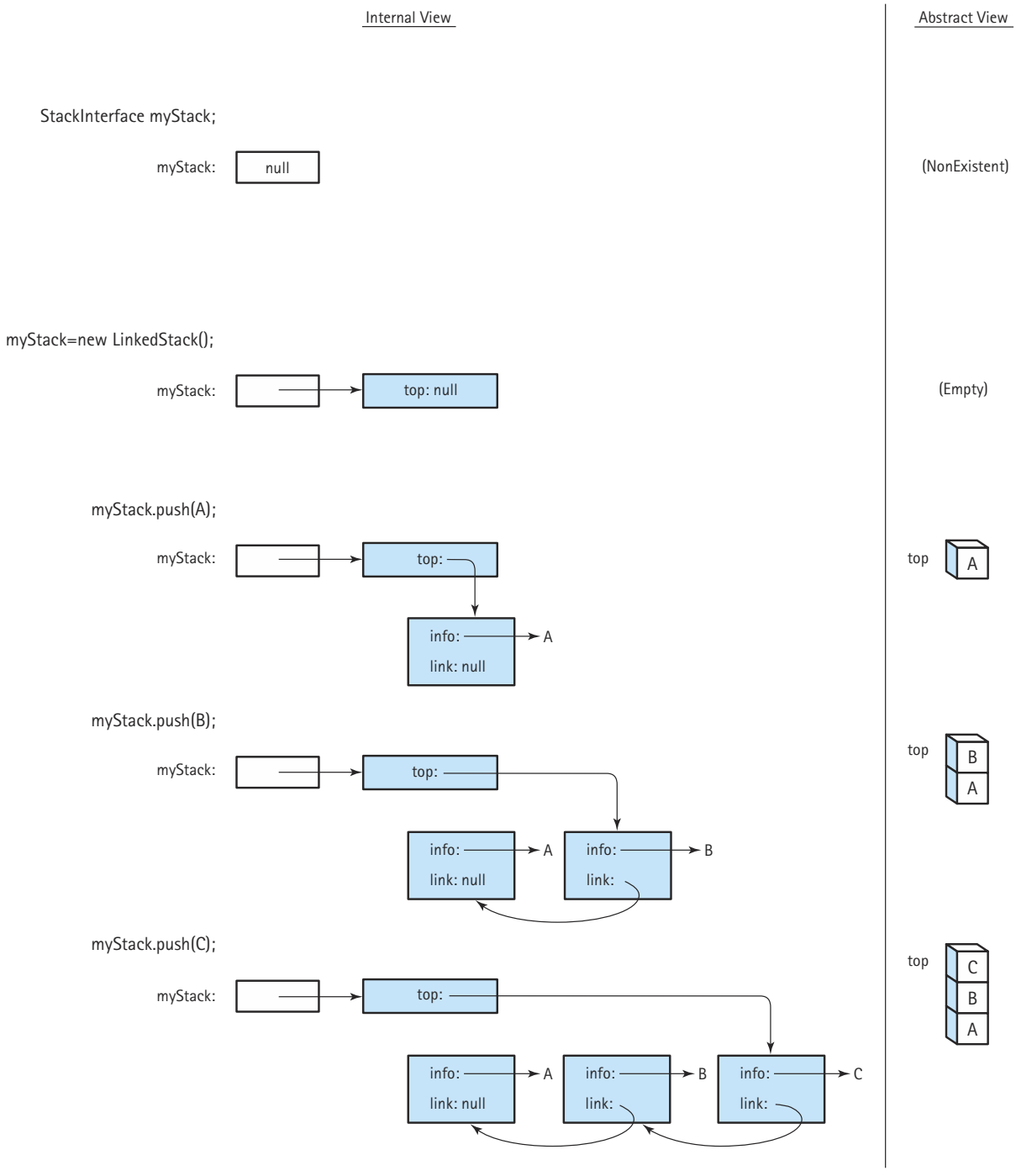


Figure 5.4 *Results of stack operations using* `StackNode`

The LinkedStack Class

We call our new stack class the `LinkedStack` class, to differentiate it from the array-based class of the previous chapter. Like the `ArrayStack` class, our new stack class implements the `StackInterface` interface. Recall that when we defined that interface our purpose was to provide a definition of a Stack ADT that did not depend on the underlying implementation. The `StackInterface` interface is defined in the `ch04.stacks` package, so we must import that package into the stack classes we define in this chapter. Since we are organizing our files by chapter, we create a new stack-related package, `ch05.stacks`, to hold the classes defined in this chapter related to defining stacks.

As implied in Figure 5.4, we need to define only one instance variable in the `LinkedStack` class. We use this variable to hold a reference to the top of the stack, so we call it `top`. It is a reference to an object of the class `StackNode`.

We define the `StackNode` class itself inside the `LinkedStack` class. This is an example of an inner class. By doing this, we allow the methods of the `LinkedStack` class to directly access the attributes of the `StackNode` class, while still hiding the information in `StackNode` objects from every other class. Alternately, we could define the `StackNode` class as a separate class and provide constructors, observers, and transformers to allow us to manipulate objects of the class. We feel that in this case, since the only purpose of `StackNode` is to provide nodes for the reference-based implementation of stacks, the easiest approach is to define `StackNode` as an inner class of `LinkedStack`. The use of inner classes can be tricky—lots of naming and reference issues arise. We suggest you restrict your use of this construct to that shown here; that is, restrict your use to that of creating self-referential structures, until you are able to spend some time studying the semantics of inner classes more thoroughly.

Inner class A class defined inside of another class. The outer class can access the private variables of the inner class

When we instantiate an object of type `LinkedStack`, we must set the top of stack variable to `null`. Therefore, we need to include a single-statement constructor in our class. The beginning of the class definition looks like this:

```
//------------------------------------------------------------------------
// LinkedStack.java            by Dale/Joyce/Weems                 Chapter 5
//
// Implements StackInterface using a linked list to hold the stack items
//------------------------------------------------------------------------

package ch05.stacks;

import ch04.stacks.*;

public class LinkedStack implements StackInterface
```

```
{
  private class StackNode
  // Used to hold references to stack nodes for the linked stack implementation
  {
    private Object info;
    private StackNode link;
  }

  private StackNode top;   // Reference to the top of this stack

  public LinkedStack()
  // Constructor
  {
    top = null;
  }
```

Let's see how we might implement our stack operations using this approach.

The push Operation

We can modify the design of the `push` operation to allocate space for each new element dynamically.

push (item)

Allocate space for new stack element
Set the element reference to item
Set the link reference to the previous top of stack
Set the top of stack to the new stack element

Figure 5.5 graphically displays the effect of each step of the algorithm, starting with a stack that already contains A and B and showing what happens when C is pushed onto it.

Let's look at the algorithm line by line. Follow our progress through both the algorithm and Figure 5.5 during this discussion. We begin by allocating space for a new stack element using Java's `new` operation:

```
StackNode newNode = new StackNode();
```

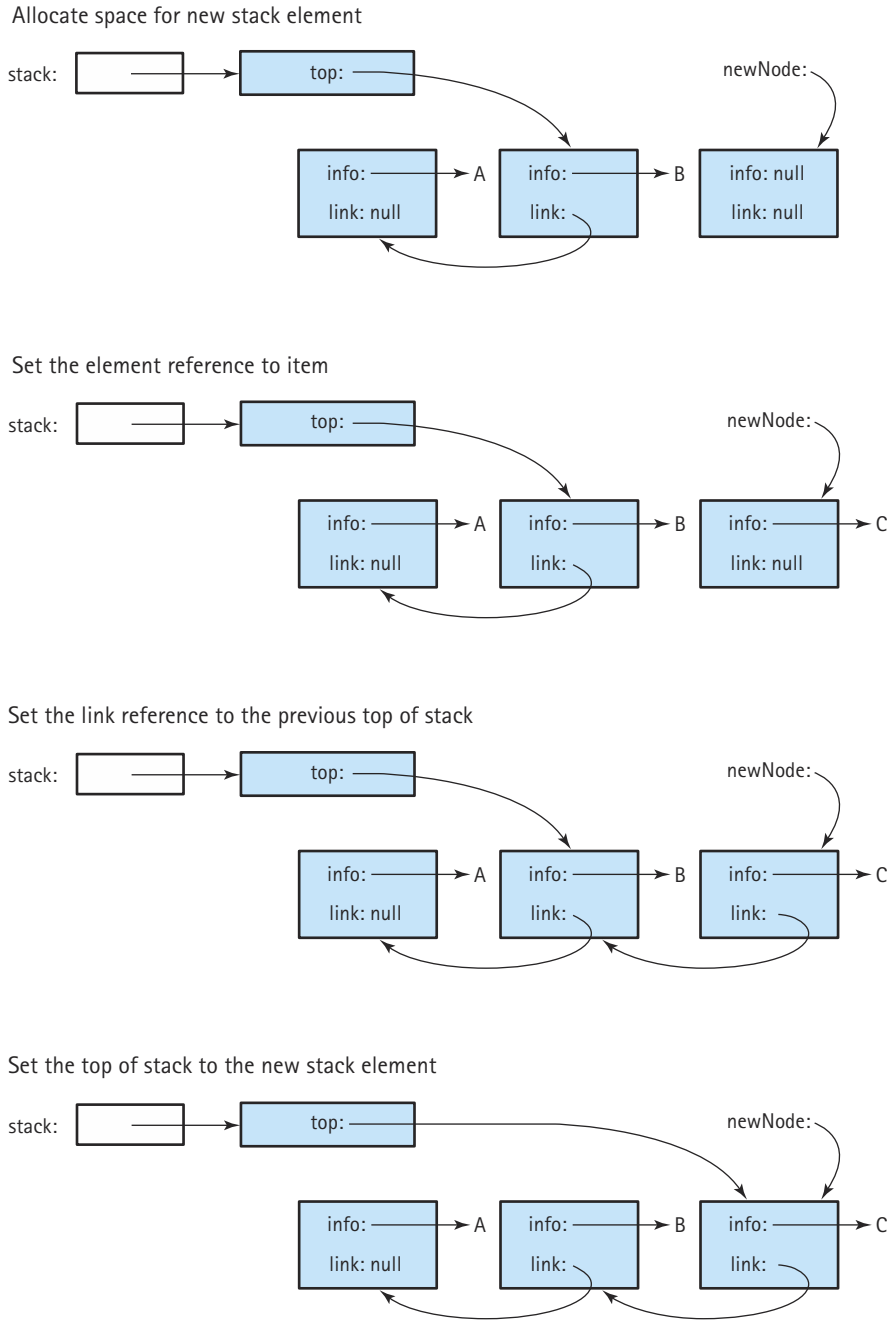


Figure 5.5 *Results of* `push` *operation*

So, newNode is a reference to an object that contains two attributes: info of class Object and link of the class StackNode. Next we need to set the values of these attributes:

```
newNode.info = item;
newNode.link = top;
```

Therefore, info references the item pushed onto the stack, and link references the previous top of stack. Finally, we need to reset the top of the stack to reference the new element:

```
top = newNode;
```

Putting it all together, the code for the push method is:

```
public void push(Object item)
// Adds an element to the top of this stack
{
  StackNode newNode = new StackNode();
  newNode.info = item;
  newNode.link = top;
  top = newNode;
}
```

Note that the order of these tasks is critical. If we changed the top variable before setting the link of the new element, we would lose access to the stack nodes! This situation is generally true when we are dealing with a linked structure: you must be very careful to change the references in the correct order, so that you do not lose access to any of the data.

You have seen how the algorithm works on a stack that contains elements. What happens if this method is called when the stack is empty? Let's trace through it again. Figure 5.6 shows graphically what occurs. Space is allocated for a reference to the new element and the element is put into the space. Does the method correctly link the new node to the top of an empty stack? The link of the new node is assigned the value of top. What is this value when the stack is empty? It is null, which is exactly what we want to put into the link of the last node of a linked stack. Then top is reset to point to the new node. So this method works for an empty stack, as well as a stack that contains elements.

The pop Operation

Recall that the pop operation is essentially the reverse of the push operation. Instead of putting an item onto the top of the stack and "pushing" the other items down, we remove the top item from the stack and "pull" the other items up.

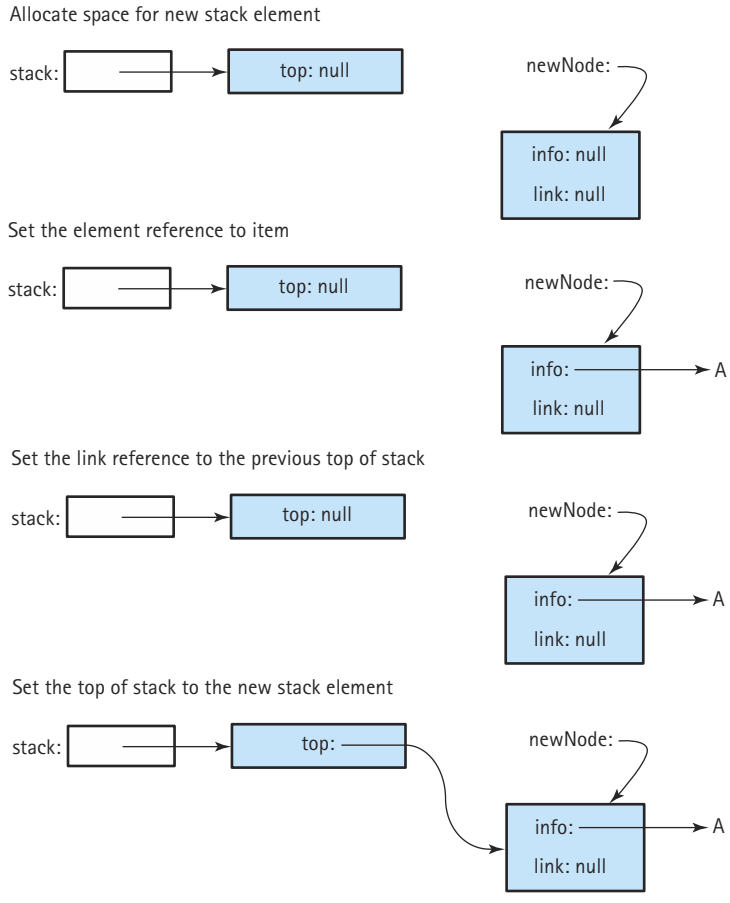


Figure 5.6 *Results of* `push` *operation on an empty stack*

Of course we don't do any actual "pushing" or "pulling," but that is how we think of the stack items behaving. In this case, "pulling" the other items up simply means resetting the stack's `top` variable to reference the next item. In fact, that is all we really have to do. Resetting `top` to the next stack item effectively removes the top item from the stack. See Figure 5.7. This only requires a single line of code:

```
top = top.link;
```

When this code is executed, `top` refers to the `StackNode` object that was previously linked to from the `link` of attribute `top`. In other words, it refers to the "second" lowest `StackNode` object. The stack can no longer be used to reference the "first" object, since we overwrote our only reference to it. As indicated in the figure, we have removed the system's only reference to the "first" node object; the system garbage collector eventually reclaims the space it uses. If the `info` attribute of this object represents the only

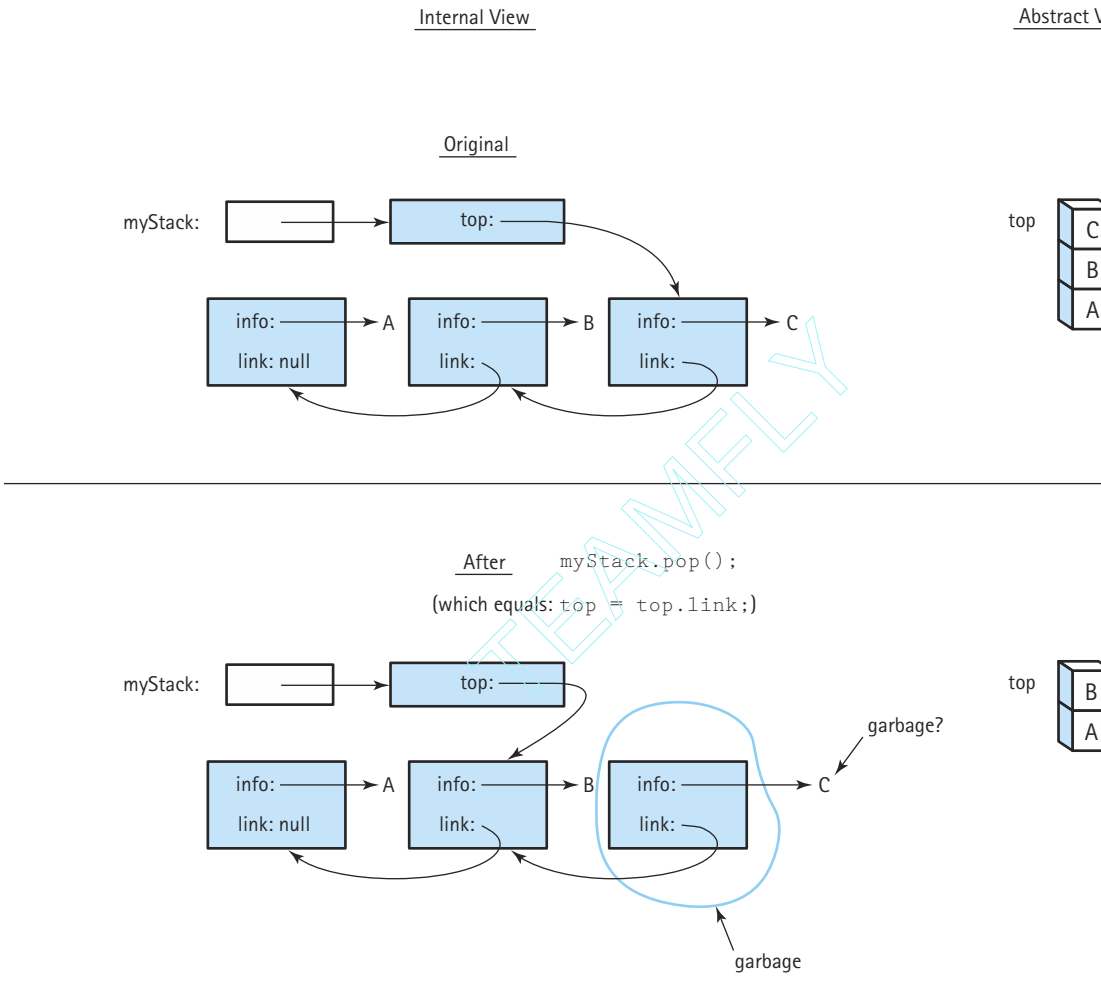


Figure 5.7 *Results of* `pop` *operation*

reference to the data object, represented in the figure by the "C", it too is garbage and its space will be reclaimed.

Are there any special cases to consider? Since we are removing an item from the stack, we should be concerned with empty stack situations. What happens if we try to pop an empty stack? In this case the `top` variable contains `null` and the assignment statement "`top = top.link;`" results in a run-time error. The Java run-time system raises a `NullPointerException`. To control this problem ourselves, we protect the

assignment statement using the Stack ADT's `isEmpty` operation. Therefore, the code for our `pop` method is:

```
public void pop()
// Removes an element from the top of this stack
{
  if (!isEmpty())
  {
    top = top.link;
  }
  else
    throw new StackUnderflowException("Pop attempted on an empty stack.");
}
```

We use the same `StackUnderflowException` that we used in Chapter 4.

Are there any more special cases to consider? We've considered popping from an empty stack, and popping from a stack with several elements. There is one more special case—popping from a stack with only one element. We need to make sure that in this case the stack is empty after the operation is performed. Let's see. When our stack is instantiated, the `top` variable is set equal to `null`. When an element is pushed onto the stack, its `link` is set equal to the current `top` variable; therefore, when the first element is pushed onto our stack, its `link` is set to `null`. Of course, the first element pushed onto the stack is the last element popped off. This means that the last element popped off the stack has a `link` value of `null`. Since the `pop` method sets `top` to the value of this `link` attribute, after the last value is popped `top` again has the value `null`, just as it did when the stack was first instantiated. We conclude that the `pop` method works for a stack of one element. Figure 5.8 graphically depicts pushing a single element onto a stack and then popping it off.

The Other Stack Operations

Recall that the `top` operation simply returns a reference to the top element of the stack. At first glance this might seem very straightforward. Simply code

```
return top;
```

since `top` references the element on the top of the stack. However, remember that `top` references a `StackNode` object. Whatever program is using the Stack ADT is not concerned about `StackNode` objects. In fact, it doesn't even "know" what a `StackNode` object is, since the definition of `StackNode` is hidden inside the `LinkedStack` class. The client program is only interested in the object that is referenced by the `info` variable of the `StackNode`.

Let's try again. To return the `info` of the top `StackNode` object we simply code:

```
return top.info;
```

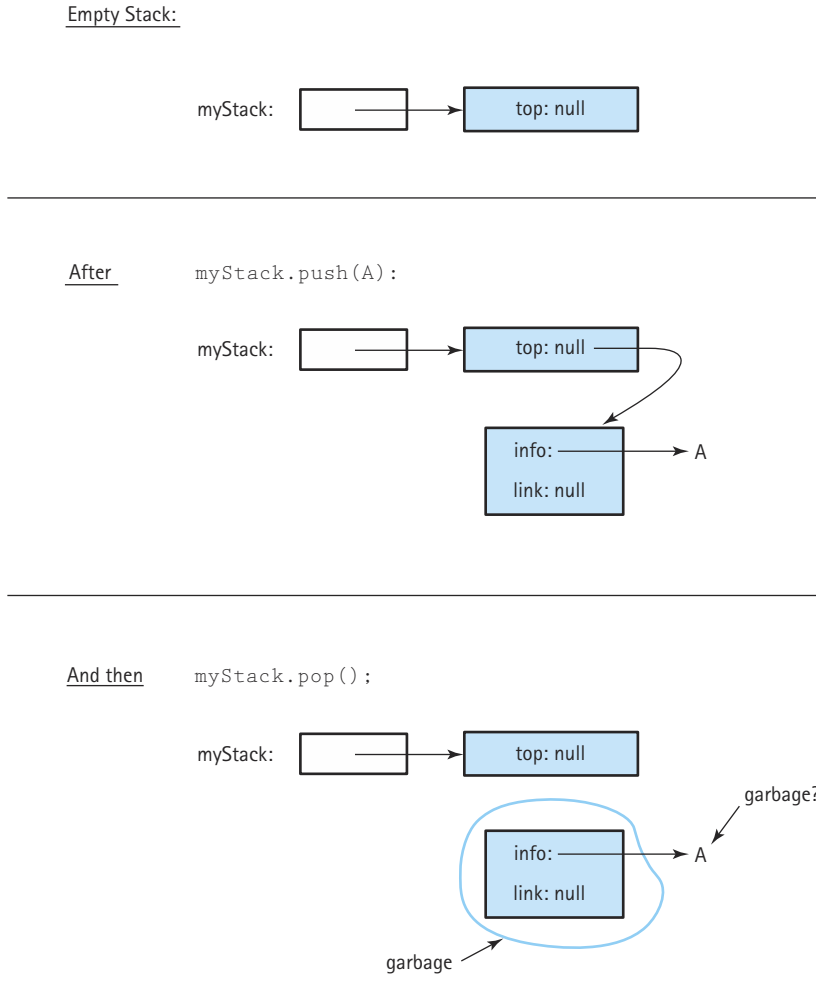


Figure 5.8 *Results of push then pop on an empty stack*

That's better, but we still need to do a little more work. What about the special case when the stack is empty? In that case we need to throw an exception instead of returning an object. The final code for the `top` method is:

```
public Object top()
// Returns the element on top of this stack
{
  if (!isEmpty())
    return top.info;
  else
    throw new StackUnderflowException("Top attempted on an empty stack.");
}
```

That wasn't bad; the `isEmpty` method is even easier. If we initialize an empty stack by setting the `top` variable to `null`, then we can detect an empty stack by checking for the value `null`.

```
public boolean isEmpty()
// Checks if this stack is empty
{
  if (top == null)
    return true;
  else
    return false;
}
```

An even simpler way of writing this is

```
return (top == null);
```

What about the `isFull` method? Using dynamically allocated nodes rather than an array, we no longer have an explicit limit on the stack size. We can continue to get more nodes until we run out of system memory. As we did when we used an `ArrayList` to implement a Stack ADT in Chapter 4, we simply always return `false` for the `isFull` method. The only way the stack could be full is if the program runs out of system space. In this rare case the Java run-time system raises an exception anyway, so we decide just to rely on the run-time system's built-in mechanisms.

```
public boolean isFull()
// Checks if this stack is full
{
    return false;
}
```

The linked implementation of the Stack ADT can be tested using the same test plan that was written for the array-based version. It can also be used with the examples and case studies presented in Chapter 4 for further verification.

Comparing Stack Implementations

Let's compare the two different implementations of the Stack ADT, `ArrayStack` and `LinkedStack`, in terms of storage requirements and efficiency of the algorithms. First the storage requirements. An array variable of the maximum stack size takes the same amount of memory, no matter how many array slots are actually used; we need to reserve space for the maximum possible. The linked implementation, using dynamically allocated storage, requires space only for the number of elements actually on the stack at run time. Note, however, that the elements are larger because we must store the reference to the next element as well as the reference to the user's data.

We compare the relative execution "efficiency" of the two implementations in terms of Big-O notation. In both implementations, `isFull` and `isEmpty` clearly are

O(1). They always take a constant amount of work. What about `push`, `pop`, and `top`? Does the number of elements in the stack affect the amount of work done by these operations? No, it does not. In both implementations, we directly access the top of the stack, so these operations also take a constant amount of work. They too have O(1) complexity.

Only the class constructor differs from one implementation to the other in terms of the Big-O efficiency. In the array-based implementation, when the array is instantiated, the system creates and initializes each of the array locations. Since in this case it is an array of objects, each of the array slots is initialized to the value `null`. The number of array slots is equal to the maximum number of possible stack elements. We call this *N* and say that the array-based constructor is O(N). For the linked approach, the constructor simply sets the `top` variable to `null`, so it is only O(1).

Overall the two stack implementations are roughly equivalent in terms of the amount of work they do.

So which is better? The answer, as usual, is: It depends on the situation. The linked implementation certainly gives more flexibility, and in applications where the number of stack items can vary greatly, it wastes less space when the stack is small. Why then would we ever want to use the array-based implementation? The reason is: It's short, simple, and efficient. If pushing occurs frequently, the array-based implementation executes faster because it does not incur the run-time overhead of the `new` operation.[1] When the maximum size is small and we can be sure that we do not need to exceed the declared maximum size, the array-based implementation is a good choice. Also, if you are programming in a language that does not support dynamic storage allocation, an array implementation may be the only good choice.

5.2 Implementing a Queue as a Linked Structure

The major weakness of the array-based implementation of a FIFO queue is identical to that of a stack: the need to create an array big enough for a structure of the maximum expected size. This size is set once and cannot change. If a much smaller number of elements is actually needed, we have wasted space. If a larger number of elements is unexpectedly needed, we are in trouble. We cannot extend the size of the array. We would have to allocate a larger array, copy the elements into it, and deallocate the smaller array.

We know, however, from our discussion of stacks, that we can get around this problem by using dynamic storage allocation to get space for each queue element only as needed. This implementation relies on the idea of linking the elements one to the next to form a chain. We can use the same approach we used for stacks. We define an inner class `QueueNode` within our Queue ADT implementation. Also as we did for stacks, we create a separate Chapter 5 queue package to hold the queue class defined in this chapter and we import the queue package defined for Chapter 4 into that class.

---

[1] This statement is true only when implementing a stack "by reference." If the stack is implemented "by copy," then a `new` statement is required for each push operation in the array-based implementation also.

In the array-based implementation of a queue, we decided to keep track of two indexes that pointed to the front and rear boundaries of the data in the queue. In a linked representation, we can use two references, `front` and `rear`, to mark the front and the rear of the queue. When the queue is empty, both of these references should equal `null`. Therefore, the constructor for the queue must initialize them both accordingly. The beginning of our class definition looks like this:

```
//--------------------------------------------------------------------------
// LinkedQueue.java            by Dale/Joyce/Weems                  Chapter 5
//
// Implements QueueInterface using a linked list to hold the queue items
//--------------------------------------------------------------------------

package ch05.queues;

import ch04.queues.*;

public class LinkedQueue implements QueueInterface
{
  private class QueueNode
  // Used to hold references to queue nodes for the linked queue implementation
  {
    private Object info;
    private QueueNode link;
  }

  private QueueNode front;   // Reference to the front of this queue
  private QueueNode rear;    // Reference to the rear of this queue

  public LinkedQueue()
  // Constructor
  {
    front = null;
    rear = null;
  }
```

Figure 5.9 graphically depicts our queue representation. A few comments about our graphical representation scheme are necessary. To make our figures easier to read, we often depict queues by showing their instance variables (`front` and `rear`) in different areas of the figure. Remember that these variables are actually collected together in a single queue object. Additionally, by now you realize that dynamically allocated nodes in linked structures exist "somewhere in the system memory" rather than in adjacent locations like array slots, but we are going to show the nodes arranged linearly for clarity.

Note the relative positions of `front` and `rear` in the figure. Had they been reversed (as in Figure 5.10), we would have difficulty implementing the `dequeue` operation.

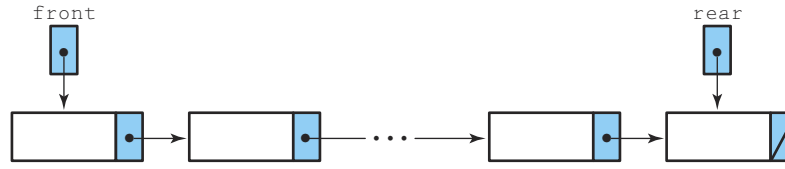


Figure 5.9 *A linked queue representation*

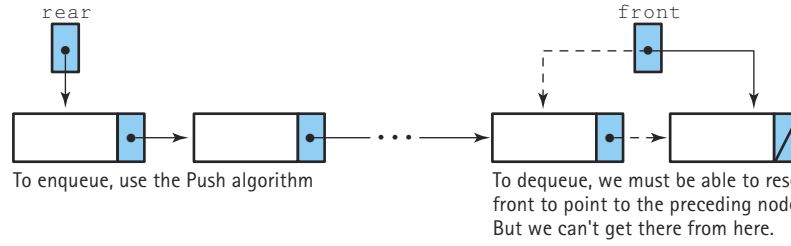


Figure 5.10 *A bad queue design*

Remember that we `dequeue` from the front end of the queue. To remove the node that represents the front of the queue, we have to reset the `front` reference to the next node on the chain. If we implement the queue as in Figure 5.10, we can't easily obtain a reference to the next node in the chain, since it is the preceding node. We would either have to traverse the whole list (an O(*N*) solution—very inefficient, especially if the queue is long) or else keep a list with references in both directions. Use of this kind of a doubly linked structure is not necessary if we set up our queue references correctly to begin with.

The Enqueue Operation

We can `dequeue` elements from the queue using an algorithm similar to our stack `pop` algorithm, with `front` pointing to the first node in the queue. Because we add new elements to the queue by inserting after the last node, however, we need a new `enqueue` algorithm (see Figure 5.11).

Enqueue (item)

Create a node for the new item
Insert the new node at the rear of the queue
Update the reference to the rear of the queue

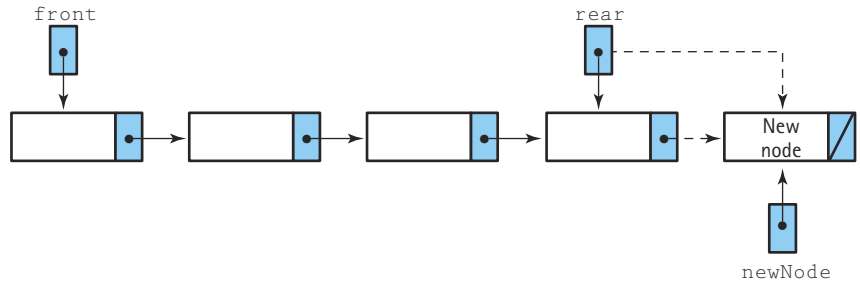


Figure 5.11 *The* `enqueue` *operation*

The first of these tasks is familiar from the stack `push` operation. We get space for a queue node using Java's `new` operator and then store the new item into the node's `info` variable. The new node is inserted at the rear end of the queue, so we also need to set the node's `link` variable to `null`. The corresponding code is:

```
QueueNode newNode = new QueueNode();
newNode.info = item;
newNode.link = null;
```

The next part of the `enqueue` algorithm involves inserting our new node at the rear of the queue. This requires setting the `link` of the current last element to reference the new node. This can be accomplished by a simple assignment statement:

```
rear.link = newNode;
```

However, we must consider what happens if the queue is empty when we `enqueue` the item. In general, when using references, always consider and handle the special case of the reference being equal to `null`; in this case, you cannot use it to access an object. If the queue is empty when we insert the item, the value of `rear` would be `null`, and the use of `rear.link` would raise a run-time exception. There is no `rear.link`. In this case, we must set `front` to point to the new node:

```
if (rear == null)
  front = newNode;
else
  rear.link = newNode;
```

The last task in the `enqueue` algorithm, updating the `rear` reference, simply involves the assignment

```
rear = newNode;
```

Does this work if this is the first node in the queue? Sure, we always want `rear` to be pointing to the rear node following a call to `enqueue`, regardless of how many items are in the queue.

Putting this all together, we get the following code for the `enqueue` method:

```
public void enqueue(Object item)
// Adds item to the rear of this queue
{
  QueueNode newNode = new QueueNode();
  newNode.info = item;
  newNode.link = null;
  if (rear == null)
    front = newNode;
  else
    rear.link = newNode;
  rear = newNode;
}
```

The Dequeue Operation

As we noted above, the `dequeue` operation can be similar to the stack's `pop` operation. However, recall that the `pop` operation only removed the top element from the stack, whereas the `dequeue` operation both removes and returns the element. Also, as with the stack's `top` operation, we do not want to return the entire `QueueNode`, but only the information in the node's `info` variable.

In writing the `enqueue` algorithm, we noticed that inserting into an empty queue is a special case because we need to make `front` point to the new node also. Similarly, in our `dequeue` algorithm we need to allow for the case of deleting the last node in the queue, leaving the queue empty. If `front` is `null` after we have deleted the front node, we know that the queue is now empty. In this case we need to set `rear` to `null` also. The algorithm for removing the front element from a linked queue is illustrated in Figure 5.12. This algorithm assumes that the test for an empty queue was performed before the `dequeue` routine was entered, so we know that the queue contains at least one node. (We can make this assumption because this is the precondition for `dequeue` in our FIFO Queue ADT specification.)

Dequeue: returns Object

Set item to the information in the front node
Remove the front node from the queue
if the queue is empty
    Set the rear to null
return item

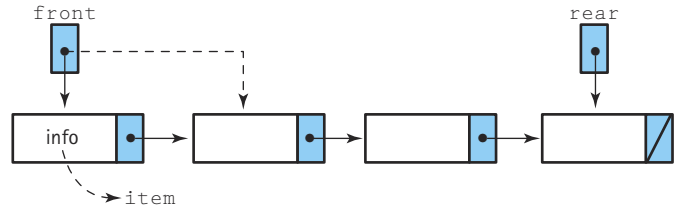


Figure 5.12 *The* `dequeue` *operation*

Again, we look at the implementation of the algorithm line by line. Since we have to return the information from the first element of the queue, we start by "remembering" the information. Recall that the information on a queue is of the class `Object`. So, we declare a local `Object` variable `item`, and assign the information (i.e., the reference to the information) from the front queue element to it:

```
Object item;
item = front.info;
```

Next we must remove the front node from the queue. This is easy. We just reassign the reference to the `front` of the queue to be the reference in the `link` of the current front element (see Figure 5.12). This works even if the resultant queue is empty, since in that case the `link` would hold the value `null`. In this latter case we must also set the `rear` of the queue to `null`, as discussed above:

```
front = front.link;
if (front == null)
  rear = null;
```

Finally, we just return the information we saved earlier:

```
return item;
```

Putting it all together, the code is:

```
public Object dequeue()
// Removes front element from this queue and returns it
{
  Object item;

  item = front.info;
  front = front.link;
  if (front == null)
    rear = null;

  return item;
}
```

The Queue Implementation

The remaining operations (`isEmpty`, `isFull`) are very straightforward. Here is the code for the entire FIFO Queue implementation based on the linked approach:

```
//--------------------------------------------------------------------------
// LinkedQueue.java            by Dale/Joyce/Weems                  Chapter 5
//
// Implements QueueInterface using a linked list to hold the queue items
//--------------------------------------------------------------------------

package ch05.queues;

import ch04.queues.*;

public class LinkedQueue implements QueueInterface
{

  private class QueueNode
  // Used to hold references to queue nodes for the linked queue implementation
  {
    private Object info;
    private QueueNode link;
  }

  private QueueNode front;   // Reference to the front of this queue
  private QueueNode rear;    // Reference to the rear of this queue

  public LinkedQueue()
  // Constructor
  {
    front = null;
    rear = null;
  }

  public void enqueue(Object item)
  // Adds item to the rear of this queue
  {
    QueueNode newNode = new QueueNode();
    newNode.info = item;
    newNode.link = null;
    if (rear == null)
      front = newNode;
    else
      rear.link = newNode;
    rear = newNode;
  }
```

```
  public Object dequeue()
  // Removes front element from this queue and returns it
  {
    Object item;

    item = front.info;
    front = front.link;
    if (front == null)
      rear = null;

    return item;
  }

  public boolean isEmpty()
  // Determines whether this queue is empty
  {
    if (front == null)
      return true;
    else
      return false;
  }

  public boolean isFull()
  // Determines whether this queue is full
  {
      return false;
  }
}
```

A Circular Linked Queue Design

Our `LinkedQueue` class contains two instance variables, one to reference each end of the queue. This design is based on the linear structure of the linked queue. Given only a reference to the front of the queue, we could follow the references to get to the rear, but this makes accessing the rear (to enqueue an item) an O(N) operation. With a reference only to the rear of the queue, we could not access the front because the references only go from front to rear.

However, we could access both ends of the queue from a single reference, if we made the queue circularly linked. That is, the `link` of the rear node would reference the front node of the queue (see Figure 5.13). Now `LinkedQueue` has only one instance variable, rather than two. One interesting thing about this queue implementation is that it differs from the logical picture of a queue as a linear structure with two ends. This queue is a circular structure with no ends. What makes it a queue is its support of FIFO access.

In order to enqueue, we access the "rear" node directly through the reference `rear`. To dequeue, we must access the "front" node of the queue. We don't have a reference to

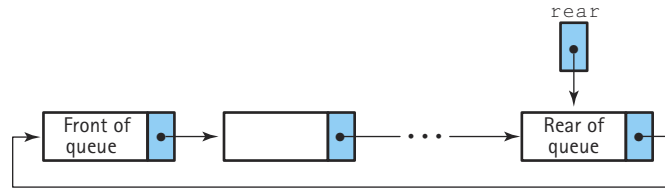


Figure 5.13 *A circular linked queue*

this node, but we do have a reference to the node preceding it—`rear`. The reference to the "front" node of the queue is in `rear.link`. An empty queue would be represented by `rear = null`. Designing and coding the queue operations using a circular linked implementation is left as a programming assignment.

Both linked implementations of the Queue ADT can be tested using the same test plan that was written for the array-based version.

Comparing Queue Implementations

We have now looked at several different implementations of the Queue ADT. How do they compare? As we compared stack implementations, we look at two different factors: the amount of memory required to store the structure and the amount of "work" the solution requires, as expressed in Big-O notation. Let's compare the two implementations that we have coded completely: the array-based implementation and the dynamically linked implementation.

An array variable of the maximum queue size takes the same amount of memory, no matter how many array slots are actually used; we need to reserve space for the maximum possible number of elements. The linked implementation using dynamically allocated storage space requires space only for the number of elements actually in the queue at run time. Note, however, that the node elements are twice as large, because we must store the link (the reference to the next node) as well as the reference to the user's data.

See Figure 5.14 for a depiction of each queue implementation approach, assuming a current queue size of 5 and a maximum queue size (for the array-based implementation) of 100. Note that the array-based implementation requires space for 4 integers and 101 references (one for `myQueue` and one for each array slot) no matter what the size of the current queue. In the example, the linked implementation only requires space for 13 references (one for `front`, one for `rear`, two for each of the current queue elements, and one for `myQueue`). However, the required space increases if the size of the queue increases, based on the formula:

number of required references = 3 + (2 * size of queue)

A simple analysis of this situation tells us that if the average queue size is less than half the maximum queue size, then the linked representation uses less space than the array representation. And vice versa: If the average queue size is larger than half the maximum queue size, the linked representation requires more space. In any case, unless you have a situation in which the maximum queue size is very large, and significantly

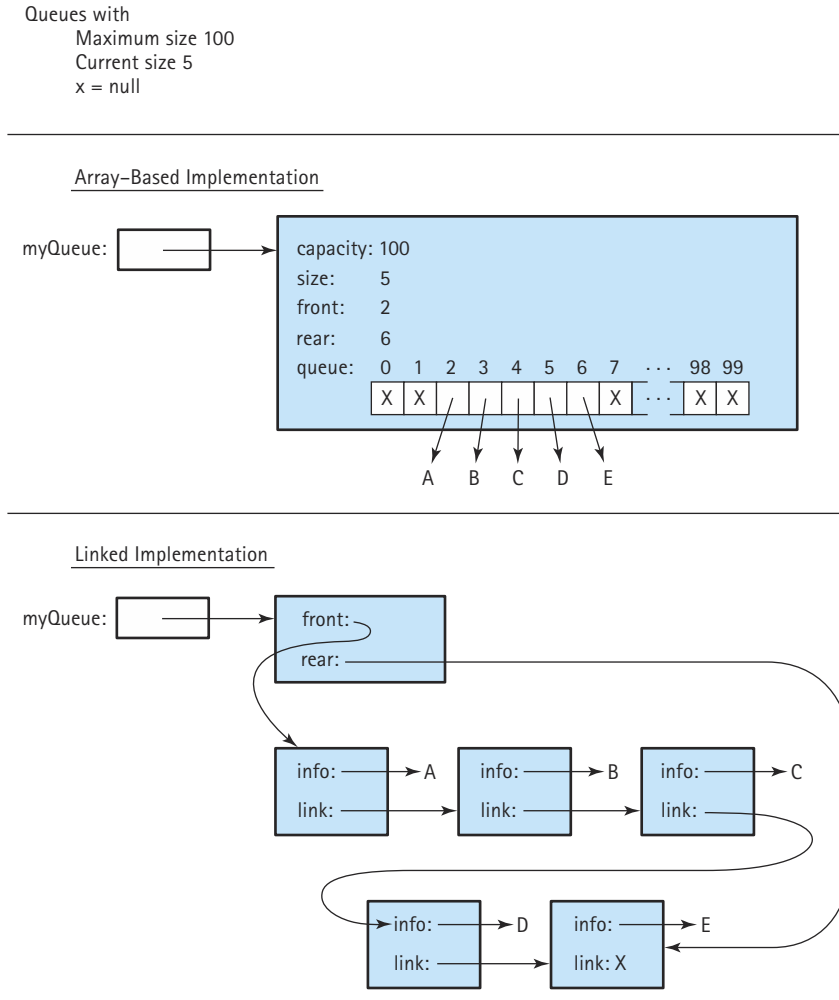


Figure 5.14 *Alternate queue implementations*

larger than the average queue size, the difference between the two implementations in terms of space is not much.

We can also compare the relative execution "efficiency" of the two implementations, in terms of Big-O. In both implementations, the `isFull` and `isEmpty` operations are clearly O(1). They always take the same amount of work regardless of how many items are on the queue. As was the case for stacks, the queue constructor requires O(*N*) steps for the array representation, but is O(1) for the linked representation. What about `enqueue` and `dequeue`? Does the number of elements in the queue affect the amount of work done by these operations? No, it does not; in both implementations, we can directly access the front and the rear of the queue. The amount of

work done by these operations is independent of the queue size, so these operations also have O(1) complexity.

As with the array-based and linked implementations of stacks, these two queue implementations are roughly equivalent in terms of the amount of work they do.

5.3 An Abstract Linked List Class

We have now implemented both our Stack ADT and our Queue ADT as reference-based structures using a self-referential node. We can use the same approach to implementing our List ADT. However, the situation with lists is slightly more complicated, since we are supporting both sorted and unsorted varieties of our ADT.

Overview

Let's review some facts about our lists:

- Our lists use the "by copy" approach to storing and returning information, unlike our stacks and queues, which use the "by reference" approach.
- As we do for our queues, we use a programming "by contract" approach for our lists. We assume that the list holds unique items (based on the key) and that calls to `delete` and `retrieve` pass an argument item whose key matches exactly one element on the list.
- In Section 3.7, we presented the `Listable` interface that allows us to manipulate generic lists; as long as an object is of a class that implements `Listable`, it can be used with our lists.
- In Section 3.7, we also presented an abstract class `List` that defines all the constructs for an array-based list that do not depend on whether or not the list is sorted.
- Finally, in Section 3.7, we presented the concrete class `SortedList` that extends the abstract `List` class, providing an array-based implementation of a sorted list.
- In the Chapter 3 exercises, we asked you to implement a concrete class `UnsortedList` that extends the abstract `List` class, providing an array-based implementation of an unsorted list.
- In Section 4.1, we presented the Java interface `ListInterface` that specifies our List ADT. We stated that, in retrospect, the abstract class `List` should implement `ListInterface`. Both of these classes are part of the `ch04.genericLists` packages.

In summation, `ListInterface` defines our List ADT, `List` is an abstract class that implements `ListInterface` using an array to hold the list elements, and `SortedList` and `UnsortedList` are concrete classes that extend and complete the `List` class. Figure 4.1 presents all of these relationships using a UML class diagram.

Perhaps we should have called our `List` class "`ArrayList`," just as we called our array-based stack implementation `ArrayStack`. Since we had already introduced the `ArrayList` class from the Java library, we decided to just use the name "`List`." Deter-

mining names for our classes is important and not always easy. See the feature section titled Naming Constructs for more discussion on this topic.

In this chapter, we extend our list framework to include implementation classes based on references. Lists implemented in this fashion are known as linked lists.

We extend our framework in a natural way. We already have an abstract class `List` that extends `ListInterface` and defines all the constructs for an array-based list that do not depend on whether or not the list is sorted. In this section, we define an abstract class `LinkedList` in the same manner, except it uses a reference-based approach. In the next two sections, we define concrete classes `UnsortedLinkedList` and `SortedLinkedList` that extend and complete the `LinkedList` class.

Naming Constructs

Choosing appropriate names for our programmer-defined constructs is an important task. In this sidebar, we discuss this task and explain some of the naming conventions used in this textbook.

Java is very lenient in terms of its rules for programmer-defined names. They must begin with a letter; after that, they can contain any combination of letters and digits. They can be as long as you want. Other than the fact that you cannot use Java reserved words such as *while* or *class* as names, there are no other restrictions. Remember, of course, that Java identifiers are case-sensitive.

We have been following standard conventions when naming the constructs created for this text. Our class and interface names all begin with an uppercase letter, such as in `List` or `StackInterface`. Our method and variable names all begin with a lowercase letter, such as in `push` or `list`. If a name contains more than one word, we capitalize the start of each additional word, such as `ArrayStack`, `getNextItem`, or `currentPos`. Other than that, we use lowercase letters in our names.

The name assigned to a construct should provide useful information to someone who is working with the construct. For example, if you declare a variable within a method that is to hold the maximum value of a set of numbers, you should name it based on its use—name it `maximum` or `maxValue` instead of `X`. The same is true for class, interface, and method names.

Since classes tend to represent objects, we usually name them using nouns; for example `ArrayStack`, `LinkedStack`, and `List`. Since methods tend to represent actions, we usually name them using verbs; for example `push`, `reset`, and `dequeue`.

We use interfaces in two ways. We use them to specify ADTs. In this case we use the name of the ADT plus the term "interface" within the name of our interface; for example `ListInterface`. Although this is a bit redundant, it is the approach favored by the Java library creators. Note that the name of the interface does not imply any implementation detail. Classes that implement the `ListInterface` interface can use arrays, vectors, array lists, or references—the interface itself does not restrict implementation options and its name does not imply anything about implementation details. The name does help us identify the purpose of the construct; thus `ListInterface` defines the interface required by the List ADT.

We also use an interface to guarantee that an object supports a certain set of operations, so that it can be used with another class that expects to deal only with objects that support

that set of operations. For example, our `Listable` interface defines operations required by objects to be used with our various list classes. In this case, we name the interface with an adjective—objects of classes that implement this class are "listable" objects. That is, they possess the quality of being able to be listed.

Implementation-Based Class Names

We must confess that we were hesitant to use names such as `ArrayStack` and `LinkedStack` for our classes. Can you guess why? Recall our goal of information hiding: We want to hide implementation detail about the underlying organizational structures and code used to support our ADTs inside the class that implements the ADT. However, if we use terms such as "Array" and "Linked" in the names of our ADTs, then we are revealing clues to the very information we are trying to hide.

One of the main differences between array-based and reference-based ADT implementations is the allocation of computer memory for holding the information associated with the ADT. For our array-based approaches, the space is allocated statically, when the structure is instantiated. For our reference-based approaches, the space is allocated and deallocated dynamically, as needed. An offshoot of this is that our array-based structures have a bounded size, whereas our reference-based structures do not. We considered using this aspect of our structures to provide informative names for our classes that did not completely reveal the underlying structure. For example, we could call our two stack classes `BoundedStack` and `UnboundedStack` instead of `ArrayStack` and `LinkedStack`.

However, we decided not to use this approach. It does not completely solve our naming problem anyway. For example, it is possible to have an unbounded implementation based on dynamically allocated arrays as in our `ArrayListStack`. So we have two unbounded stack classes—if we try to stay away from implementation-dependent names we might call them `UnboundedStack1` and `UnboundedStack2`. We prefer to stay away from using numbers to differentiate our classes when possible.

We finally settled on using implementation-dependent terms within our class names. There are several reasons why we did this:

1. This is the same approach used by the Java Library, for example, the `ArrayList` class. By the way, although we call our array-based stack and queue classes `ArrayStack` and `ArrayQueue`, we do not call our array-based list class "`ArrayList`" because we have already seen that there is a Java library construct of that same name. So we simply call it `List` instead.
2. Although information hiding is important, some information about the implementation is valuable to the client programmer, since it affects the space used by objects of the class and the execution efficiency of the methods of the class. We present comparisons of array-based and reference-based implementations along these dimensions later in this chapter. Using "array" and "linked" in the class names does help convey this information.
3. We already have a construct associated with our ADTs whose name is independent of implementation. It is the interface, for example, the `StackInterface` interface.
4. In this textbook we create multiple implementations of many different ADTs; this is fundamental to the way we study ADTs. Using implementation-dependent names makes it easier to distinguish among these different implementations.

The LinkedList Class

Our `LinkedList` class implements the `ListInterface` interface. For easy reference, we repeat the interface definition here. Note that it is in the `ch04.genericLists` package.

```
//--------------------------------------------------------------------------
// ListInterface.java            by Dale/Joyce/Weems                Chapter 5
//
// Interface for a class that implements a list of unique elements, i.e.,
// no duplicate elements as defined by the key of the list.
// The list has a special property called the current position - the position
// of the next element to be accessed by getNextItem during an iteration
// through the list. Only reset and getNextItem affect the current position
//--------------------------------------------------------------------------

package ch04.genericLists;

public interface ListInterface
{
  public boolean isFull();
  // Effect:         Determines whether this list is full
  // Postcondition:  Return value = (this list is full)

  public int lengthIs();
  // Effect:         Determines the number of elements on this list
  // Postcondition:  Return value = number of elements on this list

  public boolean isThere (Listable item);
  // Effect:         Determines whether element matching item is on this list
  // Postcondition:  Return value = (element with the same key as item is on
  //                 this list)

  public Listable retrieve(Listable item);
  // Effect:         Returns a copy of the list element with the same key as
  //                 item
  // Preconditions:  Item is on this list
  // Postcondition:  Return value = (list element that matches item)

  public void insert (Listable item);
  // Effect:         Adds a copy of item to this list
  // Preconditions:  This list is not full
  //                 Element matching item is not on this list
  // Postcondition:  Copy of item is on this list

  public void delete (Listable item);
  // Effect:         Deletes the element of this list whose key matches item's
  //                 key
```

```
    // Preconditions:  Exactly one element on this list has a key matching item's
    //                 key
    // Postcondition:  No element has a key matching the argument item's key

    public void reset();
    // Effect:         Initializes current position for an iteration through this
    //                 list
    // Postcondition:  Current position is first element on this list

    public Listable getNextItem ();
    // Effect:         Returns a copy of the element at the current position on
    //                 this list and advances the value of the current position
    // Preconditions:  Current position is defined
    //                 There exists a list element at current position
    //                 No transformers have been called since most recent call to
    //                 reset
    // Postconditions: Return value = (a copy of element at current position)
    //                 If current position is the last element then current
    //                 position is set to the beginning of this list; otherwise,
    //                 it is updated to the next position
}
```

We define the `LinkedList` class as an abstract class. We place it, and the corresponding concrete classes, in a package called `ch05.genericLists`. As we did with the array-based `List` class, we include as much implementation detail as we can, without making any assumptions about whether or not the list is sorted. Let's start with the instance variables.

Just as in the implementations of the Stack and Queue ADTs, each node in a linked list must have at least two reference variables. The first contains a reference to a copy of the user's data; the second is a reference to the next element in the list. We again use an inner class, this time named `ListNode`, to provide the definition of our nodes. The `ListNode` class defines two reference variables: `info` of type `Listable` to hold the user's information, and `next` of type `ListNode` to hold the link to the next item in the list. The list itself is accessed through an instance variable called `list`, of type `ListNode`, which references the first element of the list.

In order to implement the List ADT, we need to record two pieces of information about the structure in addition to the list of items. The `lengthIs` operation returns the number of items in the list. In the array-based implementation, `lengthIs` simply returns the value of the `numItems` instance variable. This variable defines the extent of the list within the array. Therefore, the `numItems` variable must be present. In a link-based list we have a choice: We can keep a variable to hold the number of items or we can count the number of items each time the `lengthIs` operation is called. Keeping a `numItems` variable requires an addition operation each time `insert` is called and a subtraction operation each time `delete` is called. Which approach is better? We cannot determine it in the abstract; it depends on the relative frequency of use of the

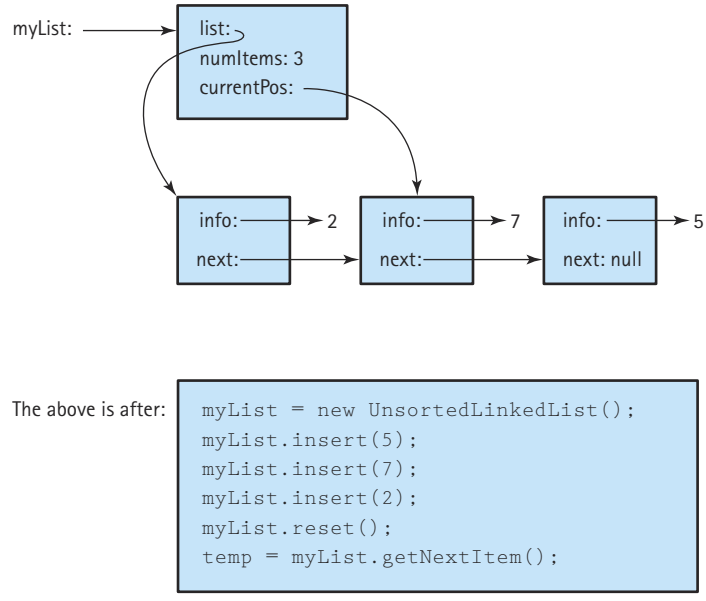


Figure 5.15 *List with three items (`reset` has been called; `getNextItem` has been called once)*

`lengthIs`, `insert`, and `delete` operations. Here, let's explicitly keep track of the length of the list by including a `numItems` variable in the `LinkedList` class.

The `reset` and `getNextItem` operations require that we keep track of the current position during an iteration, so we need a `currentPos` variable. In the array-based implementation, `currentPos` is an array index. What is the logical equivalent within a linked list? A reference to a `ListNode`, that is, to a `ListNode` object variable. Figure 5.15 pictures the structure of our list representation, assuming an unsorted list, after three items are inserted, `reset` is invoked, and `getNextItem` is invoked once. For clarity, we assume the list manipulates integers (although in actuality it must manipulate `Listable` objects).

We identified three instance variables (`list`, `numItems`, `currentPos`) to define in the `LinkedList` class. The class constructor must initialize these variables to values that represent an empty list. It sets the `numItems` variable to 0, and each of the reference variables to `null`. Here is the beginning of our `LinkedList` class.

```
//-----------------------------------------------------------------------
// LinkedList.java             by Dale/Joyce/Weems                  Chapter 5
//
// Defines all constructs for a reference-based list that do not depend on
//  whether or not the list is kept sorted
//-----------------------------------------------------------------------
```

```
package ch05.genericLists;

import ch04.genericLists.*;

public abstract class LinkedList implements ListInterface
{
  protected class ListNode
  // Used to hold references to list nodes for the linked list implementation
  {
    protected Listable info;        // The info in a list node
    protected ListNode next;        // A link to the next node on the list
  }

  protected ListNode list;          // Reference to the first node on the list
  protected int numItems;           // Number of elements in the list
  protected ListNode currentPos;    // Current position for iteration

  public LinkedList()
  // Creates an empty list object
  {
    numItems = 0;
    list = null;
    currentPos = null;
  }
}
```


Note the use of the `protected` modifier to provide access protection while allowing the variables to be inherited. Remember, `LinkedList` is an abstract class—it is extended by concrete classes that must be able to access the instance variables.

The isFull and lengthIs Methods

Now we must decide which of the methods required by the interface to implement as concrete methods, and which to leave abstract. We want to implement all the methods that do not depend on whether or not the list is sorted. For the array-based approach, we implemented `isFull`, `lengthIs`, `reset`, and `getNextItem`. We can again implement each of those methods at this level.

We handle the `isFull` method the same way we did for the reference-based implementations of the Stack and Queue ADTs. We simply return the value `false`. The only way the list could be full is if the program runs out of system space. In this case the Java run time system raises an exception anyway, so we decide just to rely on the run time systems built-in mechanisms.

```
public boolean isFull()
// Determines whether this list is full
{
  return false;
}
```

The `lengthIs` implementation is also very simple. Just return the value of the `numItems` instance variable. Remember, since we are supplying a `lengthIs` method, we do not need to supply an `isEmpty` method. The client can simply check if the length is 0, to see if the list is empty.

```
public int lengthIs()
// Determines the number of elements on this list
{
  return numItems;
}
```

The reset and getNextItem Methods

In Chapter 3 we used our list design terminology to describe algorithms for the `reset` and `getNextItem` operations. Since the design terminology was independent of the implementation approach, we can reuse it now. We repeat the algorithms here:

reset

Initialize currentPos to position of first list element

getNextItem: returns Listable

Set next to currentPos.info( )
Set currentPos to currentPos.next( )
return copy of next

Equivalent expressions for some of our list design notation, using an array-based and a reference-based approach, are shown in Table 5.1

Our list interface specification defines the "current position" to mean the location of the next element accessed during an iteration through the list. Recall that to be safe, we decided to reset the current position automatically, in the `getNextItem` method, when the end of the list is reached.

Table 5.1 *Comparing List Design Notation to Java Code*

Design Notation	Array-based	Reference-based
Initialize location to position of first list element	`location = 0`	`location = list`
Set location to location.next()	`location++`	`location = location.next`
location.info()	`list[location]`	`location.info`

In the array-based implementation, to initialize the current position, the `reset` method set `currentPos` to 0, since 0 was the index of the first element on the list. The corresponding action for the linked approach is to set the current position reference to the beginning of the list. The instance variable `list` always references the first element of the list. Therefore, the code for `reset` is:

```
public void reset()
// Initializes current position for an iteration through this list
{
  currentPos = list;
}
```

What happens if the list is empty? When the list is instantiated, the constructor sets the value of `list` to `null`. Let's be sure to maintain this property. That is, whenever the list is empty, in addition to the value of `numItems` being 0, we ensure that the value of `list` is `null`. Therefore, if we `reset` an empty list, the value of `currentPos` becomes `null`.

The `getNextItem` implementation is very similar to its array-based counterpart and follows directly from the generic algorithm. First, we make a copy of the item to be returned in a variable called `nextItemInfo`. It is of type `Listable`; in other words, it is an object of the class `Listable`. Those are the only objects we can manipulate on our lists.

Next, we increment the value of `currentPos`. Recall that the reference to the next item on the list is contained in the `next` reference of the current item. To update `currentPos` we use the following assignment statement, which is very similar to the algorithmic description:

```
currentPos = currentPos.next;
```

However, as we did in the array-based case, we must handle the special case of the current item being the last item on our list. The last item on the list is identifiable by the fact that its `next` reference contains the value `null`—it cannot reference anything since it is the end of the list. Therefore, we test the value of `next`. If it is `null` we reset the list by assigning `currentPos` the value held in `list`, just as we do in the `reset` method. See the code below.

Finally, the method returns the copy of the item it made in its first statement. The completed method looks like this:

```
public Listable getNextItem ()
// Returns copy of the next element in list
{
  Listable nextItemInfo = currentPos.info.copy();
  if (currentPos.next == null)
    currentPos = list;
  else
    currentPos = currentPos.next;

  return nextItemInfo;
}
```

Remember that we assume that the list is not empty ("Precondition: there exists a list element at current position") when `getNextItem` is called.

The Remaining Methods

We still need to implement four operations: `isThere`, `retrieve`, `insert`, and `delete`. In the case of array-based lists we used abstract methods for each of these operations, in the abstract class `List`. Do you remember why? It was because in each case we used different algorithms for the unsorted and sorted versions of the lists. Therefore, we delayed the concrete definitions of these methods until we defined the concrete classes `SortedList` and `UnsortedList`.

Let's review the differences in the method approaches between the sorted and unsorted array-based implementations. For `isThere` and `retrieve`, we used a linear search algorithm for the unsorted version and a binary search approach for the sorted version. For `insert` we had to ensure we inserted the item into the correct location in the sorted case, whereas in the unsorted case we just inserted the item at the "end" of the array. For `delete` we searched for the item to delete using a linear search in each case; but in the unsorted case we simply replaced the item to be deleted with the item in the last array slot, whereas in the sorted case we had to move all of the remaining list elements down one slot.

Are there similar differences in the reference-based approach? Let's look at each operation in turn:

- The `isThere` method—We can no longer use the binary search algorithm, even if the list is kept sorted, because there is no way for us to directly access the middle element of the list or sublists. Therefore, we must use a linear search approach. However, we can still create a slightly more efficient algorithm for the sorted version. In the case where the targeted element is not on the list, we can stop searching as soon as we reach an element that is greater than the targeted element. Since we have different algorithms for the sorted and unsorted cases, we define this method as abstract.
- The `retrieve` method—Again, we can no longer use the binary search algorithm. We use the linear search algorithm for both the sorted and the unsorted cases, and there are no improvements available for the sorted case. Since we know the element we are retrieving must be on the list, the trick of stopping our search when we reach an element that is too large does not apply. Since the algorithm we use is identical in the unsorted and sorted cases we implement `retrieve` as a concrete method in the abstract `LinkedList` class.
- The `insert` method—This is the case where it is easiest to see that a difference exists. In the unsorted case we can just insert the element anywhere, whereas in the sorted case we must insert it so that the list remains sorted. Since the two cases use different algorithms, we define this method as abstract.
- The `delete` method—In both cases (sorted and unsorted) we perform a linear search to find the element to be deleted. Once we find the element, we delete it by rearranging references. The deletion approach is the same whether or not the list is sorted, so we implement `delete` as a concrete class.

In summation, we define `isThere` and `insert` as abstract methods, leaving their implementation to the concrete classes presented in the next two sections. We implement both `retrieve` and `delete` within our abstract `LinkedList` class. Let's do `retrieve` first.

The retrieve Method

In Chapter 3, we did not introduce the `retrieve` operation until we covered generic lists. Therefore, we did not discuss how to implement `retrieve` for unsorted, array-based lists (although there is an exercise that asks you to do this). The required algorithm is similar to the one presented for the unsorted array-based `isThere` operation, except in this case we know that we eventually find the element, so we do not have to worry about falling off the end of the list. The algorithm for retrieving an item from an unsorted list is:

retrieve (item) from an unsorted list: returns Listable

```
Initialize location to position of first list element
Set found to false

while NOT found
   if item.compareTo(location.info( )) == 0
      Set found to true
   else
      Set location to location.next( )

return copy of location.info( )
```

Since we have no information about the order of the elements on the list, we must check each element until we find the one to return. If the comparison of our item to the element at the current location returns a 0, indicating equality, we set `found` to true and stop searching. Otherwise, we move to the next location.

In the case of the linked approach, sorted or unsorted, we are in the same situation. The only way to visit the elements on the list is to walk down the list one element at a time, just as we do in the above algorithm. Since we know from the preconditions that the item is on the list, there is no improvement available in the sorted case. Therefore, the algorithm for the linked implementation of `retrieve` (sorted or unsorted) is the same as the one presented above for the unsorted array-based implementation. Given the argument `item`, we traverse the list looking for a location where `item.compareTo(location.info())` returns 0, indicating that the

keys of the item and the list element are identical. Here is the referenced-based code that corresponds to the algorithm:

```
public Listable retrieve (Listable item)
// Returns a copy of the list element with the same key as item
{
  ListNode location = list;
  boolean found = false;

  while (!found)
  {
    if (item.compareTo(location.info) == 0)     // If they match
      found = true;
    else
      location = location.next;
  }

  return location.info.copy();
}
```

Figure 5.16 shows the contents of the various relevant constructs if we are trying to retrieve an item that is the second element of a list. The figure shows the situation before the *while* loop is entered, after the *while* loop is executed once, and after the *while* loop is executed twice. At that point, the *while* loop is exited and the method returns the required information.

The delete Method

In order to delete an item, we must first find it. We can do that using the same approach we used for the `retrieve` operation. However, as shown in Figure 5.16, when we use that approach, the `location` variable is left referencing the node that contains the item for which we are searching, the one to be removed. In order to actually remove it, we must change the reference in the previous node. That is, we must change the `next` reference of the previous node to the `next` reference of the one being deleted (see Figure 5.17).

Because we know from the specifications that the item to be deleted is in the list, we can change our search algorithm slightly. Rather than compare the item for which we are searching with the information in `location.info()`, we compare it with `location.next().info()`. Essentially, we are looking at the item in the position after `location`. When we find a match, we have references to both the previous node (`location`) and the one containing the item to be deleted (`location.next()`). We set the `next` reference of the previous node to the value of the `next` reference of the node to be deleted. This causes our links to "jump over" the node we wanted to delete, effectively deleting it from the list.

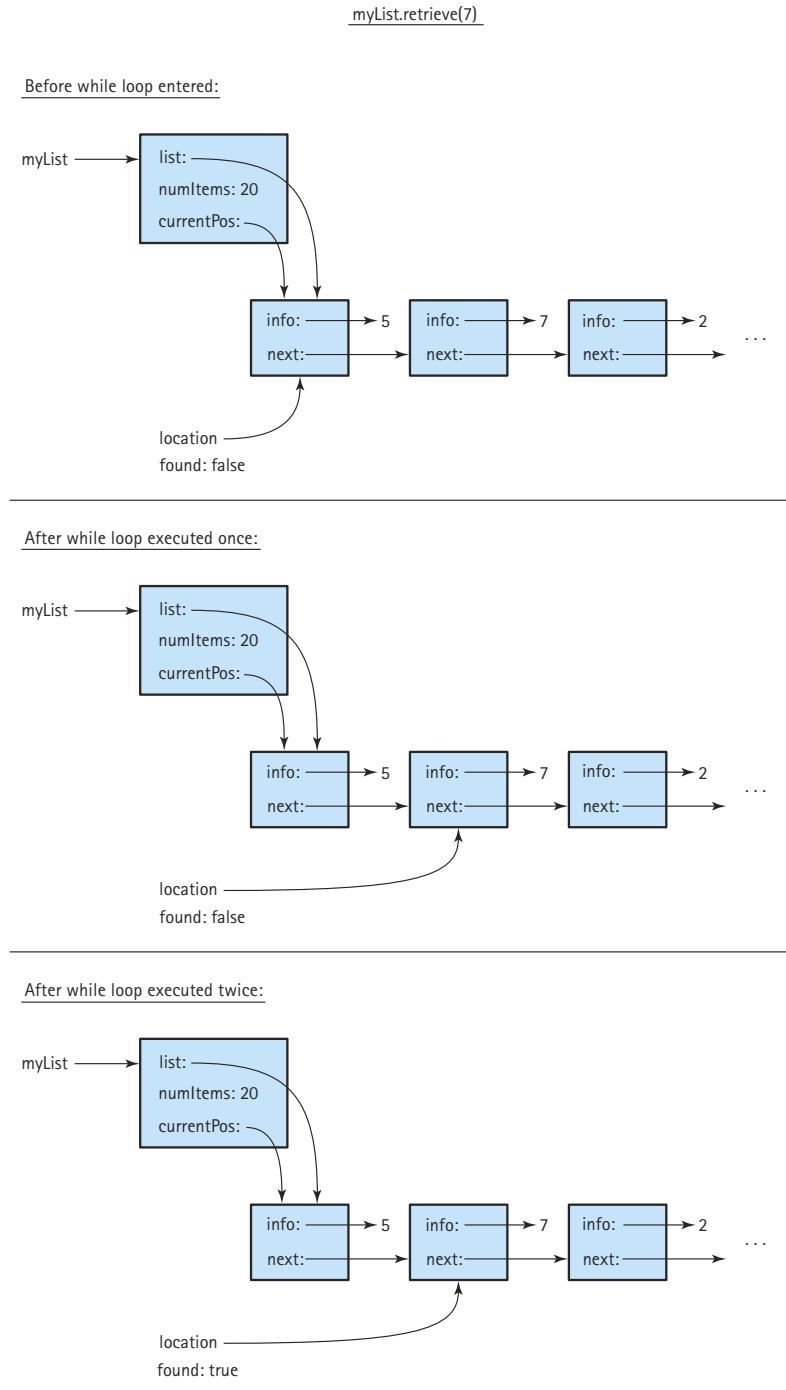


Figure 5.16 *Retrieving an item from a list*

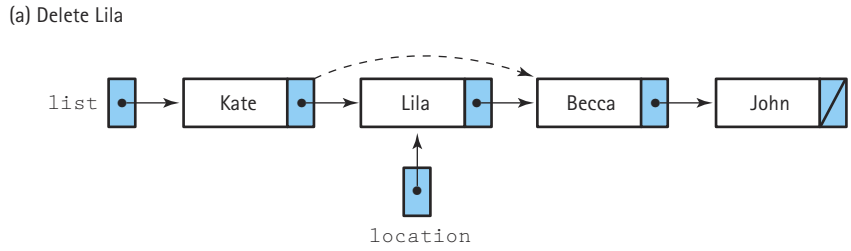


Figure 5.17 *Deleting an item from a list*

Note that removing the first node must be treated as a special case because the object's reference to the list (`list`) must be changed. We handle that special case with an *if*-statement at the beginning of our code. Is removing the last node a special case? No. The `next` reference of the node being deleted is `null` and it is stored into the `next` reference of the previous node, where it belongs.

Finally, we must remember to decrement the `numItems` instance variable since we are decreasing the size of the list. The code for the method is:

```
public void delete (Listable item)
// Deletes the element of this list whose key matches item's key
{
  ListNode location = list;

  // Locate node to be deleted
  if (item.compareTo(location.info) == 0)
    list = list.next;                          // Delete first node
  else
    {
    while (item.compareTo(location.next.info) != 0)
      location = location.next;
    // Delete node at location.next
    location.next = location.next.next;
    }
  numItems--;
}
```

The lists in the Java Library Collections Framework handle deletion in an alternative, interesting way. The delete operation is encapsulated with the list iterator methods. Clients can only delete an item when they are using an iterator object to walk through the list. They simply call the list's `delete` method, and whatever was the most recent object visited during the iteration is deleted. In this way, a client can iterate through a list one element at a time, examining elements to see if they want to delete them, in which case they just immediately invoke the `delete` operation.

5.4 Implementing the Unsorted List as a Linked Structure

The `LinkedList` class, developed in the previous section, looks like this (the abstract method signatures are emphasized):

```
//--------------------------------------------------------------------------
// LinkedList.java             by Dale/Joyce/Weems                  Chapter 5
//
// Defines all constructs for a reference-based list that do not depend on
//  whether or not the list is kept sorted
//--------------------------------------------------------------------------

package ch05.genericLists;

import ch04.genericLists.*;

public abstract class LinkedList implements ListInterface
{
  protected class ListNode
  // Used to hold references to list nodes for the linked list implementation
  {
    protected Listable info;       // The info in a list node
    protected ListNode next;       // A link to the next node on the list
  }

  protected ListNode list;         // Reference to the first node on the list
  protected int numItems;          // Number of elements in the list
  protected ListNode currentPos;   // Current position for iteration

  public LinkedList()
  // Creates an empty list object
  {
    numItems = 0;
    list = null;
    currentPos = null;
  }

  public boolean isFull()
  // Determines whether this list is full
  {
    return false;
  }
```

```
public int lengthIs()
// Determines the number of elements on this list
{
  return numItems;
}

public abstract boolean isThere (Listable item);
// Determines if element matching item is on this list

public Listable retrieve (Listable item)
// Returns a copy of the list element with the same key as item
{
  ListNode location = list;
  boolean found = false;

  while (!found)
  {
    if (item.compareTo(location.info) == 0)    // If they match
      found = true;
    else
      location = location.next;
  }

  return location.info.copy();
}

public abstract void insert (Listable item);
// Adds a copy of item to this list

public void delete (Listable item)
// Deletes the element of this list whose key matches item's key
{
  ListNode location = list;

  // Locate node to be deleted
  if (item.compareTo(location.info) == 0)
    list = list.next;                          // Delete first node
  else
    {
    while (item.compareTo(location.next.info) != 0)
      location = location.next;
    // Delete node at location.next
    location.next = location.next.next;
    }
```

```
    numItems--;
  }

  public void reset()
  // Initializes current position for an iteration through this list
  {
    currentPos  = list;
  }

  public Listable getNextItem ()
  // Returns copy of the next element in list
  {
    Listable nextItemInfo = currentPos.info.copy();
    if (currentPos.next == null)
      currentPos = list;
    else
      currentPos = currentPos.next;

    return nextItemInfo;
  }
}
```


We now extend this abstract class with two concrete classes, one that implements an unsorted list and one that implements a sorted list. In this section, we handle the simpler, unsorted version. In both cases, we only need to implement a constructor and the two abstract methods `isThere` and `insert`.

The setup for our `UnsortedLinkedList` class is very simple. We have to note that the class extends the `LinkedList` class. That lets us inherit the protected instance variables and public methods of that class. Since we cannot inherit constructors, we must implement a new constructor. It simply invokes the constructor of the `LinkedList` class.

For the `isThere` operation, we can reuse the `isThere` algorithm for the unsorted array-based list developed in Chapter 3:

isThere (item): returns boolean

Initialize location to position of first list element
Set found to false
Set moreToSearch to (have not examined last.info( ))

```
while moreToSearch AND NOT found
   if item.compareTo(location.info( )) == 0
      Set found to true
   else
      Set location to location.next( )
      Set moreToSearch to (have not examined last.info( ))

return found
```

Unlike for the `retrieve` method, we do not know if the item we are searching for is on the list. We protect our code from searching off the end of the list using the `moreToSearch` variable. The implementation of the algorithm is straightforward. It is presented with the rest of the code for the class below.

Finally, we implement the `insert` method. In the unsorted array-based implementation, we put the new item at the end because that was the easiest place to put it. What is the analogous place in the linked implementation? At the beginning of the list. Because the list is unsorted, we can put the new item wherever we choose, and we choose the easiest place: at the front of the list. In fact, `insert` is nearly identical to `push` in the linked stack implementation, although the list is implemented "by copy" and the stack "by reference." Note, we are not saying that inserting an item into an unsorted list is the same as pushing an item on a stack. The unsorted list and the stack are two entirely different data structures. We are saying, however, that the algorithms for the respective operations are the same.

Here is the code for the `UnsortedLinkedList` class:

```
//---------------------------------------------------------------------------
// UnsortedLinkedList.java          by Dale/Joyce/Weems                Chapter 5
//
// Completes the definition of a reference-based list under the assumption
//  that the list is not kept sorted
//---------------------------------------------------------------------------

package ch05.genericLists;

import ch04.genericLists.*;

public class UnsortedLinkedList extends LinkedList
{
  public UnsortedLinkedList()
  // Instantiates an empty list object
  {
    super();
  }
```

```
  public boolean isThere (Listable item)
  // Determines if element matching item is on this list
  {
    boolean moreToSearch;
    ListNode location = list;
    boolean found = false;

    moreToSearch = (location != null);

    while (moreToSearch && !found)
    {
      if (item.compareTo(location.info) == 0)  // If they match
        found = true;
      else
      {
        location = location.next;
        moreToSearch = (location != null);
      }
    }

   return found;
  }

  public void insert (Listable item)
  // Adds a copy of item to this list
  {
    ListNode newNode = new ListNode();
    newNode.info = (Listable)item.copy();
    newNode.next = list;
    list = newNode;
    numItems++;
  }
}
```

You can test the linked implementation of the Unsorted List ADT using the same test plan that was written for the array-based implementation.

Comparing Unsorted List Implementations

Now let's compare the array-based and linked implementations of the Unsorted List ADT. Just as we compared Stack and Queue ADT implementations, we look at two different factors: the amount of memory required to store the structure and the amount of work the solution does.

An array variable of the maximum list size takes the same amount of memory, no matter how many array slots are actually used, because we need to reserve space for the maximum possible. The linked implementation using dynamically allocated storage space requires only enough space for the number of elements actually in the list at run time.

However, as we discussed in detail when evaluating queue implementations, each node element is larger, because we must store the link (the `next` reference) as well as the user's data.

Again, we use Big-O notation to compare the execution efficiency of the two implementations. As mentioned before, most of the operations are nearly identical in the two implementations. The `isFull`, `reset`, and `getNextItem` methods in both implementations clearly have O(1) complexity. As with stacks and queues, the array-based constructor is O(N) but the reference-based constructor is only O(1).

The `lengthIs` method is always O(1) in an array-based implementation, but we have a choice in the linked version. We chose to make it O(1) by keeping a counter of the number of elements we insert and delete. If we had chosen to implement `lengthIs` by counting the number of elements each time the method is invoked, the operation would be O(N). The moral here is that you must know how an operation is implemented in order to specify its Big-O measure.

The `isThere` and `retrieve` operations are virtually identical for the two implementations. Beginning at the first element, they examine one element after another until the correct element is found. Because they must potentially search through all the elements in a list, both implementations are O(N).

Because the list is unsorted, we can choose to put a new item into a directly accessible place: the last position in the array-based implementation or the front in the linked version. Therefore, the complexity of `insert` is the same in both implementations: O(1). In both implementations, `delete` is O(N) because the list must be searched for the item to delete. These observations are summarized in Table 5.2. For those operations that

Table 5.2 *Big-O Comparison of Unsorted List Operations*

	Array Implementation	Linked Implementation
Class constructor	O(N)	O(1)
`isFull`	O(1)	O(1)
`lengthIs`	O(1)	O(1)
`isThere`	O(N)	O(N)
`reset`	O(1)	O(1)
`getNextItem`	O(1)	O(1)
`retrieve`		
Find	O(N)	O(N)
Process	O(1)	O(1)
Combined	O(N)	O(N)
`insert`		
Find	O(1)	O(1)
Insert	O(1)	O(1)
Combined	O(1)	O(1)
`delete`		
Find	O(N)	O(N)
Delete	O(1)	O(1)
Combined	O(N)	O(N)

require an initial search, we break the Big-O into two parts: the search and what is done following the search.

5.5 Implementing the Sorted List as a Linked Structure

To implement our Sorted List ADT as a linked structure we also extend the abstract class `LinkedList`. Again, we only need to implement a constructor and the two abstract methods `isThere` and `insert`.

The setup for our `SortedLinkedList` class is the same as for our `UnsortedLinkedList` class. Again we have the class extend the `LinkedList` class and add a constructor that just invokes the constructor of the `LinkedList` class.

The `isThere` method of the sorted array-based list implementation in Chapter 3 used the binary search algorithm. As we have already noted, we cannot use the binary search approach with linked lists. To search the list to see if a given item is on it, we must follow the links down the list one by one. We used the same approach for the unsorted linked list. However, we can improve our approach in one way. If the item is not on the list, we can stop searching when the element we are examining is larger than the item we are looking for. Since the list is sorted, we know that the rest of the elements on the list are larger still, and cannot possibly match our item.

When we encounter a list element larger than our target item, we set the `moreToSearch` variable to `false`. This same variable is set to `false` if the next location to be examined is `null`, that is, if we have reached the end of the list. Since there are two places in the code where we need to compare our item to the current list element (to see if they are equal and to see if the element is larger than the item), we declare an `int` variable `holdCompare`, to hold the result of the comparison. This way we only have to call the `compareTo` method once. Here is the code for `isThere`:

```
public boolean isThere (Listable item)
// Determines if element matching item is on this list
{
  int holdCompare;
  boolean moreToSearch;
  ListNode location = list;
  boolean found = false;

  moreToSearch = (location != null);

  while (moreToSearch && !found)
  {
    holdCompare = item.compareTo(location.info);
    if (holdCompare == 0)  // If they match
      found = true;
    else
```

```
      if (holdCompare < 0)   // If list element is larger than item
        moreToSearch = false;
      else
      {
        location = location.next;
        moreToSearch = (location != null);
      }
    }

  return found;
}
```

To complete the definition of the `SortedLinkedList` class we must define the `insert` operation. Inserting an item into a sorted list with a linked implementation is more complicated than any of the other operations we have implemented, due to the number of special cases to handle.

We base our approach on the algorithm identified in Chapter 3 for the array-based implementation. This algorithm has two parts: find the place to insert the item and insert the item. We can reuse the array-based approach for the first part of the algorithm, the part that walks through the list until we reach an element that is larger than our insertion item or we reach the end of the list. Remember that we assume that no current list element has a key value that matches the key value of our insertion item. That is why we search for an element that is "greater" than our item, rather than "greater than or equal to."

The second part of the Chapter 3 algorithm shifts the array elements between the insertion location and the end of the list down one position, to make room for the new item. With the linked approach, we know we don't have to shift any elements, so we should not use the second part of the algorithm. Here is the first part of the algorithm:

insert (item) – Locating insertion location

```
Set location to position of first element
Set moreToSearch to (have not examined last.info( ))
while moreToSearch
   if (item.compareTo(location.info( )) < 0)
      Set moreToSearch to false
   else
      Set location to location.next( )
      Set moreToSearch to (have not examined last.info( ))
```

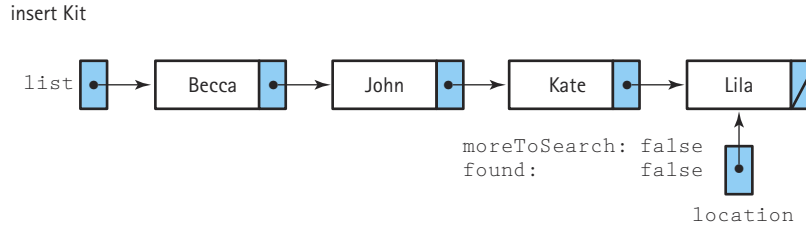


Figure 5.18 *Inserting an item into the "middle" of the list*

When we exit the loop, `location` is referencing the location where `item` goes, as shown in Figure 5.18. We just need to get a new node, put `item` into the `info` variable, put `location` into the `next` variable, and put the reference to the new node in the `next` variable of the node before it (the node containing Kate in the figure). Oops! We don't have a reference to the node before it. We must keep track of the previous reference as well as the current reference. When we had a similar problem with `delete` for the abstract `LinkedList` class, we compared one item ahead (`location.next.info`). Can we do that here? No. We were able to use that technique because we knew that the item for which we were searching was on the list. Here we know that the item for which we are searching is not on the list. If the new item were to go at the end of the list, that approach would crash because `location.next` would be `null`. (See Figure 5.19.)

We could change the way of determining `moreToSearch`, but there is an easier method for handling this situation. We use two references to search the list with one reference always trailing one node behind. We call the previous reference `prevLoc` ("previous location") and let it trail one node behind `location`. When `compareTo` returns an integer greater than 0, indicating that the element at `location` is still smaller than the insertion item, we advance both references. As Figure 5.20 shows, the process resembles the movement of an inchworm. The `prevLoc` reference (the tail of the inchworm) catches up with `location` (the head), and then `location` advances. Because there is not a node before the first node, we initialize `prevLoc` by setting it to `null`. We know we have reached the end of the list when `location` becomes `null`. Now let's summarize these thoughts into a new algorithm. We use some link-related notation in our algorithm, such as the term `null`, to simplify our transition to Java code later.

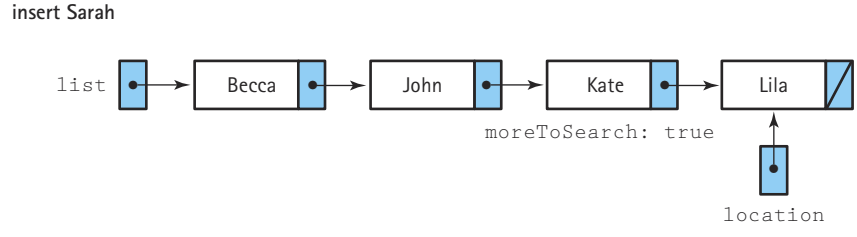


Figure 5.19 *Inserting at the end of the list*

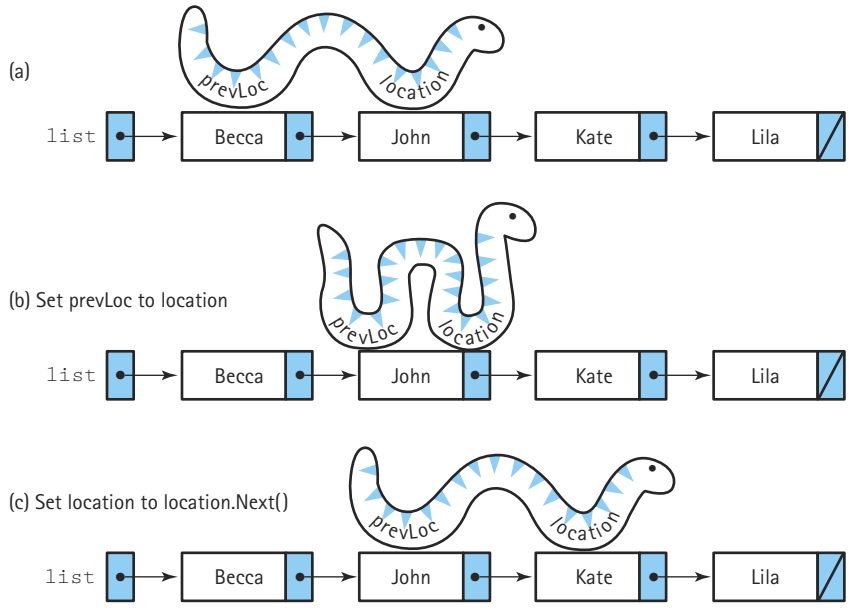


Figure 5.20 *The inchworm effect*

Let's do an algorithm walk-through before we code it. There are four cases, as shown in Figure 5.21: the new item goes before the first element, between two other

insert (item)

```
Set location to list
Set prevLoc to null
Set moreToSearch to (location != null)
while moreToSearch
   if (item.compareTo(location.info( )) < 0)
      Set moreToSearch to false
   else
      Set prevLoc to location
      Set location to location.next( )
      Set moreToSearch to (location != null)
Set newNode to a reference to a newly instantiated node
Set newNode.info( ) to copy of item
Set newNode.next( ) to location
Set prevLoc.next( ) to newNode
increment numItems
```

elements, comes after the last element, or is inserted into an empty list. In the first case (Figure 5.21a), "Alex" compared to "Becca" returns an integer less than 0, and we exit the loop. We store `location` into the `next` reference of `newNode` and `newNode` into the `next` reference of `prevLoc`. Whoops! The program crashes because `prevLoc` is `null`. Before setting the `next` reference of `prevLoc` we must check to see if `prevLoc` is `null`, and if it is, we must store `newNode` into `list` rather than the `next` reference of `prevLoc`. We amend the line of our algorithm that reads

```
Set prevLoc.next( ) to newNode
```

to read

```
if prevLoc == null
      Set list to newNode
else
      Set prevLoc.next( ) to newNode
```

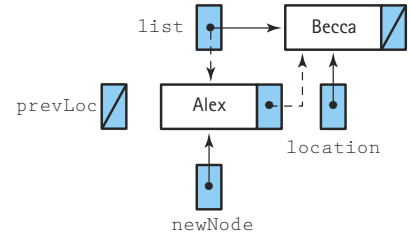
When you use a reference, remember to consider the possibility that it might be `null`.

What about the in-between case? Inserting "Kit" (Figure 5.21b) leaves `location` referencing the node with "Lila" and `prevLoc` referencing the node with "Kate." Therefore, the `next` reference of `newNode` references the node with Lila; the node with "Kate" references the new node. That's fine. What about when we insert at the end? Inserting "Kate" (Figure 5.21c) leaves `location` equal to `null` and `prevLoc` referencing the node with "Chris." Therefore, the `next` reference of `newNode` is assigned `null`, and the value of `newNode` (the reference to "Kate") is stored in the `next` reference of the node containing "Chris." That is also correct.

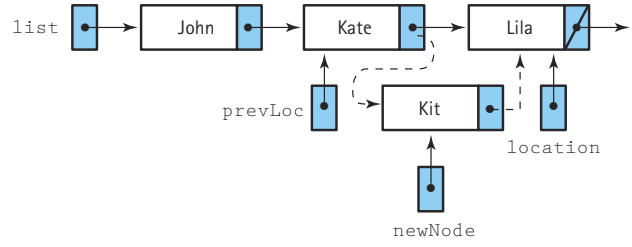
Does the algorithm work when the list is empty? Let's see. Both `location` and `prevLoc` are `null`, but we store `newNode` into `list` when `prevLoc` is `null` (remember how we updated our algorithm) so there isn't a problem. (See Figure 5.21d.)

Here is the code for the entire `SortedLinkedList` class, including the `insert` method.

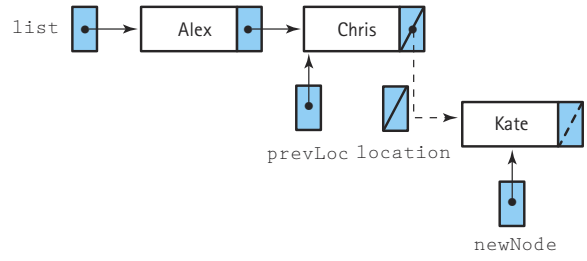
(a) insert Alex (goes at the beginning)



(b) insert Kit (goes in the middle)



(c) insert Kate (goes at the end)



(d) insert John (into an empty list)

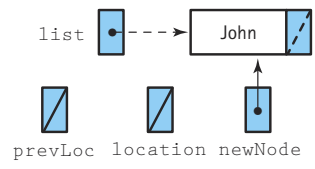


Figure 5.21 *Four insertion cases*

```
//-------------------------------------------------------------------------
// SortedLinkedList.java          by Dale/Joyce/Weems                Chapter 5
//
// Completes the definition of a link-based list under the assumption
//  that the list is kept sorted
//-------------------------------------------------------------------------

package ch05.genericLists;

import ch04.genericLists.*;

public class SortedLinkedList extends LinkedList
{
  public SortedLinkedList()
  // Instantiates an empty list object
  {
    super();
  }

  public boolean isThere (Listable item)
  // Determines if element matching item is on this list
  {
    int holdCompare;
    boolean moreToSearch;
    ListNode location = list;
    boolean found = false;

    moreToSearch = (location != null);

    while (moreToSearch && !found)
    {
      holdCompare = item.compareTo(location.info);
      if (holdCompare == 0)  // If they match
        found = true;
      else
      if (holdCompare < 0)   // If list element is larger than item
        moreToSearch = false;
      else
      {
        location = location.next;
        moreToSearch = (location != null);
      }
    }

   return found;
  }
```

```
  public void insert (Listable item)
  // Adds a copy of item to list
  {
    ListNode newNode = new ListNode();     // Reference to node being inserted
    ListNode prevLoc = new ListNode();     // Trailing reference
    ListNode location = new ListNode();    // Traveling reference
    boolean moreToSearch;

    location = list;
    prevLoc = null;
    moreToSearch = (location != null);

    // Find insertion point
    while (moreToSearch)
    {
      if (item.compareTo(location.info) < 0) // List element is larger than item
        moreToSearch = false;
      else
      {
        prevLoc = location;
        location = location.next;
        moreToSearch = (location != null);
      }
    }

    // Prepare node for insertion
    newNode.info = (Listable)item.copy();

    // Insert node into list
    if (prevLoc == null)
    // Insert as first
    {
      newNode.next = list;
      list = newNode;
    }
    else
    {
      newNode.next = location;
      prevLoc.next = newNode;
    }
    numItems++;
  }
}
```

Comparing Sorted List Implementations

As for the unsorted list, it is easy to see that for both array-based and linked approaches, the `isFull`, `lengthIs`, `reset`, and `getNextItem` methods are all O(1); that is, they require a constant number of steps. Also, the constructor for the former is O(N) and the constructor for the latter is O(1).

We discussed three algorithms for the `isThere` operation in an array-based list: a sequential search, a sequential search with an exit when the place is passed where the item would be if present, and a binary search. The first two have order O(N); the binary search has order O($\log_2 N$). The first two searches can be implemented in a linked list, but a binary search cannot. (How do you get directly to the middle of a linked list?) Therefore, the array-based algorithm for searching a list is faster than the linked version if the binary search algorithm is used.

Retrieving an item, inserting an item, and deleting an item all require, as a first step, finding the correct location in the list to perform the operation. Any of the algorithms identified for the `isThere` operation can be used to find the location. Once the correct location is determined, the `retrieve` method, in both implementations, simply has to return a copy of the item at the location. This step is O(1). Therefore, the total Big-O complexity of `retrieve` depends entirely on the algorithm used to find the correct location. For the array-based implementation this is either O(N) or O($\log_2 N$). For the link-based implementation this is O(N).

In both list implementations, the `insert` method uses a sequential search to find the insertion position; therefore, the search parts of the algorithms have O(N) complexity. As just pointed out, the binary search algorithm, with O($\log_2 N$) complexity, could be used for the array-based implementation. For the array-based `list` we must also move down all the elements that follow the insertion position to make room for the new element. The number of elements to be moved ranges from 0, when we insert at the end of the list, to `numItems`, when we insert at the beginning of the list. So the insertion part of the algorithm also has O(N) complexity for the array-based list. Because O(N) + O(N) = O(N), the array-based list's `insert` operation is O(N). Even if we used the binary search to find where the item belongs (O($\log_2 N$)), the items would have to be moved to make room for the new one (O(N)). O($\log_2 N$) + O(N) is also O(N).

The insertion part of the algorithm for the linked list representation simply requires the reassignment of a couple of references. This makes the insertion task O(1) for a linked list, which is one of the main advantages of linking. However, adding the insertion task to the search task gives us O(N) + O(1) = O(N)—the same Big-O approximation as for the array-based list! Doesn't the linking offer any advantage in efficiency? Perhaps. Remember that the Big-O evaluations are only rough approximations of the amount of work that an algorithm does.

The `delete` method is similar to `insert`. In both implementations, the search task is performed as a O(N) operation. Then the array-based list's `delete` operation "deletes" the element by moving up all the subsequent elements in the list, which adds O(N). The whole function is O(N) + O(N), or O(N). The linked list deletes the element by unlinking it from the list, which adds O(1) to the search task. The whole function is O(N) + O(1), or O(N). Thus, both `delete` operations are O(N); for large values of N, they are roughly equivalent.

Table 5.3 *Big-O Comparison of Sorted List Operations*

	Array Implementation	Linked Implementation
Class constructor	O(*N*)	O(1)
`isFull`	O(1)	O(1)
`lengthIs`	O(1)	O(1)
`isThere`	O(*N*)*	O(*N*)*
`reset`	O(1)	O(1)
`getNextItem`	O(1)	O(1)
`retrieve`		
Find	O(*N*)*	O(*N*)
Process	O(1)	O(1)
Combined	O(*N*)*	O(*N*)
`insert`		
Find	O(*N*)*	O(*N*)
Insert	O(*N*)	O(1)
Combined	O(*N*)	O(*N*)
`delete`		
Find	O(*N*)*	O(*N*)
Delete	O(*N*)	O(1)
Combined	O(*N*)	O(*N*)

*$O(\log_2 N)$ if a binary search is used.

The fact that two operations have the same Big-O measure does not mean that they take the same amount of time to execute. The array-based implementation requires, on average, a great deal of data movement for both insert and delete operations. Does all this data movement really make any difference? It doesn't matter too much in the examples in our figures; the lists are all very small. If there are 1,000 elements on the list, however, the data movement starts to add up.

Table 5.3 summarizes the Big-O comparison of the sorted list operations for array-based and linked implementations

5.6 Our List Framework

In this chapter, we further refined our list framework. The framework consists of the features of our unsorted and sorted lists, the assumptions, the supported operations, and the set of classes and interfaces that support everything else. Let's review our framework.

First, we review the features and supported operations of our lists. In Chapter 3, we defined our general list ADT. We decided that our lists would

- be keyed lists.
- contain unique items in terms of the keys.
- hold objects of classes that implement the `Listable` interface.
- use the "by copy" approach; in other words, insertions would place copies of the client's data onto the list, and retrievals would return copies of the items on the list.
- support the operations `isFull`, `lengthIs`, `isThere`, `retrieve`, `insert`, `delete`, `reset`, and `getNextItem`.
- operate by contract in terms of assuming the client would never insert a duplicate item, insert into a full list, request deletion or retrieval of a nonexistent item, or try to iterate through an empty list.

Next, we review the classes and interfaces that support these features:

- **The `Listable` interface:** Only objects of classes that implement this interface can be used with our lists
- **The `ListInterface` interface:** Defines the method signatures of all the supported list operations; the detailed comments include the effect, preconditions, and postconditions for each operation
- **The abstract class `List`:** Implements `ListInterface` and defines all the constructs for an array-based list that do not depend on whether or not the list is sorted
- **The abstract class `LinkedList`:** Implements `ListInterface` and defines all the constructs for a link-based list that do not depend on whether or not the list is sorted; includes the definition of the `ListNode` class as an inner class
- **The `UnsortedList` class:** Extends `List` and completes its definition under the assumption that the list is not kept sorted (*Note:* the creation of this class was left as an exercise.)
- **The `SortedList` class:** Extends `List` and completes its definition under the assumption that the list is kept sorted
- **The `UnsortedLinkedList` class:** Extends `LinkedList` and completes its definition under the assumption that the list is not kept sorted
- **The `SortedLinkedList` class:** Extends `LinkedList` and completes its definition under the assumption that the list is kept sorted

Figure 5.22 displays a UML static class diagram that captures the relationships among these classes and interfaces.

Please remember that this list framework is not the only possible framework. We believe it is a sound logical framework for the purposes of this textbook. However, there are many other viable approaches to supporting list abstractions using Java. For example, the lists that are part of the Java 2 Library's Collection Framework are much different from those we created.

We could define lists with different features. For example, we could drop the assumption that the lists contain unique elements—although the class and interface relationships and basic algorithms would not change, the implementation details of many

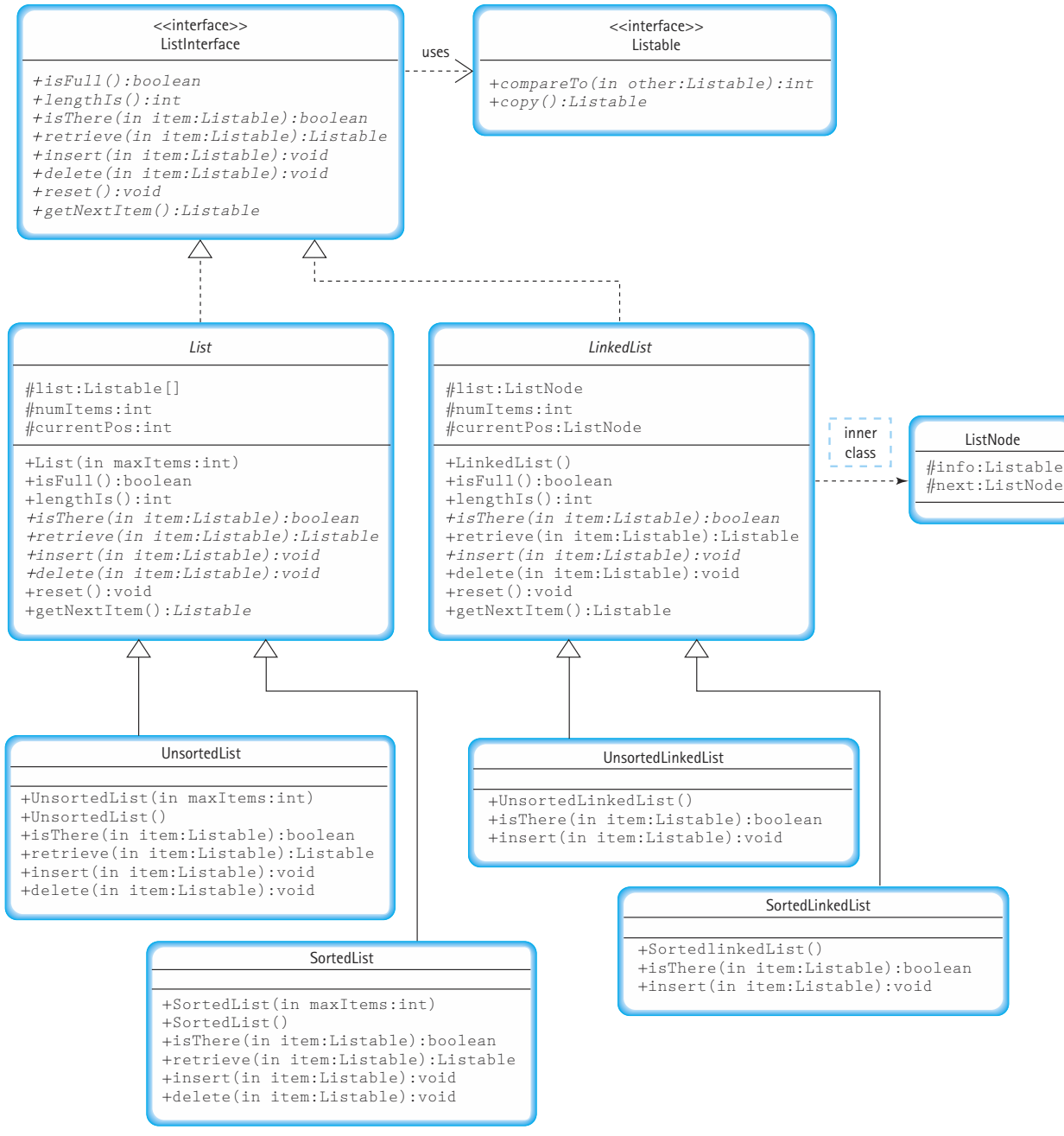


Figure 5.22 *UML class diagram for our list framework*

of the operations would be affected. As another example, we might drop the programming-by-contract approach and use Java's *exception* mechanism to enforce the stated preconditions. In that case they are no longer preconditions, they are part of the definition of the method interface.

Additionally, we could create a different class/interface architecture. Perhaps, instead of using abstract classes to factor out the commonality from the sorted and unsorted approaches, we could directly implement the `ListInterface` with concrete classes. Or maybe we could create one abstract class for unsorted lists and another for sorted lists, rather than one for the array-based approach and one for the link-based approach.

We developed the `Listable` interface to ensure that the objects we placed on our lists supported the `copy` and `compareTo` operations. Instead of creating our own interface, we could have used two of Java's predefined interfaces, the `Cloneable` interface and the `Comparable` interface. The `Cloneable` interface has a unique protocol—declaring that a class implements `Cloneable` amounts to a promise by the implementer that clients of the class can safely invoke a `clone` method on objects of the class. The class either provides its own reliable `clone` method or the implementer of the class is guaranteeing that the `Object` class's `clone` method (a bitwise copy operation, that is, an exact copy of the object's memory representation) can be used. The use of the `Cloneable` interface is not very intuitive, so we decided to define our own `Listable` interface.

Summary

We have seen how stacks, queues, unsorted lists, and sorted lists may be represented in an array-based or linked representation. The specifications for each ADT didn't mention design representation, so we were free to implement them in any way we chose. There was nothing in the specification of these ADTs to say that the structures should be array-based or linked, or that the elements were stored in statically or dynamically allocated storage.

We could specify a number of other operations for a List ADT. Some operations, such as one to find the preceding node in a list, are easy to implement for an array-based list but would be difficult to implement using a list that is linked in one direction (like the lists in this chapter). This operation would be simpler if the list had links going both forward and backward. We can think of many variations for representing a linked list in order to simplify the kinds of operations that are specified for the list: doubly linked lists, circular lists, lists that are accessed from both the beginning and the end. We look at some of these alternative implementation structures in the next chapter.

The idea of linking the elements in a data structure is not specific to stacks, queues, and lists. We use this powerful tool to implement trees and graphs later in this book.

Summary of Classes and Support Files

The classes and files are listed in the order in which they appear in the text. The package a class belongs to, if any, is listed in parentheses under Notes. The class and support files are available on our web site. They can be found in the `ch05` subdirectory of the `bookFiles` directory.

Note: The `StackNode`, `QueueNode`, and `ListNode` classes are all defined as inner classes. They do not exist as separate files. Therefore, they are not included in the table below, which list constructs by file.

Classes, Interfaces and Support Files Defined in Chapter 5

File	1st Ref.	Notes
`LinkedStack.java`	page 347	(`ch05.stacks`) Reference-based implementation of our stack ADT; implements `StackInterface`
`LinkedQueue.java`	page 357	(`ch05.queues`) Reference-based implementation of our queue ADT; implements `QueueInterface`
`LinkedList.java`	page 371	(`ch05.genericLists`) Abstract class—defines all the constructs for a reference-based generic list that do not depend on whether or not the list is sorted; the list stores objects derived from a class that implements `Listable`; implements `ListInterface`
`UnsortedLinkedList.java`	page 383	(`ch05.genericLists`) Extends `LinkedList` under the assumption that the list is *not* kept sorted
`SortedLinkedList.java`	page 392	(`ch05.genericLists`) Extends `LinkedList` under the assumption that the list *is* kept sorted

Exercises

5.1 Implementing a Stack as a Linked Structure

1. True or False? Explain your answers.
   a. An array is a random-access structure.
   b. An array-based list is a random-access structure.
   c. A linked list is a random-access structure.
   d. An array-based list is always stored in a statically allocated structure.
   e. A stack is not a random-access structure.
2. What is the main difference in terms of memory allocation, between using an array-based stack and using a reference-based stack?
3. What is a self-referential class? We provided three examples of self-referential classes in this chapter—what are they and how are they used?
4. Consider the code for the `push` method on page 350. What would be the effect of the following changes to that code?
   a. Switch the first and second lines.

b. Switch the second and third lines.

c. Switch the second and fourth lines.

5. We decide to add a new operation to our Stack ADT called `popTop`. We add the following code to our `StackInterface` interface

```
public Object popTop() throws StackUnderflowException;
// Effect:         Removes top item from this stack and returns it
// Postconditions: If (this stack is empty)
//                   an exception that communicates 'pop on stack
//                   empty' is thrown
//                 else
//                   top element has been removed from this stack.
//                   return value = (the removed element)
```

An operation like this is often included for stacks. Implement the `popTop` method for the `LinkedStack` class.

6. Suppose we decide to add a new operation to our Stack ADT called `sizeIs`, which returns a value of primitive type `int` equal to the number of items on the stack. The method signature for `sizeIs` is:

```
public int sizeIs()
```

a. Write the code for `sizeIs` for the `ArrayStack` class defined in Chapter 4.

b. Write the code for `sizeIs` for the `LinkedStack` class defined in this chapter (do not add any instance variables to the class; each time `sizeIs` is called you must "walk" through the stack and count the nodes).

c. Augment the `LinkedStack` class with an instance variable `size` that always holds the current size of the stack. Now you can implement the `sizeIs` operation by just returning the value of `size`. Identify all the methods of `LinkedStack` that you need to modify to maintain the correct value in the `size` variable and describe how you would change them.

d. Analyze the methods created/changed in Parts a, b, and c in terms of Big-O efficiency.

5.2 Implementing a Queue as a Linked Structure

7. Consider the code for the `enqueue` method on page 360. What would be the effect of the following changes to that code?

a. Switch the second and third lines.

b. Change the boolean expression "`rear == null`" to "`front == null`".

8. Given the following specification of a `front` operation for a queue:

Effect: Returns a reference to the front item on the queue.

Precondition: Queue is not empty.

Postconditions: Return value = a reference to the front item on the queue.
Queue is not changed.

a. Write this operation as client code, using operations from the `LinkedQueue` class. (Remember the client code has no access to the private variables of the class.)

b. Write this operation as a new public method of the `LinkedQueue` class.

9. Assume that an integer requires 2 bytes of space, and that a reference requires 4 bytes of space. Also assume the maximum queue size is 200. For queues of integers:

   a. How much space would our array-based queue of size 20 require?

   b. How much space would our array-based queue of size 100 require?

   c. How much space would our array-based queue of size 200 require?

   d. How much space would our reference-based queue of size 20 require?

   e. How much space would our reference-based queue of size 100 require?

   f. How much space would our reference-based queue of size 200 require?

   g. For what size queue do the array-based and reference-based approaches use approximately the same amount of space?

10. Implement the Queue ADT using a circular linked list as discussed in the section. Test your implementation using the test plan developed in Chapter 4.

5.3 An Abstract Linked List Class

11. You have studied both array-based and reference-based (linked) lists.

    a. Explain the difference between an array-based and a linked representation of a list.

    b. Give an example of a problem for which an array-based list would be the better solution.

    c. Give an example of a problem for which a linked list would be the better solution.

12. A friend of yours is having trouble instantiating an object of the `LinkedList` class and asks for your help. What do you tell your friend?

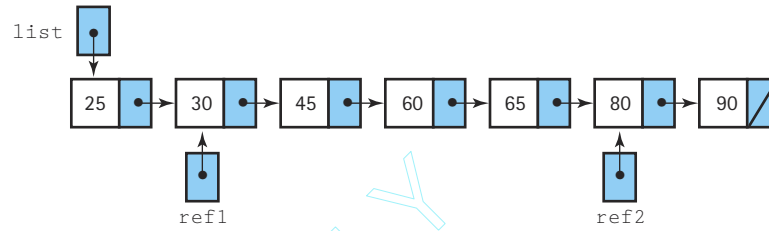
13. The `List` class and `LinkedList` class are both abstract classes that implement the `ListInterface` interface. Other than the fact that the former is based on arrays and the latter is based on references, what are some important differences between these two classes?

14. Rewrite the `lengthIs` method of the `LinkedList` class assuming that we do not keep track of the current size of the list in an instance variable.

15. Suppose we redefine the `delete` operation by dropping the precondition that the item to be deleted is definitely on the list; furthermore, we have the `delete` operation return a `boolean` value indicating whether or not it successfully deleted the item. Write this new `delete` operation for the `LinkedList` class.

16. Suppose we redefine our list so that it can contain duplicate items. We now define the `delete` operation so that it deletes every matching element from the list (we still assume that at least the element matches). Write this new `delete` operation for the `LinkedList` class.

The next three questions use the following diagram. In the diagram a reference is indicated by an arrow, the list nodes have an `info` variable containing an integer and a `next` variable containing a reference, and `list`, `ref1`, and `ref2` are references to a list node.



17. Give the values of the following expressions:
    a. `ref1.info`
    b. `ref2.next.info`
    c. `list.next.next.info`
18. Are the following expressions true or false?
    a. `list.next == ref1`
    b. `ref1.next.info == 60`
    c. `ref2.next == null`
    d. `list.info == 25`
19. Write one statement to do each of the following:
    a. Make `list` point to the node containing 45.
    b. Make `ref2` point to the last node in the list.
    c. Make `list` point to an empty list.
    d. Set the `info` variable of the node containing 45 to 60.

5.4 Implementing the Unsorted List as a Linked Structure

20. Following the style of the figures in this chapter, draw the list that would result from each of the following code sequences. To simplify the presentation of this exercise, we assume our lists hold integers rather than objects of the class `Listable`.
    a.
```
UnsortedLinkedList myList = new UnsortedLinkedList();
myList.insert(5);
myList.insert(9);
myList.insert(3);
```
    b.
```
UnsortedLinkedList myList = new UnsortedLinkedList();
myList.insert(5);
myList.insert(9);
```

```
myList.insert(3);
myList.delete(9);
```

21. Suppose we have a class called `Car` that models cars; it has instance variables for year, make, model, and price; it provides appropriate observer methods, including one called `getPrice` that returns a value of type `int` indicating the price of "this" car. The `Car` class also implements the `Listable` interface, so we can create a list of cars.

    Implement a client method `totalPrice`, that accepts a list (`UnsortedLinkedList carList`) of cars and returns an integer equal to the total cost of the cars on the list.

22. Extend our `UnsortedLinkedList` class with a public method `endInsert`, which inserts an item onto the end of the list. Do not add any instance variables to the class. The method signature is:

```
public void endInsert(Listable item)
```

23. Extend our `UnsortedLinkedList` class with a public method `ceilingList`, which returns a new list that contains all the elements of the current list that are less than or equal to the item argument. The method signature is:

```
public UnsortedLinkedList ceilingList(UnsortedLinkedList list,
                                      Listable item)
```

5.5 Implementing the Sorted List as a Linked Structure

24. Following the style of the figures in this chapter, draw the list that would result from each of the following code sequences. To simplify the presentation of this exercise, we assume our lists hold integers rather than objects of the class `Listable`.

    a.
    ```
    SortedLinkedList myList = new SortedLinkedList();
    myList.insert(5);
    myList.insert(9);
    myList.insert(3);
    ```

    b.
    ```
    SortedLinkedList myList = new SortedLinkedList();
    myList.insert(5);
    myList.insert(9);
    myList.insert(3);
    myList.delete(9);
    ```

25. Extend our `SortedLinkedList` class with a public method `ceilingList`, which returns a new list that contains all the elements of the current list that are less than or equal to the item argument. The method signature is:

```
public UnsortedLinkedList ceilingList(UnsortedLinkedList list,
                                      Listable item)
```

26. Consider the operation of merging together two sorted linked lists, `list1` and `list2`, into a single sorted linked list, `list3`. Suppose `list1` is size M, and `list2` is size N.

a. Suppose you implemented this operation at the client level by using an iterator to obtain each of the elements of `list1` and insert them into `list3`, and then using an iterator to obtain each of the elements of `list2`, using `isThere` to make sure they were not already in `list1`, and if they were not, using `insert` to insert them into `list3`. What is the Big-O complexity of this approach? (Remember to count the time taken by the list methods.)

b. Another approach is to implement the operation as a public method of the `SortedLinkedList` class. The start of the method would be

```
public void merge (SortedLinkedList list1, SortedLinkedList list2)
// Effect:          This list becomes the merger of list1 and list2
// Postconditions: Any item on list1 is also on this list
//                  Any item on list2 is also on this list
//                  No other items are on this list
//                  This list is sorted
//                  There are no duplicate items on this list
```

Describe an algorithm for implementing this method. What is the Big-O complexity of your algorithm?

5.6 Our List Framework

27. For our list framework, identify and describe the main purpose of

a. Each of the interfaces

b. Each of the abstract classes

c. The inner class

d. Each of the concrete classes

28. Recreate the structure of our list framework diagram; that is, indicate the interfaces, abstract classes, inner classes, and concrete classes and their relationships; do not include details such as instance variable and method names.

Chapter 6

Lists Plus

Goals

Measurable goals for this chapter include that you should be able to

- implement a circular linked list
- implement a doubly linked list
- implement a linked list with a header node or a trailer node or both
- implement a linked list as an array of nodes
- describe the benefits and drawbacks of each of our list approaches
- explain the relationships among the classes and interfaces of our list framework
- choose a reasonable list approach based on a set of list requirements
- if needed, define a new list approach to help solve a specified problem

This chapter begins with three new implementations of reference-based lists: circular linked lists, doubly linked lists, and lists with headers and trailers. We then introduce an array-based approach to implementing a linked list. This implementation is widely used in operating systems software.

All of the lists presented to this point are generic; that is, they are designed to be useful for many applications. At the end of this chapter we design a list ADT with very different properties for a specific application. In the case study, we design and implement the application.

Note that Figure 6.20, in the chapter summary section, features the additions to our list framework made in this chapter.

6.1 Circular Linked Lists

The linked lists that we implemented in Chapter 5 are characterized by a linear (line-like) relationship between the elements: Each element (except the first one) has a unique predecessor, and each element (except the last one) has a unique successor.

Let's consider a small change to our linked list approach and see how much it would affect our implementation and use of the Sorted List ADT. Suppose we change the linear list slightly, making the `next` reference of the last node point back to the first node instead of containing `null` (Figure 6.1). Now our list is a circular linked list rather than a linear linked list. We can start at any node in the list and traverse the whole list.

Circular linked list A list in which every node has a successor; the "last" element is succeeded by the "first" element

Of course, we must now ensure that all of our list operations maintain this new property of the list: that after the execution of any list operation, the last node continues to point to the front node. A quick consideration of each of the operations should convince us that we could continue to efficiently support all of them except when an operation changes the first element on the list. Consider, for example, if we try to delete the first element. Our previous delete approach would simply change the `list` reference to point to the second element on the list, effectively removing the first element. Now, however, we must also update the reference in the last element on the list, so that it points to the new first element. The only way to do that is to traverse the entire list to obtain access to the last element, and then make the change. A similar problem arises if we insert an item into the front of the list.

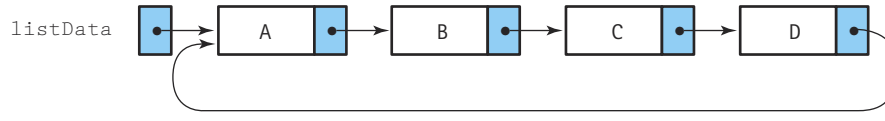


Figure 6.1 *A circular linked list*

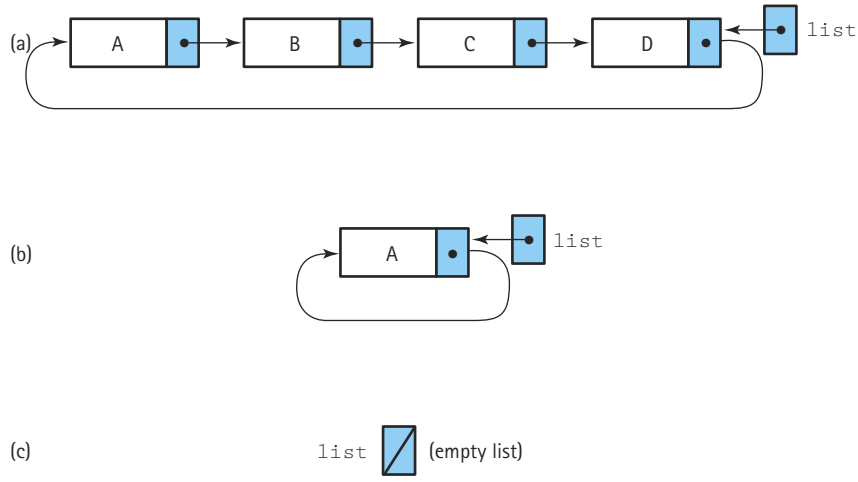


Figure 6.2 *Circular linked lists with the external pointer pointing to the rear element*

Inserting and deleting elements at the front of a list might be a common operation for some applications. Our linear linked list approach supported these operations very efficiently, but our circular linked list approach does not. We can fix this problem by letting our `list` reference point to the last element in the list rather than the first; now we have direct access to both the first and the last elements in the list (See Figure 6.2 where `list.info` references the information in the last node, and `list.next.info` references the information in the first node.) We mentioned this type of list structure in Chapter 5, when we discussed circular linked queues.

The CircularSortedLinkedList Class

In order to implement a list class using our new approach we must decide where it fits in our list framework. Certainly, we expect the new class to implement the `ListInterface` interface. The question is whether it should implement the interface directly or whether it should extend one of our current classes that already implements the interface. Perhaps we can reuse some of the methods already defined in the `LinkedList` class or the `SortedLinkedList` class. To resolve this question, let's consider more carefully the effect that our structure change has on our implementation.

There is no need to change any of the declarations in the inner class `ListNode` to make the list circular, rather than linear. After all, the design of the nodes is the same; only the value of the `next` reference of the last node has changed. Also, we can continue to use the same instance variables as before, since we still need a reference into the list, a count of the number of elements on the list, and a current position indicator for list iterations. The list constructor does not change since the reference variables are still initialized to `null`, and the number of elements on the original empty list is still zero.

How does the circular nature of the list change the implementations of the list operations? Three methods can remain unchanged. We continue to follow the convention

that a reference-based list is never "full;" the `isFull` method remains unchanged. The `lengthIs` method still needs to just return the value of `numItems`. And since the `retrieve` method asserts as a precondition that the targeted item is on the list, its code can remain the same. We do not have to worry about "looping" back to the beginning of the list since we are guaranteed to find the targeted item before that could occur.

All of the other methods require some modification. For the `isThere` method we must change the search termination condition since the end of the list is no longer marked by `null`. The `insert` and `delete` methods change the structure of the list so we must make sure they maintain the circular nature of the list; and the iterator methods need to be updated to reflect the new structure of list elements.

Considering that we can reuse the declarations, constructor, and three methods of the `LinkedList` class, we decide to implement our new list as an extension of the abstract class `LinkedList`. We call our new class `CircularSortedLinkedList`; its header is:

```
public class CircularSortedLinkedList extends LinkedList
```

Following our previous conventions, we create a new package, `ch06.genericLists`, to hold this class and the other generic list constructs defined in this chapter. Obviously, our new class requires access to the `LinkedList` class. Normally, we would just import the package containing the `LinkedList` class (`ch05.genericLists`) into the new class. However, simply importing the package does not allow our new class the privilege of accessing the attributes of the `ListNode` inner class of the `LinkedList` class. This privilege is only accorded to classes in the same package as `LinkedList`. Since access to these attributes is necessary to code the new methods, we make a copy of the `LinkedList` class file and place it in the `ch06.genericLists` package. A copy of the corresponding file can be found in the appropriate subdirectory (`bookfiles.ch06.genericLists`) on our web site.

The beginning of the new class definition looks like this:

```
//--------------------------------------------------------------------------
// CircularSortedLinkedList.java      by Dale/Joyce/Weems            Chapter 6
//
// Completes the definition of a link-based list under the assumption
//  that the list is circular and is kept sorted
//--------------------------------------------------------------------------

package ch06.genericLists;

import ch04.genericLists.*;

public class CircularSortedLinkedList extends LinkedList
{
  public CircularSortedLinkedList()
  // Instantiates an empty list object
  {
    super();
  }
```

Next we look at the changes to the remaining methods. Let's begin with the easy ones.

The Iterator Methods

The required changes here are interesting in that the `reset` method becomes more complicated and the `getNextItem` method becomes simpler. Here's the code for the linear linked list and circular linked list, side-by-side, for easy comparison:

Linear

```
public void reset()
// Initializes current position for
//  an iteration through this list
{
  currentPos = list;
}

public Listable getNextItem ()
// Returns copy of the next element
{
  Listable nextItemInfo =
     currentPos.info.copy();
  if (currentPos.next == null)
    currentPos = list;
  else
    currentPos = currentPos.next;

  return nextItemInfo;
}
```

Circular

```
public void reset()
// Initializes current position for
//  an iteration through this list
{
  if (list == null)
    currentPos = list;
  else
    currentPos = list.next;
}

public Listable getNextItem ()
// Returns copy of the next element
{
  Listable nextItemInfo =
   currentPos.info.copy();
  currentPos = currentPos.next;

  return nextItemInfo;
}
```

Since we want the `reset` method to set the current position to the start of the list and our `list` reference variable points to the last list element, we must access the start of the list through the `list.next` reference. However, if the list is empty, this reference does not exist. In that case, the `list` variable itself holds the value `null`, so we cannot access `list.next`. Therefore, we must protect the assignment statement

```
currentPos = list.next;
```

with the test for the empty list.

The `getNextItem` method has become simpler. For the linear list, this method had to explicitly test for the end of list condition, and handle it as a special case. For the circular list, this is no longer necessary. When an iteration reaches the end of the list the circular nature of the list ensures that it is redirected to the front of the list, as we wish.

Note that since the methods `reset` and `getNextItem` are both defined in the `LinkedList` class we must redefine them in the `CircularSortedLinkedList` class.

The new definitions override the `LinkedList` definitions for objects of the class `CircularSortedLinkedList`.

The isThere Method

The changes needed for `isThere` are also relatively easy. Recall that we wish to start our search at the beginning of the list, and continue searching sequentially until we find an element larger than the targeted item or we reach the end of the list. We must make sure that under the new arrangement we still begin our search at the beginning of the list. An easy way to do this is to change the list information we are comparing our targeted item to, from `location.info` to `location.next.info`. Note that we are guaranteed that `location.next.info` exists since when we access it we have already determined that the list is not empty.

Additionally, we must revise our method of determining when we have reached the end of the list. For a nonempty list, the `isThere` implementation in the `LinkedList` class terminates its search when `location` becomes `null`, indicating that the search has exhausted the entire list:

```
moreToSearch = (location != null);
```

With the circular list this test no longer works, since the list loops back upon itself rather than terminating with a `next` reference equal to `null`. The goal is to stop searching when the end of the list is reached. With the circular list, we know we reach the end of the list when `location` references the same node as the `list` variable (remember, `list` always indicates the last node on the list for the circular approach). So the terminating condition is set appropriately:

```
moreToSearch = (location != list);
```

The code, with the changes from the previous approach emphasized, is:

```
public boolean isThere (Listable item)
// Determines if element matching item is on this list
{
  int holdCompare;
  boolean moreToSearch;
  ListNode location = list;
  boolean found = false;

  moreToSearch = (location != null);

  while (moreToSearch && !found)
  {
    holdCompare = item.compareTo(location.next.info);
    if (holdCompare == 0)  // If they match
```

```
        found = true;
      else
      if (holdCompare < 0)    // If list element is larger than item
        moreToSearch = false;
      else
      {
        location = location.next;
        moreToSearch = (location != list);
      }
    }

  return found;
 }
```

Study this code and convince yourself that it works for an empty list and for lists of sizes 1, 2, and 5, for both the case when the item is on the list and the case in which the item is not on the list.

Deleting from a Circular List

We can use the same basic approach to deleting an element from a circular list as we used for a linear list. First, find the element that matches the targeted item and then delete it. To delete it we unlink it from the chain of elements by setting the `next` reference of the element previous to the identified element to reference the element after the identified element. Thus, when we delete an element we need a reference to the element that precedes it. Recall the "trick" we used for the linear list `delete` method, where we always looked at the element in the position after our current location, using the Boolean expression:

```
item.compareTo(location.next.info) != 0
```

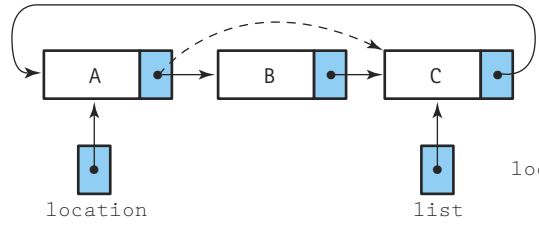
That way, when we found the element to delete, `location` held a reference to the previous node and we could "jump over" the node to be deleted with the statement

```
location.next = location.next.next;
```

In fact, using our trick with the circular list works very nicely. Since the original value of `location` is the node at the end of the list, the first information that we check is actually associated with the first node on the list. As with the linear approach, we are guaranteed to find our targeted element. When we do, `location` is referencing its predecessor on the list. To remove the targeted element from the list, we simply reset `location.next` as described above; we set it to jump over the node we are deleting. That works for the general case, at least (see Figure 6.3a).

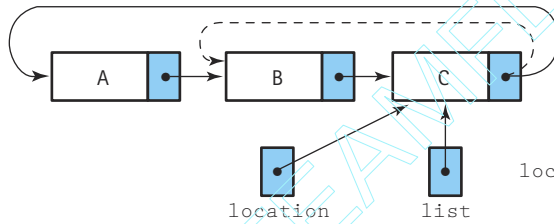
What kind of special cases do we have to consider? In the linear list version, we had to check for deleting the first (or first-and-only) element. However, the primary reason that there was a special case was that the overall list reference pointed to the first list

(a) The general case (delete B)



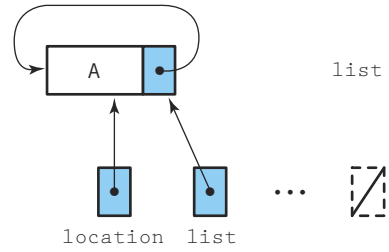

```
location.next = location.next.next;
```

(b) Special case (?): deleting the smallest item (delete A)



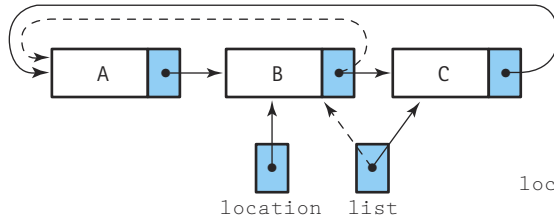

```
location.next = location.next.next;
```


(c) Special case: deleting the only item (delete A)




```
list = null;
```

(d) Special case: deleting the largest item (delete C)




```
list = location;
(the general case PLUS:)
location.next = location.next.next;
```

Figure 6.3 *Deleting from a circular linked list*

element and had to be updated if that element was deleted. In the circular version the overall list reference points to the last list element, so it is very possible that deleting the first element is not a special case. Figure 6.3(b) shows that guess to be correct. However, deleting the only node in a circular list is a special case, as we see in Figure 6.3(c). The reference to the list must be set to `null` to indicate that the list is now empty. We can detect this situation by checking, at the start of the method, whether `location` is equal to `location.next`. If it is, since we know from the method preconditions that the item we are deleting is on the list, in the case of the single element list we can simply delete it immediately. It must be the element that we wish to delete. We delete it by setting the `list` reference to `null`.

We might also guess that deleting the largest list element (the last node) from a circular list is a special case. After all, our reference to the list points to the last element, so if we delete it we must change our reference. As Figure 6.3(d) illustrates, when we delete the last node, we first update the overall list reference to point to the preceding element. We can detect this situation by checking whether `location.next` equals `list` after the search phase.

Here is the code for the `delete` method. Notice, of course, that in every case it decrements the value of `numItems`. You should trace through the code and convince yourself that it handles the general case, and all the special cases, properly.

```
public void delete (Listable item)
// Deletes the element of this list whose key matches item's key
{
  ListNode location = list;

  if (location == location.next)   // Single element list
    list = null;
  else
    {
    while (item.compareTo(location.next.info) != 0)
      location = location.next;
    if (location.next == list)         // Deleting last element
      list = location;
    // Delete node at location.next
    location.next = location.next.next;
    }
  numItems--;
}
```

The insert Method

The algorithm to insert an element into a circular linked list is also similar to its linear list counterpart. Essentially, we find the insertion location by performing a search, and insert the new item by rearranging some references. To do the insertion we need to have

access to both the node preceding the insertion point and the node following the insertion point. And, of course, we need to handle special cases carefully. A high-level description of the algorithm is:

insert (item)

Create a node for the new list element
Find the place where the new element belongs
Put the new element into the list
Increment the number of items

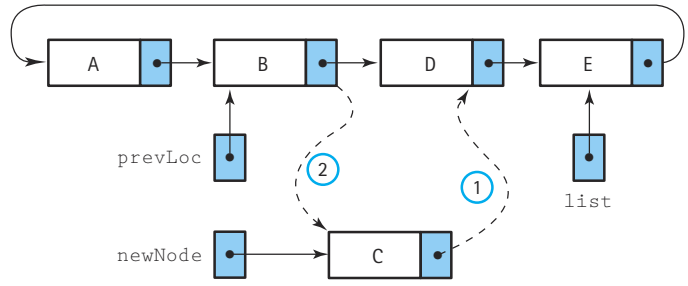
The task of creating a new node is the same as for the linear list. We allocate space for the node using the `new` operator and then store a copy of `item` into `newNode.info`. The next task is one that we are used to by now. We search through the list maintaining two references, `location` and `prevLoc`, until we find an element larger than `item` or reach the end of the list. The new node is linked into the list immediately after `prevLoc`. To put the new element into the list we store `location` into `newNode.next` and `newNode` into `prevLoc.next`.

The general case is illustrated in Figure 6.4(a). What are the special cases? First, we have the case of inserting the first element into an empty list. In this case, we want to make `list` point to the new node and to make the new node point to itself (Figure 6.4b). We handle this special case first, before doing any other processing. In the insertion algorithm for the linear linked list we also had a special case when the new element key was smaller than any other key in the list. Because the new node became the first node in the list, we had to change the reference to point to the new node. The reference to a circular list, however, doesn't point to the first node in the list—it points to the last node. Therefore, inserting the smallest list element is not a special case for a circular linked list (Figure 6.4c). However, inserting the largest list element at the end of the list is a special case. In addition to linking the node to its predecessor (previously the last list node) and its successor (the first list node), we must modify the list reference to point to `newNode`—the new last node in the circular list (Figure 6.4d).

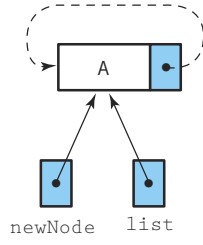
The statements to link the new node to the end of the list are the same as the general case, plus the assignment of the reference, `list`. Rather than checking for this special case before the search, we can treat it together with the general case: We search for the insertion place and link in the new node. Then, if we detect that we have added the new node to the end of the list, we reassign `list` to point to the new node. To detect this condition, we compare `item` to `list.info`.

The remaining task, incrementing `numItems`, finishes the task. The resulting implementation of `insert` is shown next.

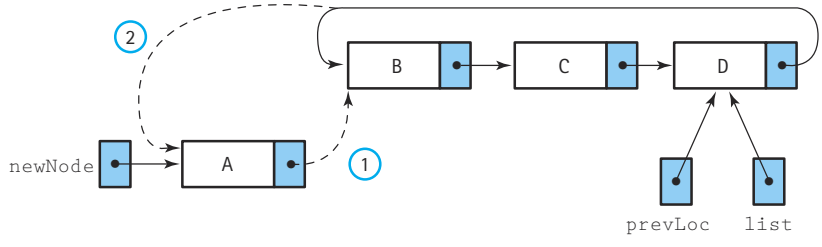
(a) The general case (insert C)



(b) Special case: the empty list (insert A)



(c) Special case: (?): inserting to front of list (insert A)



(d) Special case: inserting to end of list (Insert E)

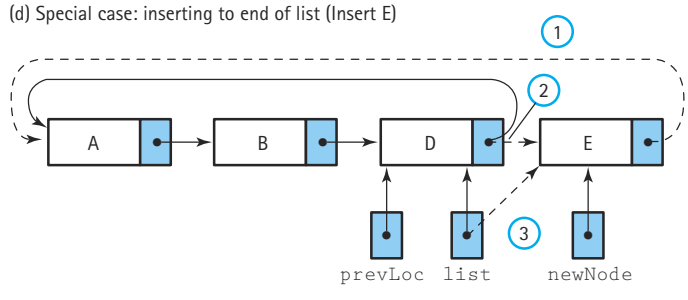


Figure 6.4 *Inserting into a circular linked list*

```
public void insert (Listable item)
// Adds a copy of item to list
{
  ListNode newNode = new ListNode();     // Reference to node being inserted
  newNode.info = (Listable)item.copy();  // Set info attribute of new node

  if (list == null)      // Insert into an empty list
  {
    list = newNode;
    newNode.next = newNode;
  }
  else                   // Insert into a nonempty list
  {
    ListNode prevLoc = new ListNode();     // Create trailing reference
    ListNode location = new ListNode();    // Create traveling reference
    boolean moreToSearch = true;

    location = list.next;
    prevLoc = list;

    // Find insertion point
    while (moreToSearch)
    {
      if (item.compareTo(location.info) < 0) // List element is larger than item
        moreToSearch = false;
      else
      {
        prevLoc = location;
        location = location.next;
        moreToSearch = (location != list.next);
      }
    }

    // Insert node into list
    newNode.next = location;
    prevLoc.next = newNode;
    if (item.compareTo(list.info) > 0)         // New item is last on this list
      list = newNode;
  }
  numItems++;
}
```

The entire `CircularSortedLinkedList` class can be found on the web site.

Circular Versus Linear

Studying circular linked lists provided good practice with using references and self-referential structures. Are circular lists good for anything else? You may have noticed that the only operation that is simpler for the circular approach, as compared to the linear approach, is `getNextItem`; that minimal advantage is counterbalanced by a more complicated `reset` operation. Why then might we want to use a circular, rather than linear, linked list?

Circular lists are good for applications that require access to both ends of the list. Our `CircularSortedLinkedList` class could be used as the basis for other classes that include operations that can take advantage of the new implementation. Perhaps we need a "maximum" operation that returns the largest list element; with the circular approach we have easy access to the largest element (through `list`). Or suppose we need an operation `inBetween` that returns a boolean value indicating whether a parameter item is "in between" the largest and smallest element of the list; as just mentioned, with the circular approach we have easy access to the largest element (through `list`) and we also have easy access to the smallest element (through `list.next`). Therefore, with the circular list, we could implement `inBetween` in O(1), whereas with our linear approach it would take O(N).

In addition, it is common for the data we want to add to a sorted list to already be in order. Sometimes people manually sort raw data before turning it over to a data entry clerk. Data produced by other programs are often in sorted order. Given a Sorted List ADT and sorted input data, we always insert at the end of the list, the most expensive place to insert in terms of machine time. It is ironic that the work done manually to order the data now results in maximum insertion times. A circular list with the list reference to the end of the list, as developed in this section, can be designed to avoid this execution overhead.

You may have realized that many of the benefits described here for circular lists could also be obtained by using the linear linked list defined in Chapter 5 augmented with a reference to the last element of the list. This is yet another list variation; as with the circular list, this variation requires changes to some of the linear list methods. We ask you to explore this variation in the exercises. The existence of many list variations is another reason for studying circular lists—it helps us understand how small changes in the structure underlying an ADT can require many subtle changes in the implementations of the ADT operations.

In the next section, we look at another important list structure, doubly linked lists. In this case, the advantages of the new approach are obvious—it lets us easily traverse a list in either direction.

6.2 Doubly Linked Lists

We have discussed using circular linked lists to enable us to reach any node in the list from any starting point. Although this structure has advantages over a simple linear linked list for some applications, it is still too limited for others. Suppose we want to be able to delete a particular node in a list, given only a reference to that node. This task involves changing the `next` reference of the node preceding the targeted node. However, given only a reference to a node, it is not easy to access its predecessor in the list.

Another task that is difficult to perform on a linear linked list (or even a circular linked list) is traversing the list in reverse. For instance, suppose we have a list of student records, sorted by grade point average (GPA) from lowest to highest. The Dean of Students might want a printout of the students' records, sorted from highest to lowest, to use in preparing the Dean's List. Or consider the Real Estate application presented in the case study at the end of Chapter 3. In that application the user can step through a list of house information, viewing the information house by house on the screen, by pressing a "next" button. Suppose the user requests an enhancement to the interface—the idea is to include a "previous" button so that the user can browse through the houses in either direction.

Doubly linked list A linked list in which each node is linked to both its successor and its predecessor

In cases like these, where we need to be able to access the node that precedes a given node, a doubly linked list is useful. In a doubly linked list, the nodes are linked in both directions. Each node of a doubly linked list contains three parts:

`info`: the data stored in the node
`next`: the reference to the following node
`back`: the reference to the preceding node

A linear doubly linked list is pictured in Figure 6.5. Note that the `back` reference of the first node, as well as the `next` reference of the last node, contains a `null`. The following definition might be used to declare the nodes in such a list:

```
protected class DLListNode
{
  // List nodes for the doubly linked list implementation
  protected Listable info;     // The info in a list node
  protected DLListNode next;   // A link to the next node on the list
  protected DLListNode back;   // A link to the next node on the list
}
```

The Insert and Delete Operations

Using the definition of `DLListNode`, let's discuss the corresponding `insert` and `delete` methods. The first step for both is to find the location to do the insertion or deletion. This step was complicated in the singly linked list situation by the need to hold onto a reference to the previous location during the search. That is why we created our inchworm search approach. That approach is no longer needed; instead, we can get the predecessor to any node through its `back` reference. This means we can revert to the simpler search approaches used with our array-based lists.

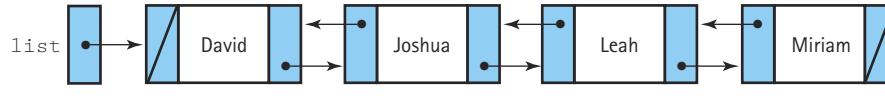


Figure 6.5 *A linear doubly linked list*

Although our search phase is simpler, the algorithms for the insertion and deletion operations on a doubly linked list are somewhat more complicated than for a singly linked list. The reason is clear: There are more references to keep track of in a doubly linked list.

For example, consider `insert`. To link a new node `newNode`, after a given node referenced by `prevLoc`, in a singly linked list, we need to change two references: `newNode.next` and `prevLoc.next` (see Figure 6.6a). The same operation on a doubly linked list requires four reference changes (see Figure 6.6b).

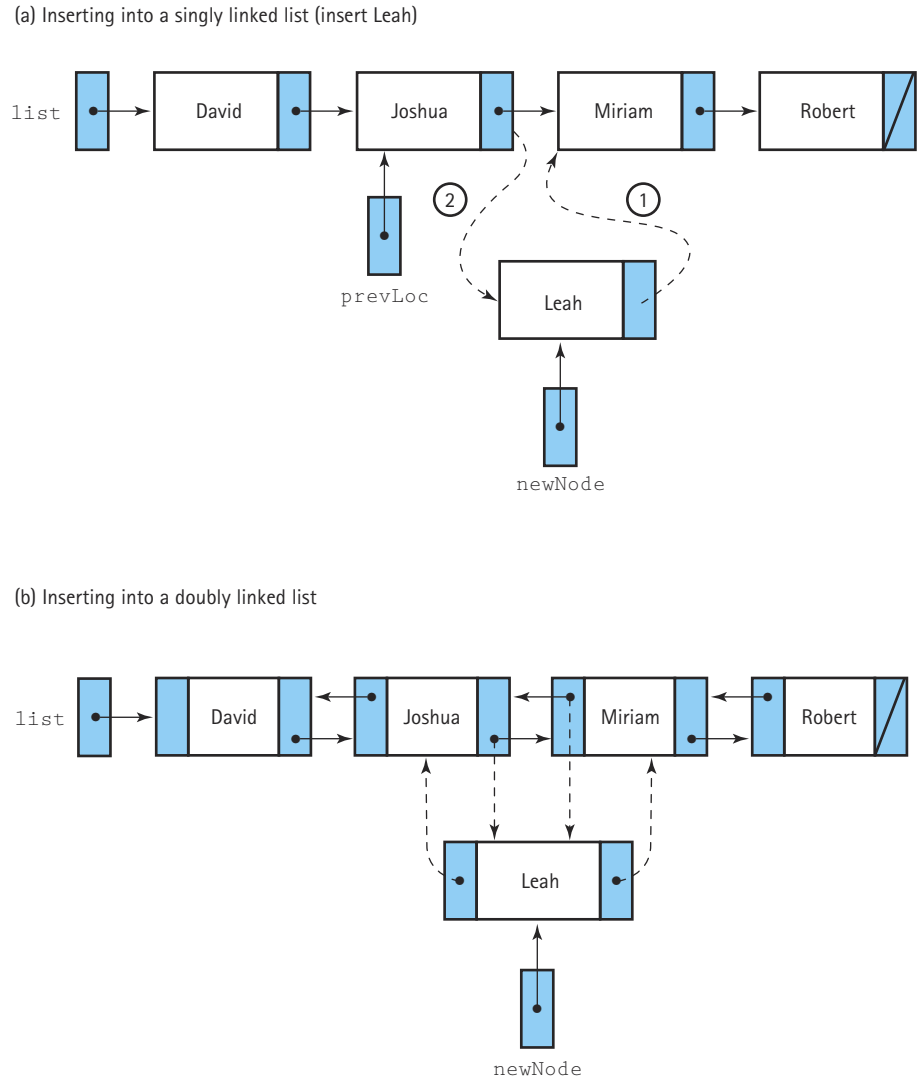


Figure 6.6 *Insertions into single and doubly linked lists*

To insert a new node we allocate space for the new node and search the list to find the insertion point. The result of our search is that `location` references the node that should follow the new node. Now we are ready to link the new node into the list. Because of the complexity of the operation, it is important to be careful about the order in which you change the references. For instance, when inserting the new node before `location`, if we change the reference in `location.back` first, we lose our reference to the node that is to precede the new node. The correct order for the reference changes is illustrated in Figure 6.7. The corresponding code would be

```
newNode.back = location.back;
newNode.next = location;
location.back.next = newNode;
location.back = newNode;
```

We do have to be careful about inserting into an empty list, as it is a special case.

Now let's consider the `delete` method. One of the useful features of a doubly linked list is that we don't need a reference to a node's predecessor in order to delete the node. Through the `back` reference, we can alter the `next` variable of the preceding node to make it jump over the unwanted node. Then we make the `back` reference of the succeeding node point to the preceding node. This operation is pictured in Figure 6.8.

We do, however, have to be careful about the end cases. If `location.back` is `null`, we are deleting the first node; if `location.next` is `null`, we are deleting the last node. If both `location.back` and `location.next` are `null`, we are deleting the only node. We leave the complete coding of the `insert` and `delete` methods for the doubly linked list as an exercise.

The List Framework

Before leaving the topic of doubly linked lists we should address the question of how they fit into our list framework. Although a doubly linked list is a form of linked list, it is based on a different underlying logical structure than our other linked lists. The rela-

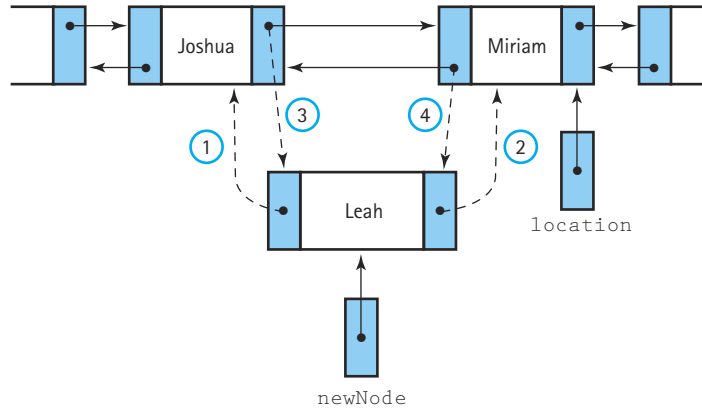


Figure 6.7 *Inserting into a doubly linked list*

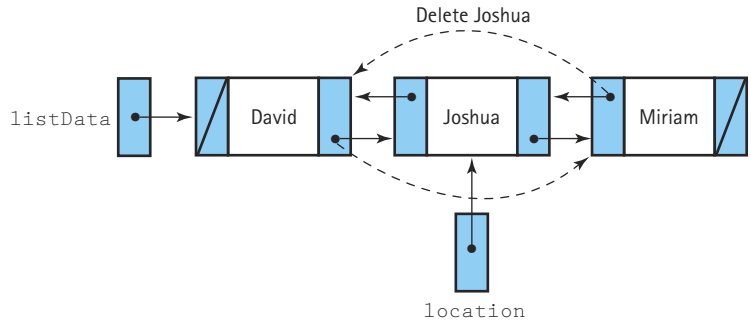


Figure 6.8 *Deleting from a doubly linked list*

tionship among its list elements is different from the case of the singly linked list. Therefore, it does not make sense to have it extend the abstract `LinkedList` class as we did for the other implementations. Instead, there are several other viable options.

We could create a new class, perhaps called `DoublyLinkedList`, and require it to implement our current `ListInterface` interface. In this case, we would probably want to add some additional methods to `DoublyLinkedList` that are not required by the interface. For example, we could add a `getPreviousItem` method; otherwise, what is the benefit of having the double links?

Alternately, we could create a new interface, perhaps called `TwoWayListInterface` that defines what we expect of lists that can be traversed in two directions. Then we could create a class based on doubly linked lists that implements the new interface. The new interface would probably require all of the operations that are part of our current `ListInterface` interface, plus a few more. Certainly, we would add the `getPreviousItem` operation. Of course, a doubly linked list is not the only possible way to implement such an interface. An array-based approach would also work well.

The fact that the proposed new interface would require all of the operations of our current `ListInterface` interface raises another possible approach. Java supports the inheritance of interfaces. (In fact, the language supports multiple inheritance of interfaces, so that a single interface can extend any number of other interfaces.) A good approach would be to define a new interface that extends `ListInterface` and adds a `getPreviousItem` method. Figure 6.20, in the chapter summary section, shows all the changes to our list framework made in this chapter; it assumes that we followed this last option with respect to doubly linked lists.

Inheritance of interfaces A Java interface can extend another Java interface, inheriting its requirements. If interface B extends interface A, then classes that implement interface B must also implement interface A. Usually, interface B adds additional abstract methods to those required by interface A.

Multiple inheritance of interfaces Unlike for classes, Java does support multiple inheritance of interfaces. If interface C extends both interface A and interface B, then classes that implement interface C must also implement both interface A and interface B. Sometimes multiple inheritance of interfaces is used simply to combine the requirements of two interfaces, without adding any additional abstract methods.

Here is the code for the new interface:

```
package ch06.genericLists;

import ch04.genericLists.*;

public interface TwoWayListInterface extends ListInterface

// Interface for a class that implements a list of elements as defined
//  by ListInterface, with the additional operation(s) defined below

{
  public Listable getPreviousItem ();
  // Effect:          Returns a copy of the element preceding the current
  //                  position on this list and moves back the value of the
  //                  current position
  //                  The last element precedes the first element
  // Preconditions:   Current position is defined
  //                  There exists a list element preceding current
  //                  position. No list transformers have been called since
  //                  most recent reset
  // Postcondition:   Return value = (a copy of element previous to current
  //                  position)
}
```


6.3 Linked Lists with Headers and Trailers

In writing the insert and delete algorithms for all implementations of linked lists, we see that special cases arise when we are dealing with the first node or the last node. One way to simplify these algorithms is to make sure that we never insert or delete at the ends of the list.

How can this be accomplished? Recall that the elements in the sorted linked list are arranged according to the value in some key—for example, numerically by identification number or alphabetically by name. If the range of possible values for the key can be determined, it is often a simple matter to set up dummy nodes with values outside of this range. A header node, containing a value smaller than any possible list element key, can be placed at the beginning of the list. A trailer node, containing a value larger than any legitimate element key, can be placed at the end of the list.

Header node A placeholder node at the beginning of a list; used to simplify list processing

Trailer node A placeholder node at the end of a list; used to simplify list processing

The header and the trailer are regular nodes of the same type as the real data nodes in the list. They have a different purpose, however; instead of storing list data, they act as placeholders.

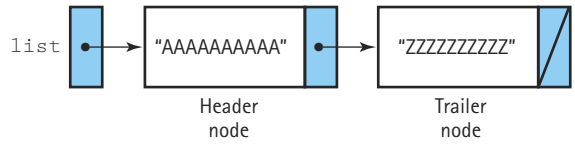


Figure 6.9 *An "empty" list with a header and a trailer*

If a list of students is sorted by last name, for example, we might assume that there are no students named "AAAAAAAAAA" or "ZZZZZZZZZZ". We could therefore initialize our linked list to contain header and trailer nodes with these values as the keys. See Figure 6.9. How can we implement a general list ADT if we must know the minimum and maximum key values? We can use a parameterized class constructor and let the user pass as arguments elements containing the dummy keys.

6.4 A Linked List as an Array of Nodes

We tend to think of linked structures as being dynamically allocated as needed, using self-referential nodes as illustrated in Figure 6.10(a), but this is not a requirement. A linked list could be implemented in an array; the elements might be stored in the array in any order, and "linked" by their indexes (see Figure 6.10b). In this section, we develop an array-based linked-list implementation.

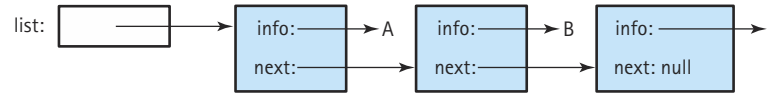
In our previous reference-based implementations of lists, we have used Java's built in memory management services when we needed a new node for insertion or when we were finished with a node and wanted to delete it. Obtaining a new node is easy in Java; we just use the familiar `new` operation. Releasing a node from use is also easy; we just remove our references to it and depend on the Java run time system's garbage collector to reclaim the space used by the node.

For the array-based linked representation developed in this section, we can no longer rely on Java's built-in space management support. Instead, we predetermine the maximum list size and instantiate an array of list nodes of that size. We then directly manage the nodes in the array, much like the Java system manages free space. We keep a separate list of the available nodes, and write routines to allocate and deallocate nodes, from and to this free list. Many programming languages do not provide the same space management support that Java does. Working with the space management required by the array-based linked approach is good practice for you, in case you ever have to use those languages.

Why Use an Array?

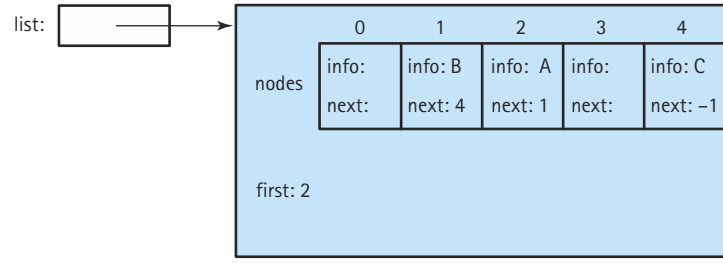
We have seen that dynamic allocation of list nodes has many advantages, so why would we even discuss using an array-of-nodes implementation instead? Remember that dynamic allocation is only one advantage of choosing a linked implementation; another

(a) a linked list in dynamic storage




```
class ListNode
{
 protected Listable info;
 protected ListNode next;
}

ListNode list = new ListNode();
```

(b) a linked list in static storage




```
class AListNode
{
 protected Listable info;
 protected int next;
}

class ArrayLinkedList
{
 AListNode[] nodes = new AListNode[5];
 int first;
}

ArrayLinkedList list = new ArrayLinkedList();
```

Figure 6.10 *Linked lists in dynamic and static storage*

advantage is the efficiency of the insert and delete algorithms. Most of the algorithms that we have discussed for operations on a linked structure can be used for either an array-based or a reference-based implementation. The main difference is the requirement that we manage our own free space in an array-based implementation. Sometimes, managing the free space ourselves gives us greater flexibility.

Another reason to use an array of nodes is that there are programming languages that do not support dynamic allocation or reference types. You can still use linked structures if you are programming in one of these languages, using the techniques presented in this section.

With some languages, using references presents a problem when we need to save the information in a data structure between runs of a program. Suppose we want to write all the nodes in a list to a file and then use this file as input the next time we run the program to recreate the list. If the links are reference values—containing memory addresses—they are meaningless on the next run of the program because the program may be placed somewhere else in memory the next time. We must save the user data

part of each node in the file and then rebuild the linked structure the next time we run the program. An array index, however, is still valid on the next run of the program. We can store the array information, including the next data index, and then read it back in the next time we run the program. (Note that Java's serialization facilities make the use of the array approach for this reason unnecessary. We discuss the serialization facilities in Section 9.4: Storing Objects/Structures in Files.)

Most importantly, there are times when dynamic allocation isn't possible or feasible, or when dynamic allocation of each node, one at a time, is too costly in terms of time—especially in real-time system software such as operating systems, air traffic controllers, and automotive systems. In such situations, an array-based linked approach provides the benefits of linked structures without the runtime costs.

How Is an Array Used?

Let's get back to our discussion of how a linked list can be implemented in an array. We can associate a `next` variable with each array node to indicate the array index of the succeeding node. The beginning of the list is accessed through a "reference" that contains the array index of the first element in the list. Figure 6.11 shows how a sorted list

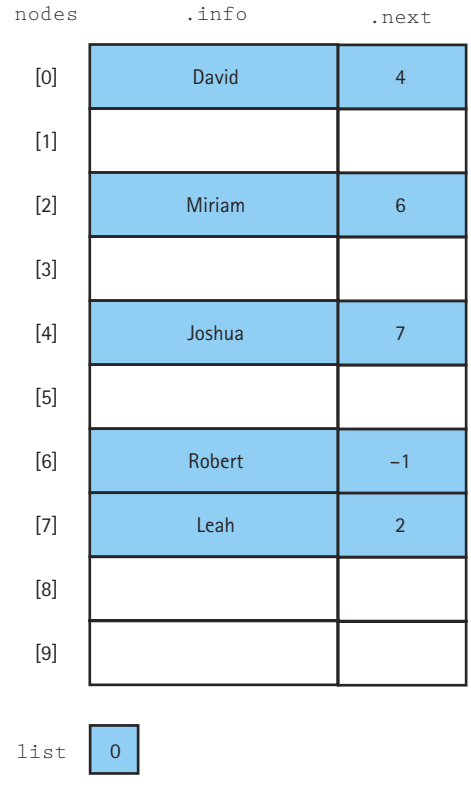


Figure 6.11 *A sorted list stored in an array of nodes*

containing the elements "David," "Joshua," "Leah," "Miriam," and "Robert" might be stored in an array of nodes called `nodes`. Do you see how the order of the elements in the list is explicitly indicated by the chain of `next` indexes?

What goes in the `next` index of the last list element? Its "null" value must be an invalid address for a real list element. Because the `nodes` array indexes begin at 0, the value −1 is not a valid index into the array; that is, there is no `nodes`[−1]. Therefore, −1 makes an ideal value to use as a "null" address. We could use the literal value −1 in our programs:

```
while (location != -1)
```

but it is better programming style to declare a named constant. We use the identifier `NUL` and define it to be −1:

```
private static final int NUL = -1;
```

When an array-of-nodes implementation is used to represent a linked list, the programmer must write routines to manage the free space available for new list elements. Where is this free space? Look again at Figure 6.11. All of the array elements that do not contain values in the list constitute free space. Instead of the built-in allocator `new`, which allocates memory dynamically, we must write our own method to allocate nodes from the free space. We call this method `getNode`. We use `getNode` when we insert new items onto the list.

When elements are deleted from the list, we need to free the node space, that is, to return the deleted node to the free space, so it can be used again later. We can't depend on a garbage collector; the node we delete remains in the allocated array so it is not reclaimed by the run-time engine. We write our own method, `freeNode`, to put a node back into the pool of free space.

We need a way to track the collection of nodes that are not being used to hold list elements. We can link this collection of unused array elements together into a second list, a linked list of free nodes. Figure 6.12 shows the array nodes with both the list of elements and the list of free space linked through their `next` values. The `list` variable is a reference to a list that begins at index 0 (containing the value David). Following the links in `next`, we see that the list continues with the array slots at index 4 (Joshua), 7 (Leah), 2 (Miriam), and 6 (Robert), in that order. The free list begins at `free`, at index 1. Following the `next` links, we see that the free list also includes the array slots at index 5, 3, 8, and 9. You see two `NUL` values in the next column because there are two linked lists contained in the nodes array; so there are two end-of-list values in the array.

There are two approaches to using an array-of-nodes implementation for linked structures. The first is to simulate dynamic memory with a single array. One array is used to store many different linked lists, just as the computer's free space can be dynamically allocated for different lists. In this approach, the references to the lists are not part of the storage structure, but the reference to the list of free nodes is part of the structure. Figure 6.13 shows an array that contains two different lists. The list indicated

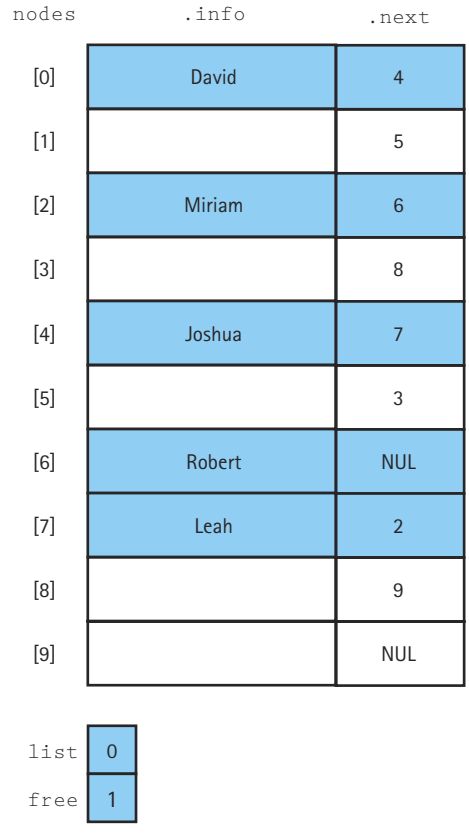


Figure 6.12 *An array with linked list of values and free space*

by `list1` contains the values "John," "Nell," "Susan," and "Suzanne" and the list indicated by `list2` contains the values "Mark," "Naomi," and "Robert." The remaining three array slots in Figure 6.13 are linked together in the free list.

Another approach is to have one array of nodes for each list. In this approach, the reference to the list is part of the storage structure itself (see Figure 6.14). This works since there is only one list. The list constructor has a parameter that specifies the maximum number of items to be on the list. This parameter is used to dynamically allocate an array of the appropriate size.

In this section, we implement this second approach. We call our new class `ArrayLinkedList`. In implementing our class methods, we need to keep in mind that there are two distinct processes going on within the array of nodes: bookkeeping relating to the space (such as initializing the array of nodes, getting a new node, and freeing

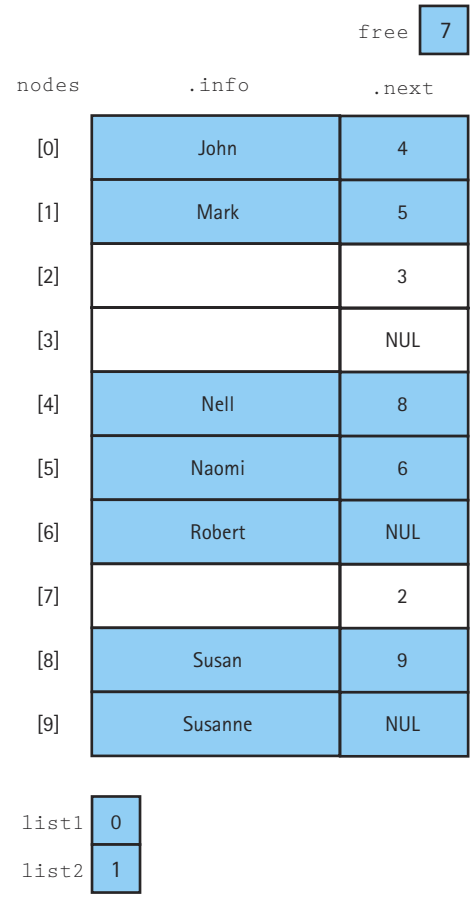


Figure 6.13 *An array with three lists (including the free list)*

a node) and the operations on the list that contains the user's data. The bookkeeping operations are transparent to the user. Our list interface does not change. In fact, our new class implements the `ListInterface` interface, just as all of our other list implementations have done. The private data, however, change. We need to include the array of nodes. Let's call this array `nodes` and have it hold elements of the class `AListNode`. Objects of the `AListNode` class, then, contain two attributes: `info` of type `Listable` that holds a reference to a copy of the user's data, and `next`, of the primitive type `int`, that holds the index of the next element on the list. In addition to the array of nodes, we need an integer "reference" to the first node of the list and another to the first free node. We call these `list` and `free`. And, of course, we still need our `numItems` and `currentPos` variables. Next is the beginning of our class file:

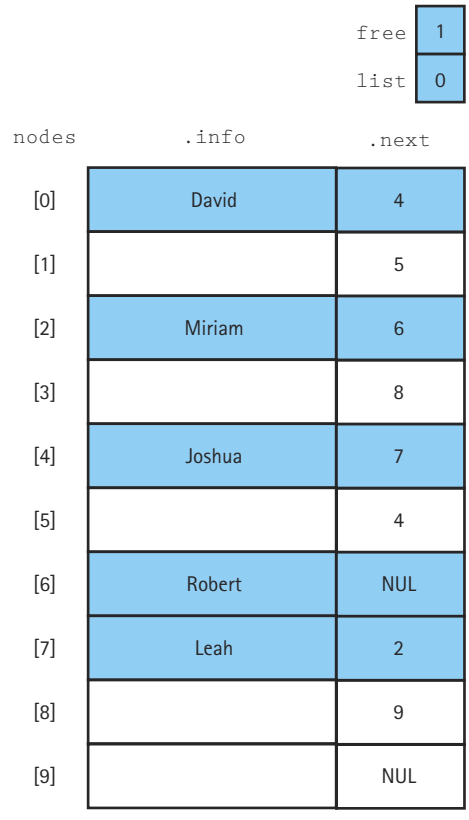


Figure 6.14 *List and link structure are together*

```
//--------------------------------------------------------------------------
// ArrayLinkedList.java          by Dale/Joyce/Weems                Chapter 6
//
// Implements an array-based sorted linked list of Listable elements
//--------------------------------------------------------------------------

package ch06.genericLists;

import ch04.genericLists.*;

public class ArrayLinkedList implements ListInterface
{
  private static final int NUL = -1;   // End of list symbol
```

```
private class AListNode
{
  private Listable info;       // The info in a list node
  private int next;            // A link to the next node on the list
}

private AListNode[] nodes;     // Array of AListNode holds the linked list

private int list;              // Reference to the first node on the list
private int free;              // Reference to the first node on the free list

private int numItems;          // Number of elements in the list
private int currentPos;        // Current position for iteration
```

The class constructors for class `ArrayLinkedList` must allocate the storage for the array of nodes and initialize all of the private instance variables. They must also set up the initial free list of nodes. At the time of instantiation, all of the nodes are on the free list. So, the variable `free` is set to 0 to "reference" the first array node, and the `next` value of that node is set to 1, and so on until all of the nodes are chained together. This initialization can be handled by a *for* loop, followed by a single assignment statement to set the last `next` value to `NUL`. To be consistent with our past array-based implementation we should provide two constructors, one that accepts a size parameter and one that uses a default maximum size. Here is the code for the constructor that takes a parameter:

```
public ArrayLinkedList(int maxItems)
// Instantiates and returns a reference to an empty list object with
//  room for maxItems elements
{
  nodes = new AListNode[maxItems];
  for (int index = 0; index < maxItems; index++)
    nodes[index] = new AListNode();

// Link together the free nodes.
  for (int index = 1; index < maxItems; index++)
    nodes[index - 1].next = index;
  nodes[maxItems - 1].next = NUL;

  list = NUL;
  free = 0;
  numItems = 0;
  currentPos = NUL;
}
```

The methods that do the bookkeeping, `getNode` and `freeNode`, are auxiliary ("helper") methods, and therefore are private methods. The `getNode` method must return the index of the next free node. The easiest node to use is the one at the beginning of the free list, so `getNode` returns the value of `free`. Therefore, `getNode` must also update the value of `free` to indicate the next node on the free list. Other than the fact that we must be careful of our order of operations, and use a temporary variable to hold the index we need to return, this method is straightforward:

```
private int getNode()
// Returns the index of the next available node from the free list
//  and updates the free list index
{
  int hold;
  hold = free;
  free = nodes[free].next;
  return hold;
}
```

The `freeNode` method must take the node index received as an argument and insert the corresponding node into the list of free nodes. As the first element in the list is the one that is directly accessible, we have `freeNode` insert the node being returned at the beginning of the free list, in the variable `free` (Yes, we are keeping the free list as a stack—not because we need the LIFO property but because the code is the simplest for what we need.)

```
private void freeNode(int index)
// Frees the node at array position index by linking it into the
//  free list
{
  nodes[index].next = free;
  free = index;
}
```

The public methods are very similar to their reference-based linked list counterparts. From the point of view of the algorithm used, they are identical. At the beginning of Chapter 3, we established a list design terminology that is independent of implementation approach. We were able to implement algorithms expressed in this notation for both the array-based lists and the reference-based lists. Now we have a third implementation approach, the array-based linked approach. The following table shows equivalent expressions in each of our views of a list. It also shows the expressions for creating and deleting nodes, where appropriate.

Design Terminology	Array-Based	Reference-Based	Array Index Links
location.node()	`list[location]`	`location`	`nodes[location]`
location.info()	`list[location]`	`location.info`	`nodes[location].info`
location.next()	`list[location+1]`	`location.next`	`nodes[location].next`
allocate a node N	done by constructor	`ListNode N = new ListNode`	`N = getNode()`
free a node N	not applicable	remove links, garbage collector	`freeNode(N)`

We look here at some of the methods, and leave the rest as an exercise. First an easy one: `isFull`. We have two ways of determining whether or not the list is full. As for our array-based list implementation, we can compare the number of items on the list to the size of the underlying array. If they are equal then the list is full. But there is an even easier way. Can you think of it? If the entire array is being used to hold our list, then the list of free space must be empty. So, we can just check to see if `free` is equal to `NUL`.

```
public boolean isFull()
// Determines whether this list is full
{
  return (free == NUL);
}
```

The remaining methods can be implemented following the same scheme devised for the reference-based approach. You must be careful, however, to correctly transform the implementation. And don't forget that you have to handle the memory management yourself. Let's look at the `delete` method. Consider the following statement from the `delete` method of the `LinkedList` class:

```
while (item.compareTo(location.next.info) != 0)
  location = location.next;
```

Think for a minute about how you would represent this statement using the approach of this section. It is not as simple as it might first appear. You need to compare the `info` value of the next element to `item`. Using the table above, you see that you access the `info` attribute of a location with `nodes[location].info`. However, you don't want the `info` value of the location, you want the `info` value of the next location. So, you must replace `location` with the expression that stands for the next location. In this case, that is `nodes[location].next`. Putting this altogether (see Figure 6.15), the corresponding code is:

```
while (item.compareTo(nodes[nodes[location].next].info) != 0)
  location = nodes[location].next;
```

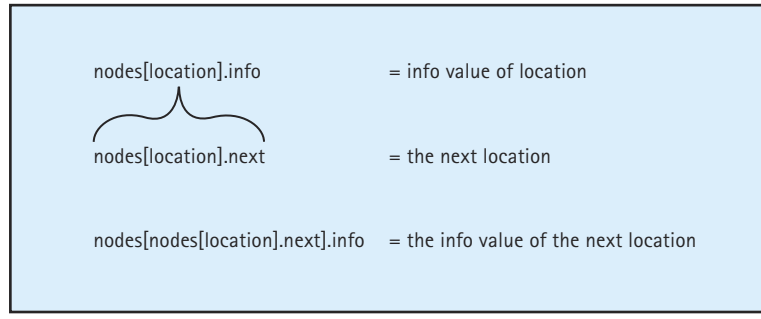


Figure 6.15 *Referencing the* `info` *value of the* `next` *node*

Below is the entire `delete` method. Notice how we carefully store the index of the node being deleted, so that we can "free" it before leaving the method. Compare the code for this `delete` method to the code presented in Chapter 5 for the `delete` method of the `LinkedList` class.

```
public void delete (Listable item)
// Deletes the element of this list whose key matches item's key
{
  int hold;                          // To remember deleted node index
  int location = list;

  // Locate node to be deleted
  if (item.compareTo(nodes[location].info) == 0)
  {
    hold = list;
    list = nodes[list].next;
// Delete first node
  }
  else
    {
    while (item.compareTo(nodes[nodes[location].next].info) != 0)
      location = nodes[location].next;
    // Delete node at nodes[location].next
    hold = nodes[location].next;
    nodes[location].next = nodes[nodes[location].next].next;
    }
  freeNode(hold);
  numItems--;
}
```

6.5 A Specialized List ADT

We have defined Unsorted and Sorted List ADTs and several implementations of each. Our lists can be used for many applications. However, there are always some applications that need special purpose lists; perhaps they require specific list operations that are not defined by our List ADTs or perhaps the specific qualities of our lists (unique elements, store by copy) do not fit with the requirements of the application. In such cases, we may be able to extend one of our list classes to create a new list that meets the needs of the application. Alternately, it may be easier just to create a new specialized list class, specifically for the application in question.

In the case study in the next section, we need lists with a unique set of properties and operations. The lists must hold elements of the primitive type `byte`; duplicate elements are allowed. The lists need not support `isFull`, `isThere`, `retrieve`, or `delete`. In fact, the only list operations that we have been using that are still required by this new list construct are the `lengthIs` operation and the iterator operations. For the case study, we are going to need to process elements from left-to-right and from right-to-left, so we need to support two iterators. In addition, we are going to need to insert items at the front and at the back of our lists. The reasons for these requirements are made clear in the case study; for now we just accept the requirements as stated and consider how to implement this new list.

The Specification

Given this unique set of requirements, we decide to start from scratch for this new List ADT. Of course, we can reuse our knowledge of lists and maybe even reuse (through cut and paste) some of the code from the previous list implementations, but we are not going to implement the `ListInterface` interface, and we are not going to extend any of our current classes. Since the new list construct creates a specialized list for a specific application, we call the list class `SpecializedList`, and we specify its behavior in an interface called `SpecializedListInterface`. The new list does not provide a list of generic elements; instead, it provides a list of `byte` elements. Recall that a byte is one of Java's primitive integer types. A byte can hold an integer between −128 and +127. We place the classes related to our new list in a package called `ch06.byteLists`.

Given the requirement that we must be able to iterate through the list in both directions, instead of our standard "current position" property, lists of the class `SpecializedList` have both a "current forward position" and a "current backward position" and provide iterator operations for traversing the list in either direction. Note that this does not mean that an iteration can change directions—it means that there can be two separate iterations going on at the same time, one forward and one backward.

Here is the formal specification of the new list ADT:

```
//--------------------------------------------------------------------------
// SpecializedListInterface.java     by Dale/Joyce/Weems              Chapter 6
//
// Interface for a class that implements a list of bytes
// There can be duplicate elements on the list
```

```
// The list has two special properties called the current forward position
//  and the current backward position -- the positions of the next element
//  to be accessed by getNextItem and by getPriorItem during an iteration
//  through the list. Only resetForward and getNextItem affect the current
//  forward position. Only resetBackward and getPriorItem affect the current
//  backward position. Note that forward and backward iterations may be in
//  progress at the same time
//-------------------------------------------------------------------------

package ch06.byteLists;

public interface SpecializedListInterface
{
  public void resetForward();
  // Effect:         Initializes current forward position for this list
  // Postcondition:  Current forward position is first element on this list

  public byte getNextItem ();
  // Effect:         Returns the value of the byte at the current forward
  //                 position on this list and advances the value of the
  //                 current forward position
  // Preconditions:  Current forward position is defined
  //                 There exists a list element at current forward position
  //                 No list transformers have been called since most recent
  //                 call
  // Postconditions: Return value = (value of byte at current forward position)
  //                 If current forward position is the last element then
  //                 current forward position is set to the beginning of this
  //                 list, otherwise it is updated to the next position

  public void resetBackward();
  // Effect:         Initializes current backward position for this list
  // Postcondition:  Current backward position is first element on this list

  public byte getPriorItem ();
  // Effect:         Returns the value of the byte at the current backward
  //                 position on this list and advances the value of the
  //                 current backward position (towards front of list)
  // Preconditions:  Current backward position is defined
  //                 There exists a list element at current backward position
  //                 No list transformers have been called since most recent
  //                 call
  // Postconditions: Return value = (value of byte at current backward
  //                 position)
  //                 If current backward position is the first element then
  //                 current backward position is set to the end of this list;
  //                 otherwise, it is updated to the prior position
```

```
    public int lengthIs();
    // Effect:         Determines the number of elements on this list
    // Postcondition:  Return value = number of elements on this list

    public void insertFront (byte item);
    // Effect:         Adds the value of item to the front of this list
    // PostCondition:  Value of item is at the front of this list

    public void insertEnd (byte item);
    // Effect:         Adds the value of item to the end of this list
    // PostCondition:  Value of item is at the end of this list
}
```

The Implementation

The unique requirement for the `SpecializedList` is that we are able to traverse it either frontward or backward. Because a doubly linked list is linked in both directions, traversing the list either way is equally simple. Therefore, we use a reference-based doubly linked structure for our implementation.

To begin our backward traversals, and to support the new `insertEnd` operation, it is clear that we need easy access to the end of a list. We have already seen how keeping the list reference pointing to the last element in a circular structure gives direct access to both the front element and the last element. So, we could use a doubly linked circular structure. However, another approach is also possible. We can maintain two list references, one for the front of the list and one for the back of the list. We use this approach in Figure 6.16.

Here is the beginning of the `SpecializedList` class. We use a doubly linked reference based approach, with instance variables to track the first list element, the last list element, the number of items on the list, and the positions for both the forward traversal and the backward traversal. Note that the `info` attribute of the `SListNode` class holds a value of the primitive `byte` type, as was discussed above.

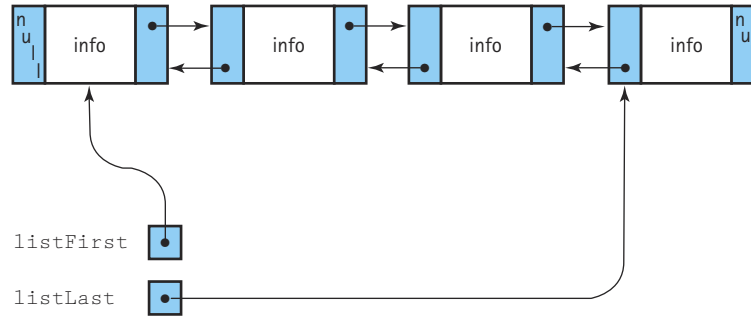


Figure 6.16 *A doubly linked list with two references*

```
//--------------------------------------------------------------------------
// SpecializedList.java          by Dale/Joyce/Weems                Chapter 6
//
// Implements the specialized list ADT using a doubly linked list of nodes
//--------------------------------------------------------------------------

package ch06.byteLists;

public class SpecializedList implements SpecializedListInterface
{
  protected class SListNode
  // List nodes for the specialized list implementation
  {
    protected byte info;        // The info in a list node
    protected SListNode next;   // A link to the next node on the list
    protected SListNode back;   // A link to the next node on the list
  }

  protected SListNode listFirst;      // Reference to the first node on list
  protected SListNode listLast;       // Reference to the last node on the list
  protected int numItems;             // Number of elements in the list
  protected SListNode currentFPos;    // Current forward position for iteration
  protected SListNode currentBPos;    // Current backward position for
                                      //  iteration

  public SpecializedList()
  // Creates an empty list object
  {
    numItems = 0;
    listFirst = null;
    listLast = null;
    currentFPos = null;
    currentBPos = null;
  }
```

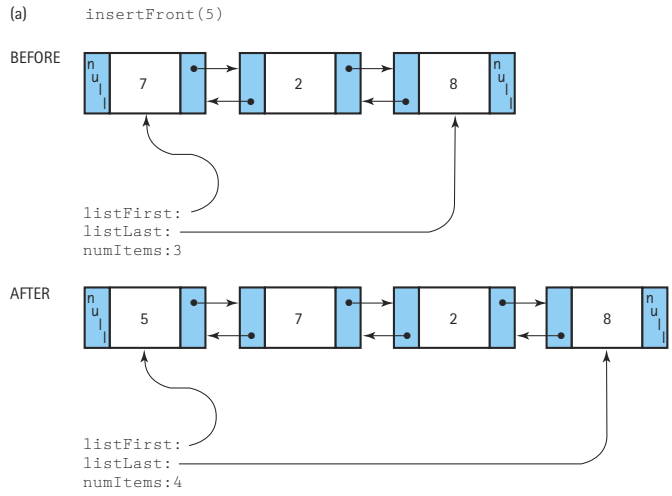
The `lengthIs` method is essentially unchanged from previous implementations—it simply returns the value of the `numItems` instance variable.

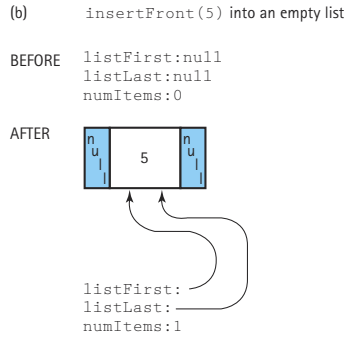
```
  public int lengthIs()
  // Determines the number of elements on this list
  {
    return numItems;
  }
```

The iterator methods are straightforward. Resetting an iteration simply requires setting the appropriate instance variable to either the front of the list or the back of the list. The methods that return the next element for an iteration work as they have in the past. A copy is made of the element to return, the current position is changed appropriately, and the information is returned. Changing the current position is guarded by an *if* statement, that handles the case of wrapping around the list.

```
public void resetForward()
// Initializes current forward position for an iteration through this list
{
  currentFPos = listFirst;
}

public byte getNextItem ()
// Returns the value of the next element in list in forward iteration
{
  byte nextItemInfo = currentFPos.info;
  if (currentFPos == listLast)
    currentFPos = listFirst;
  else
    currentFPos = currentFPos.next;

  return nextItemInfo;
}

public void resetBackward()
// Initializes current backward position for an iteration through this list
{
  currentBPos = listLast;
}

public byte getPriorItem ()
// Returns the value of the next element in list in backward iteration
{
  byte nextItemInfo = currentBPos.info;
  if (currentBPos == listFirst)
    currentBPos = listLast;
  else
    currentBPos = currentFPos.back;

  return nextItemInfo;
}
```

The insertion methods are simpler then the insertion method for the doubly linked list we used before. This is because we do not have to handle the general case insertion. The `insertFront` method always inserts at the front of the list and the `insertEnd` method always inserts at the end of the list. Let's look at `insertFront` (see Figure 6.17a). The method begins by creating the new node and initializing its attributes. Since





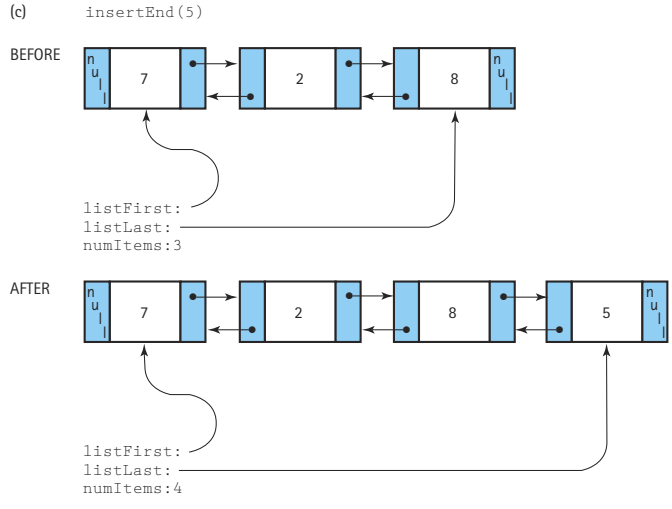


Figure 6.17 *Inserting at the front and at the end*

the new node is the new front of the list, we know that its `next` link should reference the current front of the list, and its `back` link should be `null`. An `if` statement guards the case when the insertion is into an empty list (see Figure 6.17b). In that case, both the `listFirst` and `listLast` instance variables must reference the new node, since it is both the first and last elements of the list. Otherwise, the `back` link of the previous first element is set to reference the new element, along with the `listFirst` instance variable. And, of course, we must increment the value of `numItems`.

```
public void insertFront (byte item)
// Adds the value of item to the front of this list
{
  SListNode newNode = new SListNode();   // Reference to node being inserted
  newNode.info = item;
  newNode.next = listFirst;
  newNode.back = null;
  if (listFirst == null)             // Inserting into an empty list
  {
    listFirst = newNode;
    listLast = newNode;
  }
  else                               // Inserting into a nonempty list
  {
    listFirst.back = newNode;
    listFirst = newNode;
  }
  numItems++;
}
```

The code for the `insertEnd` method is similar (see Figure 6.17c):

```
public void insertEnd (byte item)
// Adds the value of item to the end of this list
{
  SListNode newNode = new SListNode();   // Reference to node being inserted
  newNode.info = item;
  newNode.next = null;
  newNode.back = listLast;
  if (listFirst == null)             // Inserting into an empty list
  {
    listFirst = newNode;
    listLast = newNode;
  }
  else                               // Inserting into a nonempty list
  {
    listLast.next = newNode;
    listLast = newNode;
  }
  numItems++;
}
```

Case Study

Large Integers

The range of integer values that can be supported in Java varies from one primitive integer type to another. Appendix C contains a table showing, for each primitive integer type, the kind of value stored by the type, the default value, the number of bits used to implement the type, and the possible range of values. Note that the largest integer type, type `long`, can be used to represent values between −9,223,372,036,854,775,808 and 9,223,372,036,854,775,807. Wow! That range of numbers would seem to suffice for most applications that we might want to write. However, even given that large range of integers, some programmer is bound to want to represent integers with larger values. Let's design and implement a class `LargeInt` that allows the client programmer to manipulate integers in which the number of digits is only limited by the size of the available memory.

Because we are providing an alternate implementation for a mathematical object, an integer number, most of the operations are already specified: addition, subtraction, multiplication, division, assignment, and the relational operators. For this case study, we limit our attention to addition (`add`) and subtraction (`subtract`). We ask you to enhance this ADT with some of the other operations in the exercises.

In addition to the standard mathematical operations, we need an operation that constructs a number digit by digit. This operation cannot be a constructor with an integer parameter, because the desired integer argument might be too large to represent in Java—after all, that is the idea of this ADT. So we must have a special member method (`addDigit`) that can be called within a loop that inserts the digits one at a time, most significant digit to least significant digit, as we would normally read a number. We assume the sign of a large integer is positive, but we do need a way to set it to negative if desired (`setNegative`). Additionally, we must have an observer operation that returns a string representation of the large integer. We follow the Java convention and call this operation `toString`.

The Underlying Representation Before we can begin to look at the algorithms for these operations, we need to decide on our underlying representation for a large integer. Because we said earlier that we were designing the class `SpecializedList` to use in this case study, you know that we are going to use it to represent our large integers. Nevertheless, let's assume we don't already know the answer to our question, and look at the reasons we developed the specific requirements that we used for `SpecializedList`.

The fact that we intend to place no limits on the size of a large integer leads us to a dynamic memory-based representation. And considering that an integer is a list of digits, it is natural to investigate the possibility of representing a large integer as a dynamically allocated linked list of digits. Figure 6.18 shows several examples of numbers in a singly linked list and an addition. Figure 6.18(a) and (c) show one digit per node; Figure 6.18(b) shows several digits per node. We develop our large integer ADT using a single digit per node. You are asked in the exercises to explore the necessary changes to include more than one digit in each node.

At this point, we have decided to represent our large integers as linked lists of digits. Since a single digit can be represented by Java's smallest integer type, the `byte`, we decide to use linked lists of `byte` values. Throughout Chapters 5 and 6 we developed several implementations of linked lists. Perhaps we can use one of them to support our large integers.

What kind of linked list should we use? Can we use one of our predefined generic list classes such as `UnsortedArrayList` or `CircularSortedLinkedList`? No, we cannot for

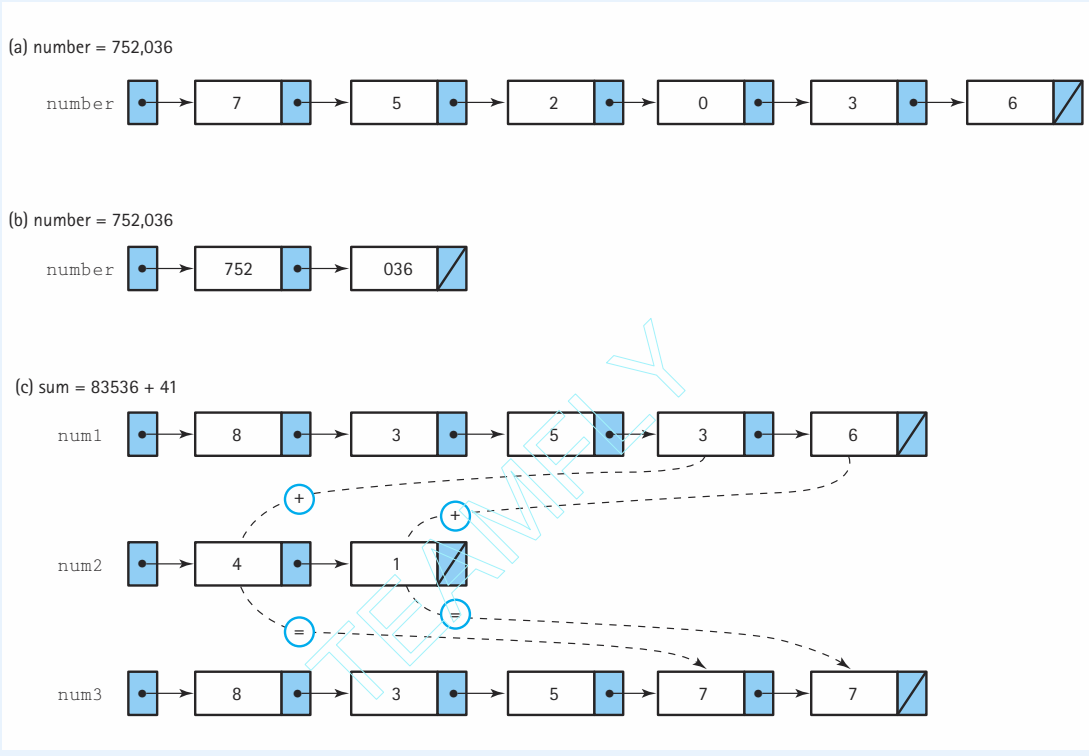


Figure 6.18 *Representing large integers with linked lists*

several reasons. Most importantly, those lists were for keyed elements and carried the restriction that duplicate elements were not allowed. Obviously, we need to allow duplicate digits in a large integer. Additionally, we need to know that the order in which we insert digits into a large integer is preserved by the representation. None of our generic list implementations can guarantee that property. So, we must define a special-purpose list class just for our large integers. What operations must be supported by this class?

Note!! For this case study we are developing a Large Integer ADT that can be used by any application program that requires large integers. This ADT provides operations to build large integers, perform arithmetic operations on large integers, and return strings representing large integers. Now we are discussing using a List ADT to hold the underlying representation of a large integer. In other words, the application program uses the Large Integer ADT since it provides large integers, and the Large Integer ADT uses the List ADT since it provides a list of `byte` values. See Figure 6.19.

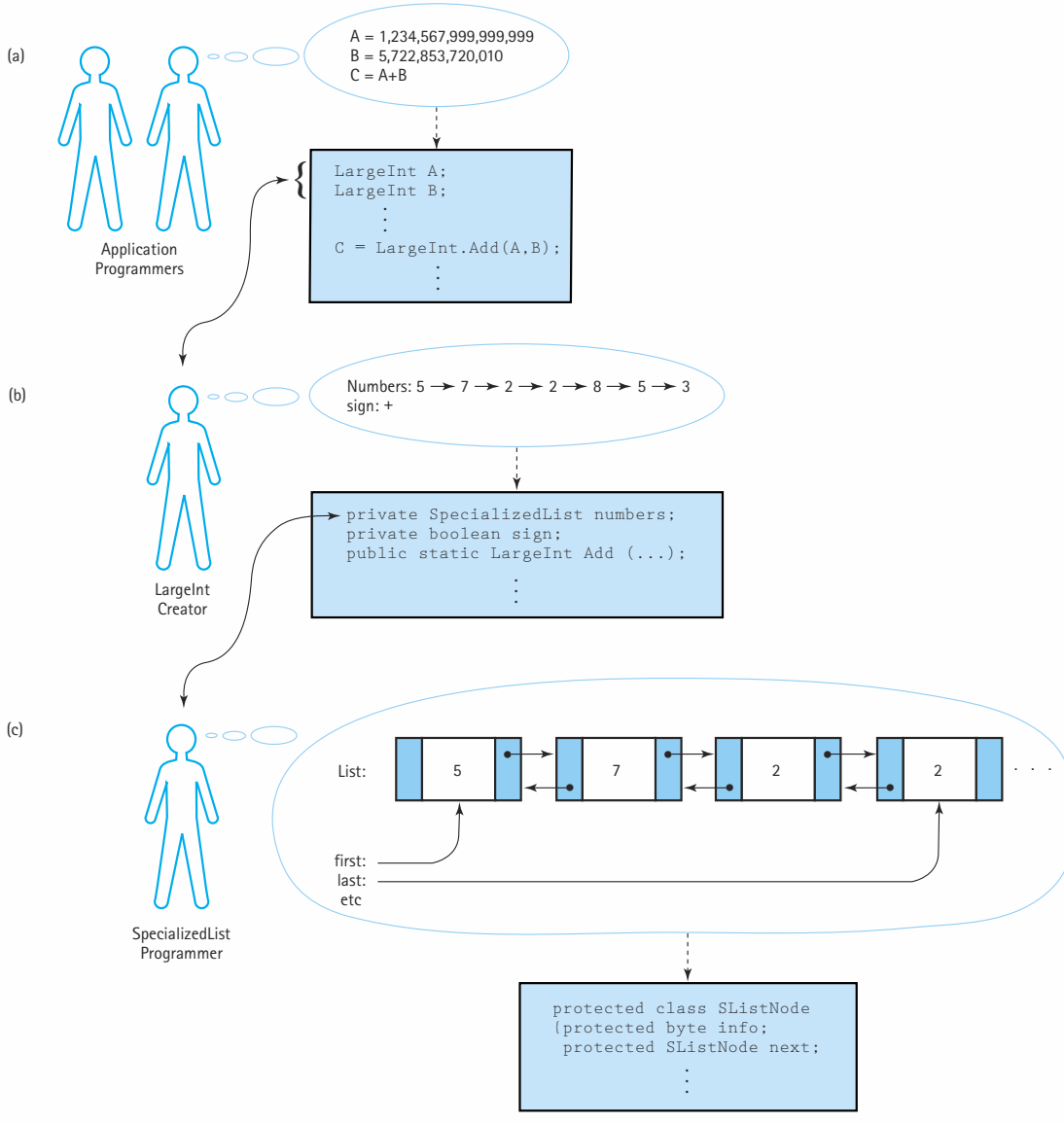


Figure 6.19 *Three points of view of large integers*

The first thing to consider is how large integer objects are to be constructed. We have already decided that we need to be able to build our representation, one digit at a time, working from left to right across a particular large integer. That is how we initialize large integers directly. But large integers can also be created as a result of arithmetic operations. Think about how you perform arithmetic operations such as addition—you work from the least significant digit to the most significant digit, building the result of the addition from right to left as you proceed. Therefore, we also need to create large integers by inserting digits from least significant to most significant; this enables us to construct an object that is the result of an arithmetic operation. So, our linked list should support insertion of digits at both the beginning and the end of the list.

Now we ask what type of access to the representation do we need? We must be able to access our representation one digit at a time, working from left to right to build a string for a large integer for display. And to support arithmetic operations on large integers, we realize that we must also be able to access the digits of the large integer from right to left. We conclude that we should use a list that supports both forward and backward iterations. Yes, this is beginning to sound familiar.

As stated in the previous section, the requirements set forth for a list to support the representation of large integers are exactly the requirements we used for our `SpecializedList` ADT. For organizational purposes we wanted to present our list implementations before getting into this case study, so we decided to present the ADT before we had developed the impetus for it. Let's review what we stated about the specifications in the previous section: "The lists must hold elements of type `byte`; duplicate elements are allowed. The lists need not support `isFull`, `isThere`, `retrieve`, or `delete`. In fact, the only operation that we have been using that is still required by this new list construct is the `lengthIs` operation. For the case study, we are going to need to process elements from left to right and from right to left. In addition, we are going to need to insert items at the front and at the back of our lists."

The LargeInt Class Now that we know we can use the `SpecializedList` class to hold the list of digits for a large integer, we can concentrate on the rest of the definition of the large integer class. In addition to digits, integers also have a sign, an indication of whether they are positive or negative. We represent the sign of a large integer with a boolean instance variable `sign`. Furthermore, we define two `boolean` constants, `PLUS = true` and `MINUS = false`, to use with `sign`.

Here is the first approximation of the beginning of the class `LargeInt`. It includes the instance variables, constructor, and methods `setNegative` (to make a large integer negative), `addDigit` (to build a large integer digit by digit), and `toString` (to provide a string representation of a large integer, complete with commas separating every three digits). We place it in the package `ch06.largeInts`. Since it uses the `SpecializedList` class it must import the `ch06.byteLists` package.

```
//---------------------------------------------------------------------------
// LargeInt.java              by Dale/Joyce/Weems                  Chapter 6
//
// Provides a Large Integer ADT. Large Integers can consist of any number
//  of digits, plus a sign. Supports an add and a subtract operation.
//---------------------------------------------------------------------------
```

```
package ch06.largeInts;

import ch06.byteLists.*;

public class LargeInt
{
  private SpecializedList numbers;    // Holds digits

  // Constants for sign variable
  private static final boolean PLUS = true;
  private static final boolean MINUS = false;

  private boolean sign;

  public LargeInt()
  {
    numbers = new SpecializedList();
    sign = PLUS;
  }

  public void setNegative()
  {
    sign = MINUS;
  }

  public void addDigit(byte digit)
  {
    numbers.insertEnd(digit);
  }
  public String toString()
  {
    String largeIntString;
    if (sign == PLUS)
      largeIntString = "+";
    else
      largeIntString = "-";

    int length;
    length = numbers.lengthIs();
    numbers.resetForward();
    for (int count = length; count >= 1; count--)
    {
      largeIntString = largeIntString + numbers.getNextItem();
      if ((((count - 1) % 3) == 0) && (count != 1))
       largeIntString = largeIntString + ",";
    }
    return(largeIntString);
  }
```

Note that classes in a program typically exhibit one of the following relationships: they are independent of each other, they are related by composition, or they are related by inheritance. Class `LargeInt` and class `SpecializedList` are related by composition. As you see in the beginning of the class definition, a `LargeInt` object is composed of (or contains) a `SpecializedList` object. Just as inheritance expresses an *is a* relationship (a `SortedLinkedList` object is a `LinkedList` object), composition expresses a *has a* relationship (a `LargeInt` object has a `SpecializedList` object inside it).

Addition and Subtraction Do you recall when you learned addition of integers? Remember how there were many special rules depending on the signs of the operands and which operand had the larger absolute value? For example, to perform the addition (−312) + (+200), what steps would you take? Let's see … the numbers have unlike signs, therefore you subtract the smaller absolute value (200) from the larger absolute value (312), giving you 112, and use the sign of the larger absolute value (−), giving the final result of (−112). Here, try a couple more additions:

(+200) + (+100) = ?
(–300) + (–134) = ?
(+34) + (–62) = ?
(–34) + (+62) = ?

The answers are (+300, −434, −28, +28) right?

Did you notice anything about the actual arithmetic operations that you had to perform to calculate the results of the summations listed above? You only performed two kinds of operations: adding two positive numbers and subtracting a smaller positive number from a larger positive number. That's it. When you learned arithmetic, you learned these two basic operations and used them, in combination with rules about how to handle signs, to do all of your sums.

Helper Methods In programming, we also like to factor out and reuse common operations. Therefore, to support the general addition operation we first define a few helper operations. First, note that the base operations we need should apply to the absolute values of our numbers. This means we can just use the `numbers` object (the `SpecializedList` of digits) and ignore the `sign` for now. Second, what common operations do we need? Based on the above discussion, we need to be able to add together two lists of digits, and to subtract a smaller list from a larger list. That means we also have to be able to identify which of two lists of digits is larger. So, we need three operations, which we call `addLists`, `subtractLists`, and `greaterList`.

Let's begin with `greaterList`. It is probably the simplest of the three helper methods. We pass `greaterList` two `SpecializedList` arguments and it returns `true` if the first represents a larger number than the second, and `false` otherwise. When comparing strings, we compare the characters in each character position one at a time from left to right. The first characters that do not match determine which string comes first. When comparing positive numbers (we are ignoring the sign for now), we only have to compare the numbers digit by digit if they are the same length. We first compare the lengths, and if they are not the same,

we return the appropriate result. If the number of digits is the same, we compare the digits from left to right. The first unequal pair determines the result in this case. In the code, we originally set a `boolean` variable `greater` to `false`, and only change this setting if we discover that the first number is not larger than the second number. In the end, we return the `boolean` value of the `greater` variable.

```
private static boolean greaterList(SpecializedList first,
                                   SpecializedList second)
// Effect: returns true if first represents a larger number than second,
//         otherwise returns false;
// Precondition: no leading zeros
{
  boolean greater = false;
  if (first.lengthIs() > second.lengthIs())
    greater = true;
  else
  if (first.lengthIs() < second.lengthIs())
    greater = false;
  else
  {
    byte digitFirst;
    byte digitSecond;
    first.resetForward();
    second.resetForward();

    // Set up loop
    int length = first.lengthIs();
    boolean keepChecking = true;
    int count = 1;

    while ((count <= length) && (keepChecking))
    {
      digitFirst = first.getNextItem();
      digitSecond = second.getNextItem();
      if (digitFirst > digitSecond)
      {
        greater = true;
        keepChecking = false;
      }
      else
      if (digitFirst < digitSecond)
      {
        greater = false;
        keepChecking = false;
      }
```

```
        count++;
      }
    }
    return greater;
  }
```

Notice that if we get the whole way through the *while* loop without finding a difference between any digit pairs, the numbers are equal and we return the original value of `greater`, which is `false` (since the `first` is not greater than the `second`). Also note that since we blindly look at the lengths of the lists we must assume that the numbers do not include leading zeros (for example, the method would report that 005 > 14). Finally, note that the `greaterList` method is private—helper methods are not intended for use by the client programmer; they are only intended for use within the `LargeInt` class itself.

Let's look at `addLists` next. We pass `addLists` its two summation operands as `SpecializedList` parameters and it returns a new `SpecializedList` as the result. The processing for `addLists` can be simplified if we assume that the first argument is larger (or equal to) the second argument. This simplifies traversing the lists. Since we already have access to a `greaterList` method, we make this simplifying assumption.

We begin by adding the two least significant digits (the units position). Next, we add the digits in the tens position (if present) plus the carry from the sum of the least significant digits (if any). This process continues until we finish with the digits of the smaller operand. Then we continue the same sort of processing until we finish with the digits of the larger operand. Finally, if there is still a carry value left over, we add it to the most significant location. We use integer division and modulus operators to determine the carry value and the value to insert into the result. The algorithm is:

addLists (SpecializedList larger, SpecializedList smaller) returns SpecializedList

```
Set result to new SpecializedList();
Set carry to 0;
larger.resetBackward( );
smaller.resetBackward( );

for the length of the smaller list
  Set digit1 to larger.getPriorItem( );
  Set digit2 to smaller.getPriorItem( );
  Set temp to digit1 + digit2 + carry
  Set carry to temp/10
  result.insertFront(temp % 10)
Finish up digits in larger, adding carries if necessary
if (carry != 0))
  result.insertFront(carry)
return result
```

Apply the algorithm to the following examples to convince yourself that it works. The code follows.

```
322    388    399     999     3      1     988    0
 44    108      1      11    44     99     100    0
---    ---    ---    ----    --    ---    ----    -
366    496    400    1010    47    100    1088    0
```

```
private static SpecializedList addLists(SpecializedList larger,
                                        SpecializedList smaller)
// Returns a specialized list that is a byte-by-byte sum of the two
//  argument lists
// Assumes larger >= smaller
{
  byte digit1;
  byte digit2;
  byte temp;
  byte carry = 0;

  int largerLength = larger.lengthIs();
  int smallerLength = smaller.lengthIs();
  int lengthDiff;

  SpecializedList result = new SpecializedList();

  larger.resetBackward();
  smaller.resetBackward();

  // Process both lists while both have digits
  for (int count = 1; count <= smallerLength; count++)
  {
    digit1 = larger.getPriorItem();
    digit2 = smaller.getPriorItem();
    temp = (byte)(digit1 + digit2 + carry);
    carry = (byte)(temp / 10);
    result.insertFront((byte)(temp % 10));
  }

  // Finish processing of leftover digits
  lengthDiff = (largerLength - smallerLength);
  for (int count = 1; count <= lengthDiff; count++)
  {
    digit1 = larger.getPriorItem();
    temp = (byte)(digit1 + carry);
    carry = (byte)(temp / 10);
    result.insertFront((byte)(temp % 10));
  }
```

```
    if (carry != 0)
      result.insertFront((byte)carry);

    return result;
  }
```

Now let's examine subtracting lists. Remember that for our helper method `subtractLists` we are handling only the simplest case: both integers are positive and the smaller one is subtracted from the larger one. As with `addLists`, we accept two `SpecializedList` parameters, the first being larger than the second, and we return a new `SpecializedList`. Again, we begin with the digits in the units position. Let's call the digit in the larger argument `digit1` and the digit in the smaller argument `digit2`. If `digit2` is less than `digit1`, we subtract and insert the resulting digit at the front of the result. If `digit2` is greater than `digit1`, we borrow 10 and subtract. Then we access the digits in the tens position. If we have borrowed, we subtract 1 from the new `digit1` and proceed as before. Because we have limited our problem to the case where `digit1` is larger than `digit2`, both either run out of digits together or `digit1` still contains digits when `digit2` has been processed. Also note that this constraint guarantees that borrowing does not extend beyond the most significant digit of `digit1`. See if you can follow the algorithm we just described in the code.

```
private static SpecializedList subtractLists(SpecializedList larger,
                                             SpecializedList smaller)
// Returns a specialized list that is the difference of the two argument lists
//  Assumes larger >= smaller
{
  byte digit1;
  byte digit2;
  byte temp;
  boolean borrow = false;

  int largerLength = larger.lengthIs();
  int smallerLength = smaller.lengthIs();
  int lengthDiff;

  SpecializedList result = new SpecializedList();

  larger.resetBackward();
  smaller.resetBackward();

  // Process both lists while both have digits.
  for (int count = 1; count <= smallerLength; count++)
  {
    digit1 = larger.getPriorItem();
    if (borrow)
```

```
    {
      if (digit1 != 0)
      {
        digit1 = (byte)(digit1 - 1);
        borrow = false;
      }
      else
      {
        digit1 = 9;
        borrow = true;
      }
    }

    digit2 = smaller.getPriorItem();

    if (digit2 <= digit1)
      result.insertFront((byte)(digit1 - digit2));
    else
    {
      borrow = true;
      result.insertFront((byte)(digit1 + 10 - digit2));
    }
  }

  // Finish processing of leftover digits
  lengthDiff = (largerLength - smallerLength);
  for (int count = 1; count <= lengthDiff; count++)
  {
    digit1 = larger.getPriorItem();
    if (borrow)
    {
      if (digit1 != 0)
      {
        digit1 = (byte)(digit1 - 1);
        borrow = false;
      }
      else
      {
        digit1 = 9;
        borrow = true;
      }
    }
    result.insertFront(digit1);
  }

  return result;
}
```

Addition Now that we have finished the helper methods, we can turn our attention to the public methods provided to clients of the `LargeInt` class. First, addition. Here are the rules for addition you learned when studying arithmetic.

Addition Rules

1. If both operands are positive, add the absolute values and make the result positive
2. If both operands are negative, add the absolute values and make the result negative
3. If one operand is negative and one operand is positive, subtract the smaller absolute value from the larger absolute value and give the result the sign of the larger absolute value.

We use these rules to help us design our `add` method. Note that we can combine the first two rules into "If the operands have the same sign, add the absolute values and make the sign of the result the same as the sign of the operands." Our code uses the appropriate helper method to generate the new list of digits and then sets the sign based on the rules. Remember that to use our helper methods we pass them the required arguments in the correct order (larger first). Here is the code for `add`:

```
public static LargeInt add(LargeInt first, LargeInt second)
// Returns a LargeInt that is the sum of the two argument LargeInts
{
  LargeInt sum = new LargeInt();

  if (first.sign == second.sign)
  {
    if (greaterList(first.numbers, second.numbers))
      sum.numbers = addLists(first.numbers, second.numbers);
    else
      sum.numbers = addLists(second.numbers, first.numbers);
    sum.sign = first.sign;
  }
  else   // Signs are different
  {
    if (greaterList(first.numbers, second.numbers))
    {
      sum.numbers = subtractLists(first.numbers, second.numbers);
      sum.sign = first.sign;
    }
    else
    {
      sum.numbers = subtractLists(second.numbers, first.numbers);
      sum.sign = second.sign;
    }
  }

  return sum;
}
```

The add method accepts two `LargeInt` objects and returns a new `LargeInt` object equal to their sum. Since it is passed both summation operands as parameters, and since it returns the result explicitly, it is defined as a `static` method. It is invoked through the class, rather than through an object of the class. For example, the code

```
LargeInt LI1 = new LargeInt();
LargeInt LI2 = new LargeInt();
LargeInt LI3;

LI1.addDigit((byte)9);
LI1.addDigit((byte)9);
LI1.addDigit((byte)9);

LI2.addDigit((byte)9);
LI2.addDigit((byte)8);
LI2.addDigit((byte)7);

LI3 = LargeInt.add(LI1, LI2);

System.out.println("LI3 is " + LI3);
```

would result in the output of the string "LI3 is +1,986".

Subtraction Remember how subtraction seemed harder than addition when you were learning arithmetic? Not anymore. We only need to use one subtraction rule: "Change the sign of the subtrahend, and add." We do have to be careful about how we "change the sign of the subtrahend," because we do not want to change the sign of the actual argument passed to `subtract`, since that would produce an unwanted side effect of our method. Therefore, we create a new `LargeInt` object, make it a copy of the second parameter, invert its sign, and then invoke `add`:

```
public static LargeInt subtract(LargeInt first, LargeInt second)
// Returns a LargeInt that is the difference of the two argument LargeInts
{
  LargeInt diff = new LargeInt();

  // Create an inverse of second
  LargeInt negSecond = new LargeInt();
  negSecond.sign = !second.sign;
  second.numbers.resetForward();
  int length = second.numbers.lengthIs();
  for (int count = 1; count <= length; count++)
    negSecond.numbers.insertEnd(second.numbers.getNextItem());
```

```
    // Add first to inverse of second
    diff = add(first, negSecond);

    return diff;
  }
```

Test Plan

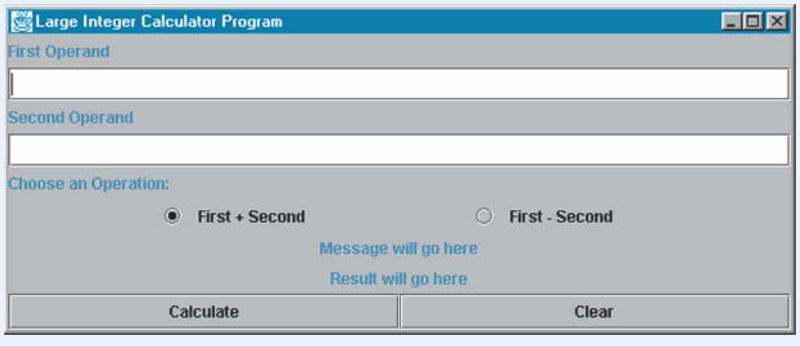
Each `LargeInt` operation must be unit tested. The complexity of the code for each operation is evident in the number of *if* statements in the corresponding method. The more complex the code, the more test cases are necessary to test it. A white-box strategy would require going through the code of each operation and determining data that test at least all branches (including the branches of any helper methods). A black-box strategy involves choosing data that test the varieties of possible input. This would involve varying combinations of signs and relative relationships between the absolute values of operands. The examples used in the discussion can serve as test data for those operations. However, other end cases should be included, such as cases in which one or both of the operands are zero, or the expected result is zero.

It is not difficult to create a test driver for the `LargeInt` class by modifying one of our previous test drivers. However, in keeping with the approach we've established in our case studies, we have created an application with a graphical interface that uses `LargeInt`. Using this application, a Large Integer Calculator, you can test whatever combinations of large integers and operations that you want.

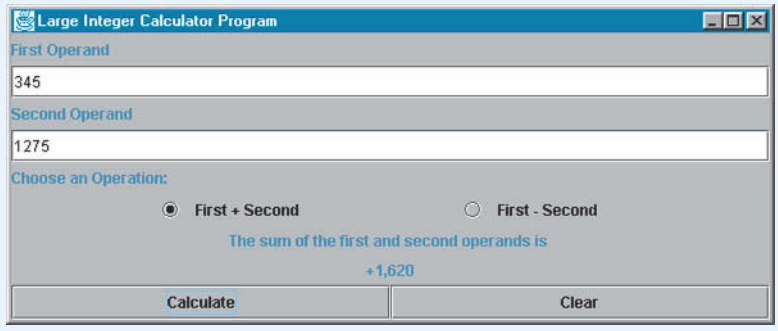
A Large Integer Calculator

The Large Integer Calculator allows the user to enter two large integers and perform either addition or subtraction. Here are some screen shots, to give you a feeling for the application.

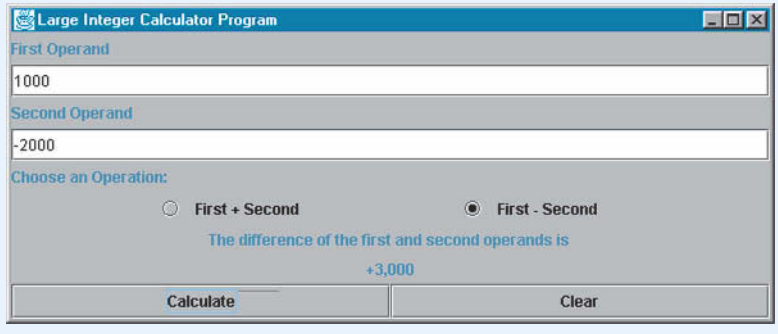
The user first sees:



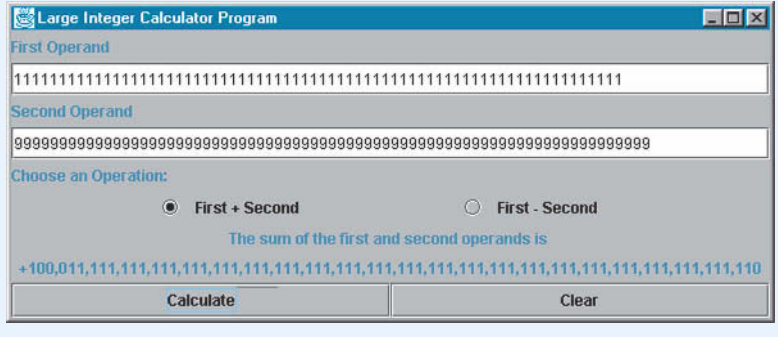
Here is the result of entering two operands, choosing addition, and pressing Calculate:



How about subtraction?



Wait a second, is that answer correct? Of course … remember that 1000 − (−2000) = 1000 + 2000. Let's look at an example using really big integers; that is the purpose of the Large Integer ADT:



The code for the Large Integer Calculator is included on the web site. Since it is an application program, we do not place it in a package. It is in the `bookFiles.ch06` subdirectory. You are encouraged to try it for yourself. If you do, you may discover a few problem situations. These situations form the basis for some section exercises.

The Code

In the Large Integer Calculator code we use a helper method, `getLargeInt`, which accepts a string argument and returns the corresponding `LargeInt` object. This method is passed the strings from the operand text fields. Even if we had used our standard test approach, creating a test driver that takes its test input from a file, we would have needed a similar method. It might be a good idea to consider enhancing our Large Integer ADT with such an operation, so that we do not have to continually create it every time we use the ADT.

Study the code below. See if you can find the statements that declare, instantiate, initialize, transform, and observe Large Integers.

The user interface code for the Large Integer Calculator is similar to the user interface code for the Postfix Expression Evaluator of Chapter 4, since both applications provide a type of calculator/evaluator. The only new construct, radio buttons, is described in the Java I/O IV feature section below. The lines of code directly related to radio buttons are emphasized.

```
//--------------------------------------------------------------------------
// LargeIntCalculator.java         by Dale/Joyce/Weems                Chapter 6
//
// Evaluates addition and subtraction of large integers
//--------------------------------------------------------------------------

import ch06.largeInts.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;

public class LargeIntCalculator
{
  // Large integers
  private static LargeInt first;
  private static LargeInt second;
  private static LargeInt result;

  private static LargeInt getLargeInt(String intString)
  // Returns the large integer indicated in intString
  // Precondition: intString contains a well-formatted integer
  {
    LargeInt temp = new LargeInt();
    int firstDigitPosition;          // Position of first digit in intString
    int lastDigitPosition;           // Position of last digit in intString
```

```
    // Used to translate character to byte
    char digitChar;
    int digitInt;
    byte digitByte;

    firstDigitPosition = 0;
    if (intString.charAt(0) == '+')   //  Skip leading plus sign
      firstDigitPosition = 1;
    else
    if (intString.charAt(0) == '-')   // Handle leading minus sign
    {
      firstDigitPosition = 1;
      temp.setNegative();
    }

    lastDigitPosition = intString.length() - 1;

    for (int count = firstDigitPosition; count <= lastDigitPosition; count++)
    {
      digitChar = intString.charAt(count);
      digitInt = Character.digit(digitChar, 10);
      digitByte = (byte)digitInt;
      temp.addDigit(digitByte);
    }

    return temp;
}

// Text field
private static JTextField operandAText;  // Text field for operand A
private static JTextField operandBText;  // Text field for operand B

// Status Label
private static JLabel statusLabel;         // Label for status info
private static JLabel resultLabel;         // Label for status info

// Radio Buttons and Group for choosing operation
private static JRadioButton plusButton;
private static JRadioButton minusButton;
private static ButtonGroup operationGroup;

// Define a button listener
private static class ActionHandler implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    // Listener for the button events
```

```
    {
      if (event.getActionCommand().equals("Calculate"))
      {
        // Handles Calculate event
        first = getLargeInt(operandAText.getText());
        second = getLargeInt(operandBText.getText());
        result = LargeInt.add(first, second);

        String choice = operationGroup.getSelection().getActionCommand();
        if (choice == "plus")
        {
          statusLabel.setText("The sum of the first and second operands is ");
          result = LargeInt.add(first, second);
        }
        else
        {
          statusLabel.setText("The difference of the first and second operands
                               is ");
          result = LargeInt.subtract(first, second);
        }
        resultLabel.setText(result.toString());
      }
      else
      if (event.getActionCommand().equals("Clear"))
      {
        // Handles Clear event
        statusLabel.setText("cleared");
        resultLabel.setText("cleared");
        operandAText.setText("");
        operandBText.setText("");
      }
    }
  }

  public static void main(String args[]) throws IOException
  {
    // Declare/instantiate/initialize display frame
    JFrame displayFrame = new JFrame();
    displayFrame.setTitle("Large Integer Calculator Program");
    displayFrame.setSize(600,250);
    displayFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // Text box for operands
    operandAText = new JTextField( 60);
    operandBText = new JTextField( 60);
```

```
// Radio Buttons for choosing operation
JRadioButton plusButton = new JRadioButton("   First + Second   ");
plusButton.setSelected(true);
JRadioButton minusButton = new JRadioButton("   First - Second   ");
plusButton.setActionCommand("plus");
minusButton.setActionCommand("minus");
operationGroup = new ButtonGroup();
operationGroup.add(plusButton);
operationGroup.add(minusButton);

// Status/Result labels
statusLabel = new JLabel("Message will go here", JLabel.CENTER);
resultLabel = new JLabel("Result will go here", JLabel.CENTER);

// Various labels
JLabel operandALabel   = new JLabel("First Operand", JLabel.LEFT);
JLabel operandBLabel   = new JLabel("Second Operand", JLabel.LEFT);
JLabel operatorsLabel   = new JLabel("Choose an Operation:", JLabel.LEFT);
JLabel blankLabel1 = new JLabel("");
JLabel blankLabel2 = new JLabel("");
JLabel blankLabel3 = new JLabel("");

// Calculate and clear buttons
JButton calculate   = new JButton("Calculate");
JButton clear       = new JButton("Clear");

// Button event listener
ActionHandler action = new ActionHandler();

// Register button listeners
calculate.addActionListener(action);
clear.addActionListener(action);

// Instantiate content pane and information panels
Container contentPane = displayFrame.getContentPane();
JPanel setupPanel = new JPanel();
JPanel operatorPanel = new JPanel();
JPanel resultPanel = new JPanel();
JPanel buttonPanel = new JPanel();

// Initialize setup panel
setupPanel.setLayout(new GridLayout(6,1));
setupPanel.add(operandALabel);
setupPanel.add(operandAText);
setupPanel.add(operandBLabel);
setupPanel.add(operandBText);
setupPanel.add(operatorsLabel);
operatorPanel.setLayout(new GridLayout(1,5));
```

```
        operatorPanel.add(blankLabel1);
        operatorPanel.add(plusButton);
        operatorPanel.add(blankLabel2);
        operatorPanel.add(minusButton);
        operatorPanel.add(blankLabel3);
        setupPanel.add(operatorPanel);

        // Initialize result panel
        resultPanel.setLayout(new GridLayout(2,1));
        resultPanel.add(statusLabel);
        resultPanel.add(resultLabel);

        // Initialize button panel
        buttonPanel.setLayout(new GridLayout(1,2));
        buttonPanel.add(calculate);
        buttonPanel.add(clear);

        // Set up and show the frame
        contentPane.add(setupPanel, "North");
        contentPane.add(resultPanel, "Center");
        contentPane.add(buttonPanel, "South");

        displayFrame.show();
    }
}
```

Java Input/Output IV

The only new I/O construct used in the Large Integer Calculator program is the *radio button*. The related statements are emphasized in the code listing.

A radio button can be selected or deselected by the user of the application. When selected, a dot appears in the small circle beside the radio button text. A set of radio buttons can be designated as a button group—this means that only one member of the set can be selected at any time. If button A of a group is currently selected and the user selects another button from the group, then button A is automatically deselected. Designating which buttons belong to which groups is done within the program and need not be related to where the radio buttons are displayed on the screen. Of course, a good interface makes the grouping of radio buttons obvious to the user.

The radio buttons, and their button group, are declared in the outer level of the program:

```
private static JRadioButton plusButton;
private static JRadioButton minusButton;
private static ButtonGroup operationGroup;
```

These must be declared at the outer level since they are used in the `ActionHandler` class. We return to that soon. First let's see how the buttons and button group are instantiated and initialized in the main program:

```
JRadioButton plusButton = new JRadioButton("   First + Second   ");
plusButton.setSelected(true);
JRadioButton minusButton = new JRadioButton("   First - Second   ");
plusButton.setActionCommand("plus");
minusButton.setActionCommand("minus");
operationGroup = new ButtonGroup();
operationGroup.add(plusButton);
operationGroup.add(minusButton);
```

You can see how the text for the buttons is initialized through the constructor. Also note the use of the `setSelected` method to set the `plusButton` as selected at the start of the program. It is good form to always have exactly one radio button of a group selected, even at the start of a program.

The `setActionCommand` statements associate an action with a radio button. Note that this action is not dynamically invoked when a radio button is selected; it is just returned by a call to the button's group when needed. We do not need to register the actions with the event listener in this case. In general, radio buttons by themselves should not be used to initiate events. Instead, radio buttons are used to select a single item from a set of items—any actions based on this selection should be initiated by another event, such as pushing a Calculate button.

Finally, note in the preceding code how the buttons are added to the `ButtonGroup` object `operationGroup`. Now, when an event occurs (such as pushing the Calculate or Clear buttons) that generates an action to be handled, the action handler can observe which radio button of the `operationGroup` is selected, if needed:

```
String choice = operationGroup.getSelection().getActionCommand();
if (choice == "plus")
{
  ...
}
else
{
  ...
}
```

Since there are only two radio buttons in our group, we only need one *if* statement here. More buttons could be handled by including *if-else* statements.

The only job left is to lay out the radio buttons in a panel and display them. We use a 1-by-5 grid layout to organize the radio buttons into the `operatorPanel` horizontally, following the pattern <blank label, radio button, blank label, radio button, blank label> to create the spacing desired. The `operatorPanel` is added to the `setupPanel`, which is later added to the north section of the `contentPane`.

```
operatorPanel.setLayout(new GridLayout(1,5));
operatorPanel.add(blankLabel1);
operatorPanel.add(plusButton);
operatorPanel.add(blankLabel2);
operatorPanel.add(minusButton);
operatorPanel.add(blankLabel3);
setupPanel.add(operatorPanel);
```

Summary

The idea of linking the elements in a list has been extended to include lists with header and trailer nodes, circular lists, and doubly linked lists. In addition to using dynamically allocated nodes to implement a linked structure, we looked at a technique for implementing linked structures in an array of nodes. In this technique the links are not references into the free store but indexes into the array of nodes. This type of linking is used extensively in systems software.

The case study at the end of the chapter designed a Large Integer ADT. The number of digits is bounded only by the size of memory. The Large Integer ADT required a specialized list for its implementation; none of the lists developed so far provided the needed functionality. Instead of implementing the specialized list "inside" the large integer implementation, we created a new class `SpecializedList`. This case study provided a good example of how one ADT can be implemented by another ADT and thus of the importance of viewing systems as a hierarchy of abstractions.

Although a linked list can be used to implement almost any list application, its real strength is in applications that largely process the list elements in order. This is not to say that we cannot do "random access" operations on a linked list. Our specifications include operations that logically access elements in random order. For instance, public methods `retrieve` and `delete` manipulate an arbitrary element in the list. However, at the implementation level the only way to find an element is to search the list, beginning at the first element and continuing sequentially to examine element after element. This search is O(N), because the amount of work required is directly proportional to the number of elements in the list. A particular element in a sequentially sorted list in an array, in contrast, can be found with a binary search, decreasing the search algorithm to O($\log_2 N$). For a large list, the O(N) sequential search can be quite time-consuming. There is a linked structure that supports O($\log_2 N$) searches: the binary search tree. We discuss the binary search tree data structure in detail in Chapter 8.

Summary of Classes and Support Files

The classes and files are listed in the order in which they appear in the text. The package a class belongs to, if any, is listed in parentheses under Notes. The class and support

files are available on our web site. They can be found in the `ch06` subdirectory of the `bookFiles` directory. Note that inner classes are not included in the table.

Classes, Interfaces and Support Files Defined in Chapter 6

File	1st Ref.	Notes
`CircularSortedLinkedList.java`	page 408	(`ch06.genericLists`) Linked list based on an array of nodes; implements `ListInterface`
`TwoWayListInterface.java`	page 422	(`ch06.genericLists`) An interface that specifies a doubly linked list; partial implementation details are presented and implementation completion is left as an exercise
`ArrayLinkedList.java`	page 429	(`ch06.genericLists`) Linked list based on an array of nodes; implements `ListInterface`
`SpecializedListInterface.java`	page 434	(`ch06.byteLists`) An interface that specifies the list needed to support the chapter's case study
`SpecializedList.java`	page 437	(`ch06.byteLists`) Implements `SpecializedListInterface` using a dynamic doubly linked list with references to the first and last nodes
`LargeInt.java`	page 444	(`ch06.largeInts`) Provides large integers and some arithmetic operations
`LargeIntCalculator.java`	page 456	Allows user to enter two operands and choose an operation.

Figure 6.20 shows our current set of list-related classes and their relationships. The figure does not show the `SpecializedList` class developed in Section 6.5 since it is a standalone class, and is not related to any of the other list classes.

On page 465 is a list of the Java Library Classes that were used in this chapter for the first time in the textbook. The classes are listed in the order in which they are first used. We only list classes used in our programs, not classes just mentioned in the text. Note that in some classes the methods listed might not be defined directly in the class; they might be defined in one of its superclasses. With the methods we also list constructors, if appropriate. For more information about the library classes and methods, the reader can check Sun's Java site.

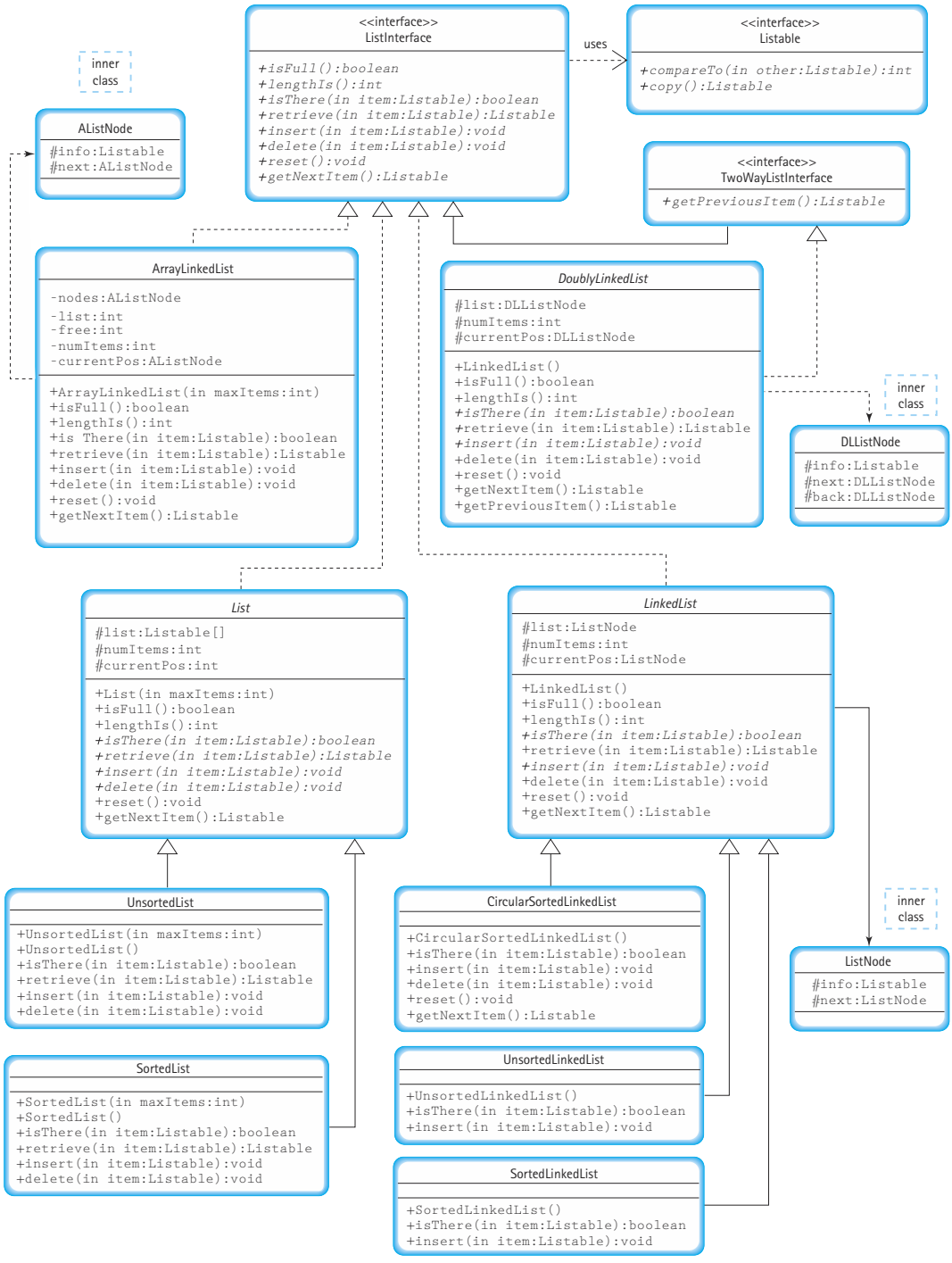


Figure 6.20 *Our current set of list related classes*

Library Classes Used in Chapter 6 for the First Time:

Class Name	Package	Overview	Methods Used	Where Used
`JRadioButton`	`swing`	Radio button objects	`JRadioButton`, `setSelected`, `setActionCommand`	`LargeIntCalculator`
`ButtonGroup`	`awt`	Allows grouping of button objects	`ButtonGroup`, `add`, `getSelection`	`LargeIntCalculator`

Exercises

6.1 Circular Linked Lists

1. In our `CircularSortedLinkedList` class, we reuse the constructor and the `isFull`, `lengthIs`, and `retrieve` methods of our `LinkedList` class. The way we reused the constructor was different from the way we reused the methods. Explain why and how.
2. Implement a copy constructor for the circular linked list class. The signature for your method should be:

   ```
   public CircularSortedLinkedList(CircularSortedLinkedList inList)
   ```

3. If you were going to implement the FIFO Queue ADT as a circular linked list with the reference accessing the "rear" node of the queue, which public methods would require changing?
4. Write a public method `printReverse` that prints the elements of a `CircularSortedLinkedList` object in reverse order. For instance, for the list X Y Z, `list.printReverse()` would output Z Y X. Assume that the list elements all have an associated `toString` method. You may use as a precondition that the list is not empty.
5. Suppose we define an operation on a list called `inBetween` that accepts an item as a parameter and returns true if the item is "in between" the smallest and largest list elements. That is, based on the `compareTo` method defined for list elements, the item is larger than the smallest list element and smaller than the largest list element. Otherwise, the method returns false (even if the item "matches" the smallest or largest element).
   a. Design and code `inBetween` as client code, using operations of the `SortedList` class (the array-based sorted list class from Chapter 3).
   b. Design and code `inBetween` as a public method of the `SortedList` class.

c. Design and code `inBetween` as a public method of the `SortedLinkedList` class.

d. Design and code `inBetween` as a public method of the `CircularSortedLinkedList` class.

e. State the Big-O complexity of each of your implementations in terms of *N*, the size of the list.

6. We implemented the `CircularSortedLinkedList` by maintaining a single reference into the linked list, to the last element of the list. Suppose we changed our approach to maintaining two references into the linked list, one to the first list element and one to the last list element.

   a. Would the new approach necessitate a change in the class constructor? If so, describe the change.

   b. Would the new approach necessitate a change in the `getNextItem` method? If so, describe the change.

   c. Would the new approach necessitate a change in the `isThere` method? If so, describe the change.

   d. Would the new approach necessitate a change in the `insert` method? If so, describe the change.

7. At the end of the section on circular linked lists, it was suggested that many of the claimed benefits could also be obtained by simply augmenting our `SortedLinkedList` class with a private variable that references the last element of the list. Outline the changes to the public methods of the `SortedLinkedList` class that should be made due to such a change. For each change, identify whether the change is necessary to support the new implementation, or whether it is an improvement made possible by the new implementation.

6.2 Doubly Linked Lists

8. We discussed the Insert operation for a doubly linked list and showed that the correct order for the reference changes is:

```
newNode.back = location.back;
newNode.next = location;
location.back.next = newNode;
location.back = newNode;
```

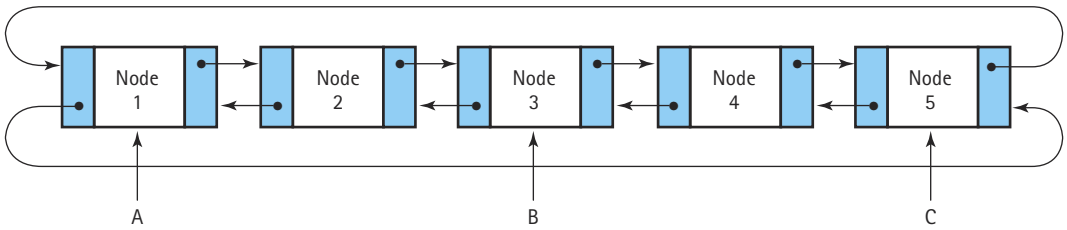
Describe the ramifications of making the reference changes in a different order as shown here:

a.
```
location.back = newNode;
newNode.back = location.back;
newNode.next = location;
location.back.next = newNode;
```

b.
```
newNode.back = location.back;
location.back.next = newNode;
newNode.next = location;
location.back = newNode;
```

c.
```
newNode.next = location;
newNode.back = location.back;
location.back.next = newNode;
location.back = newNode;
```

9. We discussed the implementation of the `insert` method for a sorted doubly linked list class. Implement the method in Java.
10. We discussed the implementation of the `delete` method for a sorted doubly linked list class. Implement the method in Java.
11. Outline the changes to the other public methods of the Sorted List ADT (besides `insert` and `delete`) that should be made to implement the Sorted List as a doubly linked list instead of as a singly linked list. For each change, identify whether the change is necessary because of the new implementation, or whether it is an improvement made possible by the new implementation.
12. Using the circular doubly linked list below, give the expression corresponding to each of the following descriptions.



For example, the expression for the info member of Node 1, referenced from reference A, would be `A.info`.

a. The info member of Node 1, referenced from reference C

b. The info member of Node 2, referenced from reference B

c. The next member of Node 2, referenced from reference A

d. The next member of Node 4, referenced from reference C

e. Node 1, referenced from reference B

f. The back member of Node 4, referenced from reference C

g. The back member of Node 1, referenced from reference A

6.3 Linked Lists with Headers and Trailers

13. Dummy nodes are used to simplify list processing by eliminating some "special case."
    a. What special case is eliminated by a header node in a reference-based linked list?
    b. What special case is eliminated by a trailer node in a reference-based linked list?
    c. Would dummy nodes be useful in implementing a linked stack? That is, would their use eliminate a special case?
    d. Would dummy nodes be useful in implementing a linked queue with a reference to both the head and the rear elements?
    e. Would dummy nodes be useful in implementing a circular linked queue?
14. Of the three variations of linked lists (circular, with header and trailer nodes, and doubly linked), which would be most appropriate for each of the following applications?
    a. You must search a list for a key and return the keys of the two elements that come before it and the keys of the two elements that come after it.
    b. A text file contains integer elements, one per line, sorted from smallest to largest. You must read the values from the file and create a sorted linked list containing the values.
    c. A list that is short and frequently becomes empty. You want a list that is optimum for inserting an element into the empty list and deleting the last element from the list.
15. John and Mary are programmers for the local school district. One morning John commented to Mary about the funny last name the new family in the district had: "Have you ever heard of a family named Zzuan?" Mary replied "Uh oh; we have some work to do. Let's get going." Can you explain Mary's response?

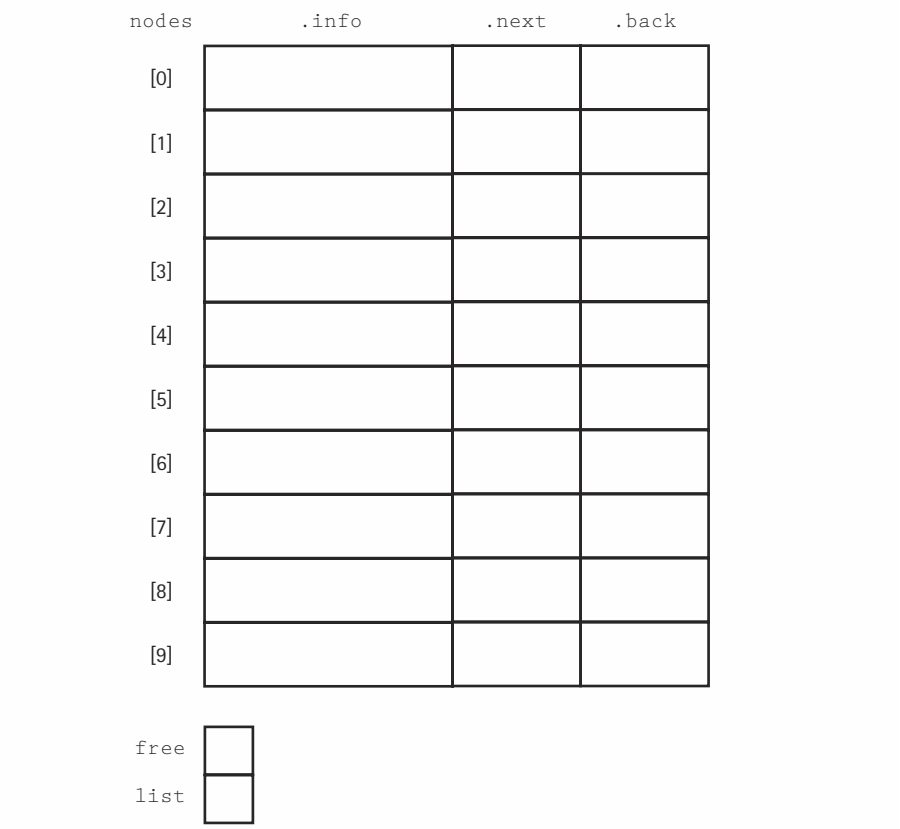
6.4 A Linked List as an Array of Nodes

16. What is the Big-O measure for initializing the free list in the array-based linked implementation? For the methods `getNode` and `freeNode`?
17. Use the linked lists contained in the array pictured in Figure 6.13 to answer the following questions:
    a. What elements are in the list pointed to by `list1`?
    b. What elements are in the list pointed to by `list2`?
    c. What array positions (indexes) are part of the free space list?
    d. What would the array look like after the deletion of "Nell" from the first list?

e. What would the array look like after the insertion of "Anne" into the second list? Assume that before the insertion that the array is as pictured in Figure 6.13.

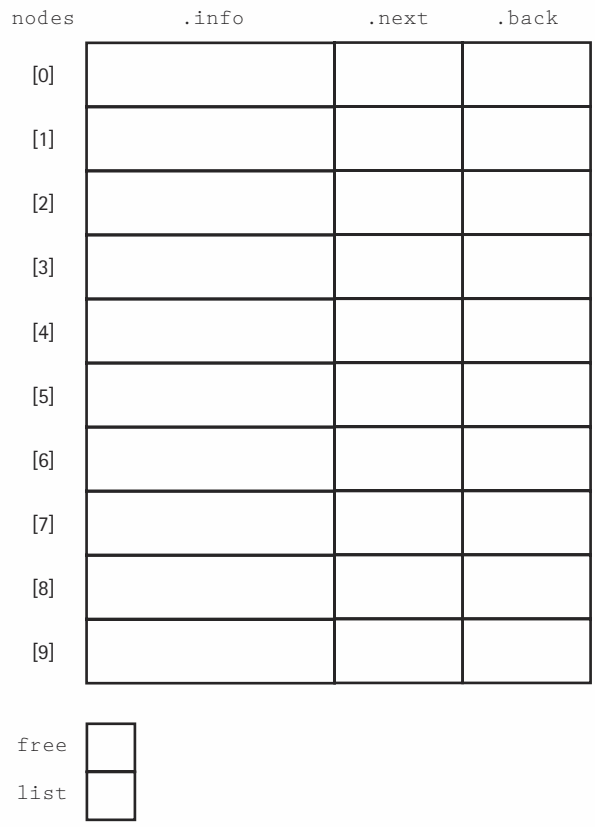
18. An array of records (nodes) is used to contain a doubly linked list, with the next and back members indicating the indexes of the linked nodes in each direction.

a. Show how the array would look after it was initialized to an empty state, with all the nodes linked into the free-space list. (Note that the free-space nodes only have to be linked in one direction.)

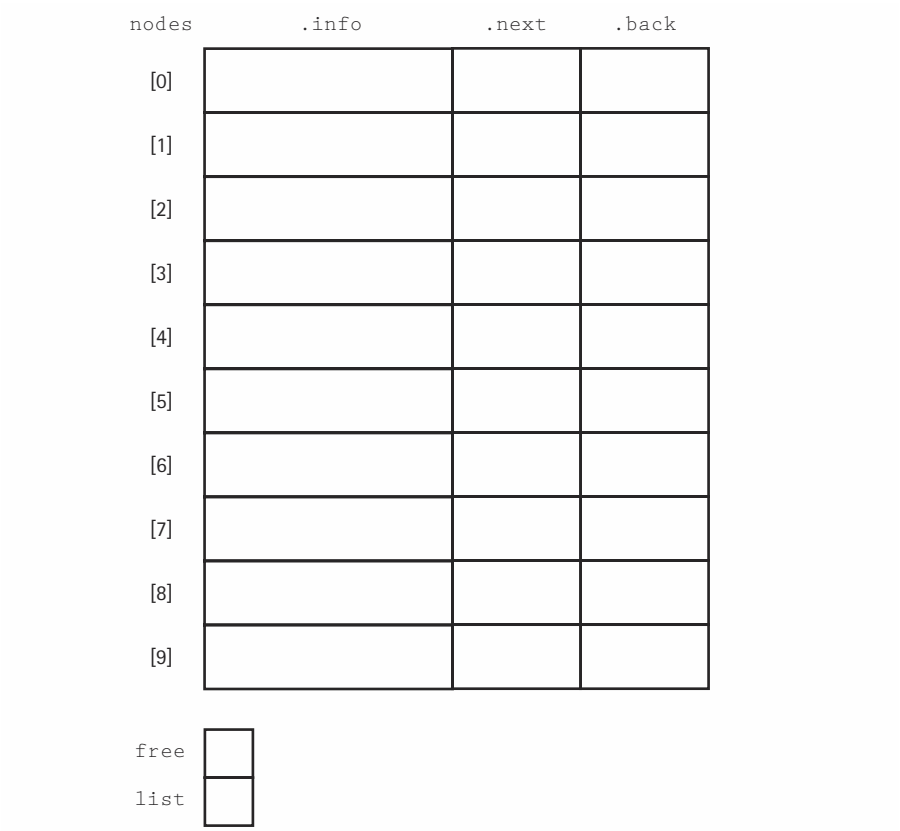


b. Draw a box-and-arrow picture of an abstract doubly linked list into which the following numbers are inserted into their proper places: 17, 4, 25.

c. Show how the array in part (a) would look after the listed numbers are inserted into their proper places in the doubly linked list: 17, 4, 25.



d. Show how the array in part (c) would look after 17 is deleted from the doubly linked list.



19. We developed code for the constructor and the `getNode`, `freeNode`, `isFull`, and `delete` methods of our `ArrayLinkedList` class. Develop the code for
    a. the `lengthIs` method
    b. the `isThere` method
    c. the `retrieve` method
    d. the `insert` method
    e. the `reset` method
    f. the `getNextItem` method

6.5 A Specialized List ADT

20. True or False? The `SpecializedList` class
    a. uses the "by copy" approach with its elements.
    b. implements the `ListInterface` interface.
    c. keeps its data elements sorted.

d. allows duplicate elements.

e. throws an exception if an iteration "walks off" the end of the list.

f. can hold objects of any Java class.

g. has only O(1) operations, including its constructor.

21. Describe the difference between the `getPriorItem` method of the `SpecializedList` class and the proposed `getPreviousItem` method of the `DoublyLinkedList` class.

22. Can you derive a class `DLList` from the class `SpecializedList` that has a public method `insert` that inserts the item into its proper place in the list? If so, derive the class and implement the method. If not, explain why not.

A Case Study: Large Integers

23. Discuss the changes that would be necessary within the `LargeInt` class if more than one digit is stored per node.

24. Implement a copy constructor for the `LargeInt` class. The signature for your method should be:

```
public LargeInt(LargeInt inLargeInt)
```

Does the existence of this copy constructor affect the design of any of the methods of the Large Integer Calculator program? If so, which ones, and how?

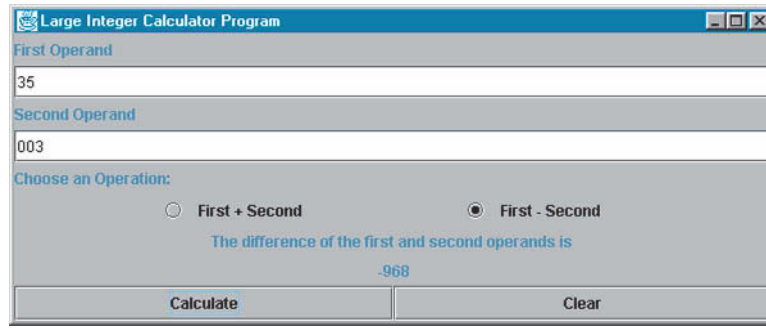
25. Consider the multiplication of large integers.

a. Describe an algorithm.

b. Implement a multiply method for the `LargeInt` class.

c. Add multiplication as an option for the Large Integer Calculator program.

26. The private method `greaterList` of the `LargeInt` class assumes that its arguments have no leading zeros. When this assumption is violated, strange results can occur. Consider the following screen shot from the Large Integer Calculator program that shows that 35 − 3 is −968:



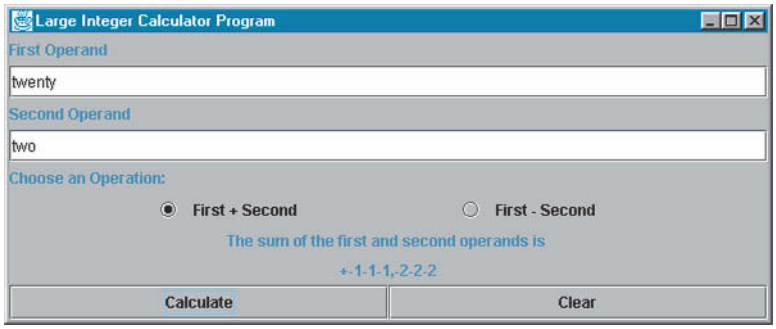
a. Why do leading zeros cause a problem?

b. Identify at least two approaches to correcting this problem.

c. Describe benefits and drawbacks to each of your identified approaches.

d. Choose one of your approaches and implement the solution.

27. The Large Integer Calculator program does not "catch" ill-formatted input like the Postfix Expression Evaluator program did. For example, consider the following screen shot:



Fix the program so that it is more robust, and in situations such as that shown above it writes an appropriate error message to the display.

Chapter 7

Programming with Recursion

Goals

Measurable goals for this chapter include that you should be able to

- discuss recursion as another form of repetition
- do the following, given a recursive method:
  - determine whether the method halts
  - determine the base case(s)
  - determine the general case(s)
  - determine what the method does
  - determine whether the method is correct and, if it is not, correct it
- do the following, given a recursive-problem description:
  - determine the base case(s)
  - determine the general case(s)
  - design and code the solution as a recursive void or value-returning method
- verify a recursive method, according to the Three-Question Method
- decide whether a recursive solution is appropriate for a problem
- compare and contrast dynamic storage allocation and static storage allocation in relation to using recursion
- explain how recursion works internally by showing the contents of the run-time stack
- replace a recursive solution with iteration and/or the use of a stack
- explain why recursion may or may not be a good choice to implement the solution of a problem

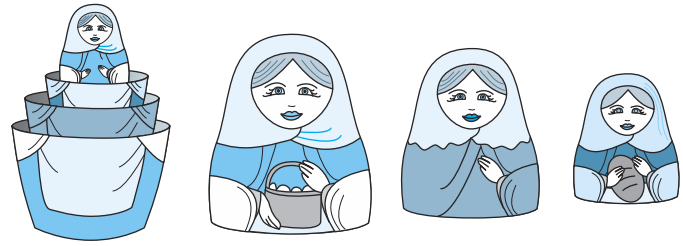
This chapter introduces the topic of recursion—a unique problem-solving approach supported by many computer languages (Java included). With recursion, you solve a problem by repeatedly breaking it into smaller versions of the same problem, until the problem is reduced to a trivial size that can be easily solved; then you repeatedly combine your solutions to the subproblems until you arrive at a solution to the original problem.

Although recursion can at first appear unwieldy and awkward, when applied properly it is an extremely powerful and useful problem-solving tool.

7.1 What is Recursion?

You may have seen a set of gaily-painted Russian dolls that fit inside one another. Inside the first doll is a smaller doll, inside of which is an even smaller doll, inside of which is yet a smaller doll, and so on. A recursive algorithm is like such a set of Russian dolls. It reproduces itself with smaller and smaller versions of itself until a version is reached that can no longer be subdivided—that is, until the smallest doll is reached. The recursive algorithm is implemented by using a method that makes recursive calls to itself; analogous to taking the dolls apart one by one. The solution often depends on passing back from the recursive calls, larger and larger subsolutions, analogous to putting the dolls back together again.

Recursive call A method call in which the method being called is the same as the one making the call



In Java, any method can invoke another method. A method can even invoke itself! When a method invokes itself, it is making a recursive call. The word *recursive* means "having the characteristic of coming up again, or repeating." In this case, a method invocation is being repeated by the method itself. This type of recursion is sometimes called direct recursion, since the method directly calls itself. All of the examples in this chapter are of direct recursion. Indirect recursion occurs when method A calls method B, and method B calls method A; the chain of method calls could be even longer, but if it eventually leads back to method A, then it is indirect recursion.

Direct recursion Recursion in which a method directly calls itself

Indirect recursion Recursion in which a chain of two or more method calls returns to the method that originated the chain

Recursion is a powerful programming technique. However, you must be careful when using recursion. Recursive solutions can be less efficient than iterative solutions to the same problem. In fact, some of the examples used in this chapter are better suited to iterative methods (see the discussion in Section 7.10). Still, many problems lend themselves to simple, elegant, recursive solutions and are exceedingly cumbersome to solve iteratively. Some programming languages, such as early versions of FORTRAN, BASIC, and COBOL, do not allow recursion. Other languages are especially oriented to recursive approaches—LISP is one of these. Java lets us take our choice; we can implement both iterative and recursive algorithms in Java.

A Classic Example of Recursion

Mathematicians often define concepts in terms of the process used to generate them. For instance, $n!$ (read "n factorial") is used to calculate the number of permutations of n elements. One mathematical description of $n!$ is

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n * (n-1) * (n-2) * \cdots * 1, & \text{if } n > 0 \end{cases}$$

Consider the case of 4!. Because $n > 0$, we use the second part of the definition:

$$4! = 4 * 3 * 2 * 1 = 24$$

This description of $n!$ provides a different definition for each value of n, for the three dots stand in for the intermediate factors. That is, the definition of 2! is 2 * 1, the definition of 3! is 3 * 2 * 1, and so forth.

We can also express $n!$ without using the three dots:

$$n! = \begin{cases} 1, & \text{if } n = 0 \\ n * (n-1)! & \text{if } n > 0 \end{cases}$$

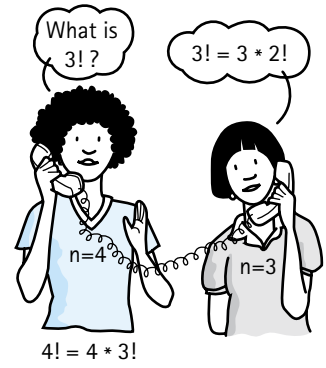
This is a recursive definition because we express the factorial function in terms of itself.

Recursive definition A definition in which something is defined in terms of smaller versions of itself

Let's consider the recursive calculation of 4! intuitively. Because 4 is not equal to 0, we use the second half of the definition:

$$4! = 4 * (4 - 1)! = 4 * 3!$$

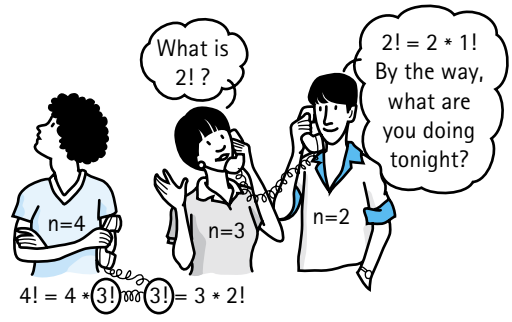
Of course, we can't do the multiplication yet, because we don't know the value of 3!. So we call up our good friend Sue Ann, who has a Ph.D. in math, to find the value of 3!.



Sue Ann has the same formula we have for calculating the factorial function, so she knows that

$$3! = 3 * (3 - 1)! = 3 * 2!$$

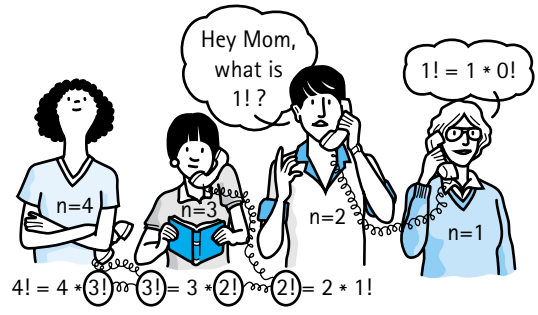
She doesn't know the value of 2!, however, so she puts you on hold and calls up her friend Max, who has an M.S. in math.



Max has the same formula Sue Ann has, so he quickly calculates that

$$2! = 2 * (2 - 1)! = 2 * 1!$$

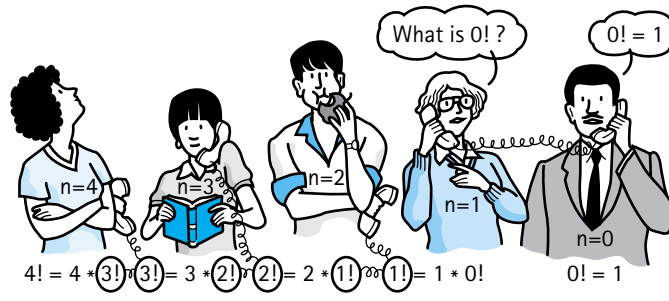
But Max can't complete the multiplication because he doesn't know the value of 1!. He puts Sue Ann on hold and calls up his mother, who has a B.A. in math education.



Max's mother has the same formula Max has, so she quickly figures out that

$$1! = 1 * (1 - 1)! = 1 * 0!$$

Of course, she can't perform the multiplication, because she doesn't have the value of 0!. So Mom puts Max on hold and calls up her colleague Bernie, who has a B.A. in English literature.



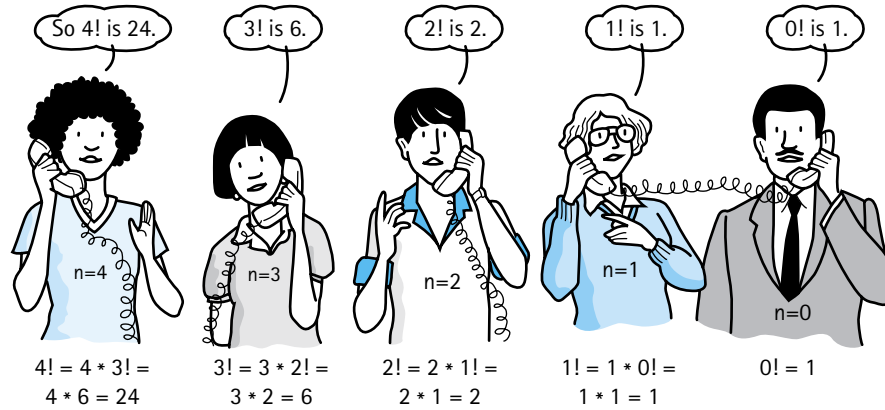
Bernie doesn't need to know any math to figure out that 0! = 1 because he can read that information in the first clause of the formula ($n! = 1$, if $n = 0$). He reports the answer immediately to Max's mother. She can now complete her calculations:

$$1! = 1 * 0! = 1 * 1 = 1$$

She reports back to Max that 1! = 1, who now performs the multiplication in his formula and learns that

$$2! = 2 * 1! = 2 * 1 = 2$$

He reports back to Sue Ann, who can now finish her calculation:



$$3! = 3 * 2! = 3 * 2 = 6$$

Sue Ann calls you with this exciting bit of information. You can now complete your calculation:

$$4! = 4 * 3! = 4 * 6 = 24$$

Notice when the recursive calls stop. They stop when we have reached a case for which we know the answer without resorting to a recursive definition. In this example, Bernie knew that 0! = 1 directly from the definition without having to resort to recursion. The case (or cases) for which an answer is explicitly known is called the base case. The case for which the solution is expressed in terms of a smaller version of itself is called the recursive or general case. A recursive algorithm is an algorithm that expresses the solution in terms of a call to itself, a recursive call. A recursive algorithm must terminate; that is, it must have a base case.

Base case The case for which the solution can be stated nonrecursively

General (recursive) case The case for which the solution is expressed in terms of a smaller version of itself

Recursive algorithm A solution that is expressed in terms of (a) smaller instances of itself and (b) a base case

7.2 Programming Recursively

Of course, the use of recursion is not limited to mathematicians with telephones. Computer languages, such as Java, that support recursion, give the programmer a powerful tool for solving certain kinds of problems by reducing the complexity or by hiding the details of the problem.

In this chapter we consider recursive solutions to several problems. In our initial discussion, you may wonder why a recursive solution would ever be preferred to an iterative, or nonrecursive one, for the iterative solution may seem simpler and be more efficient. Don't worry. There are, as you see later, situations in which the use of recursion produces a much simpler—and more elegant—program.

Coding the Factorial Function

A recursive method is one that calls itself. Here we construct a recursive Java method `factorial` that returns the value of `number`! when passed the argument `number`. Therefore, the method needs to return the value `number * (number - 1)`!. Where can we get the value `(number - 1)`! that we need in the formula? We already have a method for doing this calculation: `factorial`. So we just call `factorial` from within `factorial`. Notice the recursive call to `factorial` in the *else* statement in the code below (throughout this chapter we emphasize the recursive calls within our code). Of course, the argument, `number - 1`, in the recursive call is different from the argument in the original call, `number`. This is an important and necessary consideration; otherwise the method would continue calling itself indefinitely.

```
public static int factorial(int number)
{
  if (number == 0)
    return 1;           // Base case
  else
    return (number * factorial(number - 1));      // General case
}
```

Let's trace this method with an original `number` of 4:

Call 1	`number` is 4. Because `number` is not 0, the *else* branch is taken. The *return* statement cannot be completed until the recursive call to `factorial` with `number - 1` as the argument has been completed.
Call 2	`number` is 3. Because `number` is not 0, the *else* branch is taken. The *return* statement cannot be completed until the recursive call to `factorial` with `number - 1` as the argument has been completed.
Call 3	`number` is 2. Because `number` is not 0, the *else* branch is taken. The *return* statement cannot be completed until the recursive call to `factorial` with `number - 1` as the argument has been completed.
Call 4	`number` is 1. Because `number` is not 0, the *else* branch is taken. The *return* statement cannot be completed until the recursive call to `factorial` with `number - 1` as the argument has been completed.
Call 5	`number` is 0. Because `number` equals 0, this call to the method returns, sending back 1 as the result.
Call 4	The *return* statement in this copy can now be completed. The value to be returned is `number` (which is 1) times 1. This call to the method returns, sending back 1 as the result.
Call 3	The *return* statement in this copy can now be completed. The value to be returned is `number` (which is 2) times 1. This call to the method returns, sending back 2 as the result.
Call 2	The *return* statement in this copy can now be completed. The value to be returned is `number` (which is 3) times 2. This call to the method returns, sending back 6 as the result.
Call 1	The *return* statement in this copy can now be completed. The value to be returned is `number` (which is 4) times 6. This call to the method returns, sending back 24 as the result. Because this is the last of the calls to `factorial`, the recursive process is over. The value 24 is returned as the final value of the call to `factorial` with an argument of 4.

Figure 7.1 summarizes the execution of the `factorial` method with an argument of 4.

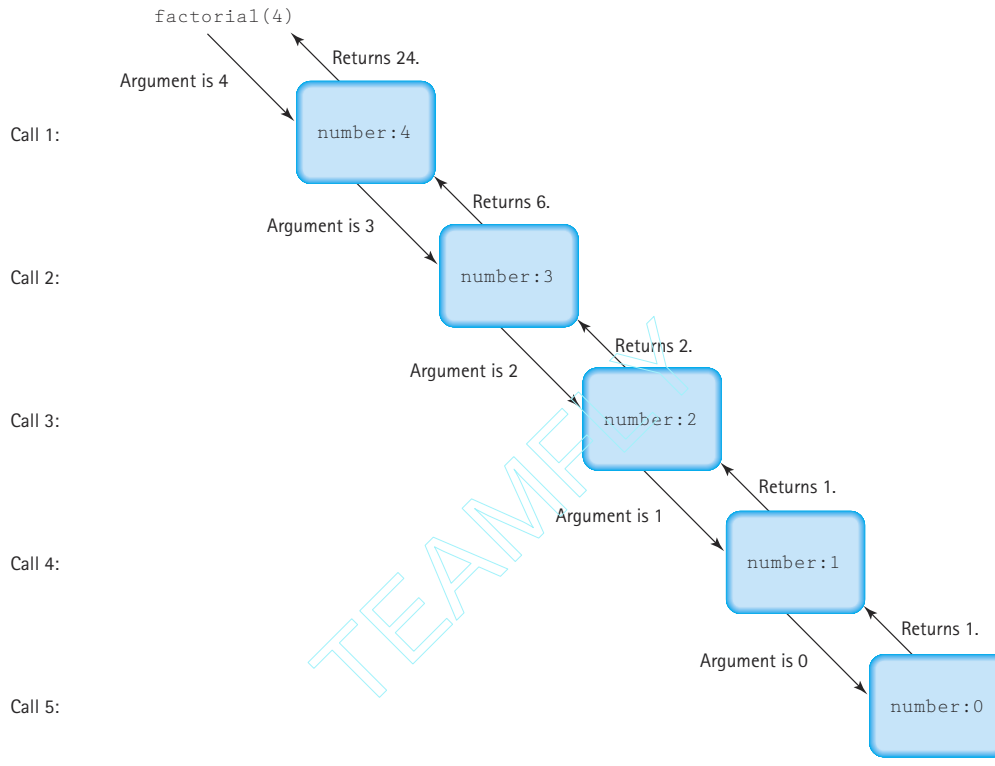


Figure 7.1 *Execution of* `factorial(4)`

Comparison to the Iterative Solution

We have used the factorial algorithm to demonstrate recursion because it is familiar and easy to visualize. In practice, one would never want to solve this problem using recursion. The iterative solution is simpler and much more efficient because starting a new iteration of a loop is a faster operation than calling a method. Let's look at an iterative solution to the problem:

```
// Iterative solution
public static int factorial(int number)
{
  int value = 1;
  for (int count = 2; count <= number; count++)
  {
    value = value * count;
  }
  return value;
}
```

For easy comparison, we repeat the recursive solution:

```
// Recursive solution
public static int factorial(int number)
{
  if (number == 0)
    return 1;           // Base case
  else
    return (number * factorial (number - 1));      // General case
}
```

Iterative solutions tend to employ loops, whereas recursive solutions tend to have selection statements—either an *if* or a *switch* statement. A branching structure is the main control structure in a recursive method. A looping structure is the main control structure in an iterative method. The iterative version of `factorial` has two local variables (`value` and `count`), whereas the recursive version has none. There are usually fewer local variables in a recursive method than in an iterative method. Sometimes, as we see later, a recursive solution needs more parameters than the equivalent iterative one. Data values used in the iterative solution are usually initialized inside the method, above the loop. Similar data values used in a recursive solution are usually initialized by the choice of parameter values in the initial call to the method.

Note that the code for both approaches is included in the `tryFact.java` application provided on our web site.

7.3 Verifying Recursive Methods

The kind of walk-through we did in the previous section, to check the validity of a recursive method, is time consuming, tedious, and often confusing. Furthermore, simulating the execution of `factorial(4)` tells us that the method works when the argument equals 4, but it doesn't tell us whether the method is valid for all nonnegative values of the argument. It would be useful to have a technique that would help us determine inductively whether a recursive algorithm works.

The Three-Question Method

We use the Three-Question Method of verifying recursive methods. To verify that a recursive solution works, you must be able to answer yes to all three of these questions.

1. *The Base-Case Question:* Is there a nonrecursive way out of the method, and does the method work correctly for this base case?
2. *The Smaller-Caller Question:* Does each recursive call to the method involve a smaller case of the original problem, leading inescapably to the base case?
3. *The General-Case Question:* If the recursive call(s) works correctly, does the whole method work correctly?

Let's apply these three questions to method `factorial`. (We use the mathematical symbol *n* rather than the variable `number` in this discussion.)

1. *The Base-Case Question:* The base case occurs when $n = 0$. The `factorial` method then returns the value of 1, which is the correct value of 0!, and no further (recursive) calls to `factorial` are made. The answer is yes.
2. *The Smaller-Caller Question:* To answer this question we must look at the parameters passed in the recursive call. In method `factorial`, the recursive call passes $n - 1$. Each subsequent recursive call sends a decremented value of the parameter, until the value sent is finally 0. At this point, as we verified with the base-case question, we have reached the smallest case, and no further recursive calls are made. The answer is yes.
3. *The General-Case Question:* In the case of a method like `factorial`, we need to verify that the formula we are using actually results in the correct solution. Assuming that the recursive call `factorial(n - 1)` gives us the correct value of $(n - 1)!$, the *return* statement computes $n * (n - 1)!$. This is the definition of a factorial, so we know that the method works for all positive integers. In answering the first question, we have already ascertained that the method works for $n = 0$. (The factorial function is defined only for nonnegative integers.) So the answer is yes.

If you are familiar with inductive proofs, you should recognize what we have done. Having made the assumption that the method works for $(n - 1)$, we have shown that applying the method to the next value, $(n - 1) + 1$, or n, results in the correct formula for calculating $n!$. Since we have also shown that the formula works for the base case, $n = 0$, we have inductively shown that it works for any integral argument >= 0.

7.4 Writing Recursive Methods

The questions used for verifying recursive methods can also be used as a guide for writing recursive methods. You can use the following approach to write any recursive method:

1. Get an exact definition of the problem to be solved. (This, of course, is the first step in solving any programming problem.)
2. Determine the size of the problem to be solved on this call to the method. On the initial call to the method, the size of the whole problem is expressed in the value(s) of the parameter(s).
3. Identify and solve the base case(s) in which the problem can be expressed nonrecursively. This ensures a yes answer to the base-case question.
4. Identify and solve the general case(s) correctly in terms of a smaller case of the same problem—a recursive call. This ensures yes answers to the smaller-caller and general-case questions.

In the case of `factorial`, the definition of the problem is summarized in the definition of the factorial function. The size of the problem is the number of values to be multi-

plied: *n*. The base case occurs when *n* = 0, in which case we take the nonrecursive path. Finally, the general case occurs when *n* > 0, resulting in a recursive call to `factorial` for a smaller case: `factorial(n - 1)`.

A Recursive Version of isThere

Let's apply this approach to writing a recursive version of the `isThere` method for our abstract `List` class. Recall the specification of the `isThere` method:

```
public boolean isThere (Listable item);
// Effect:        Determines if element matching item is on this list
// Postcondition: Return value = (element with the same key as item
//                is on this list)
```

You need to also remember that the underlying structure used to implement our Unsorted List ADT by our `List` class is the array:

```
protected Listable[] list;      // Array to hold this list's elements
protected int numItems;         // Number of elements on this list
```

The `isThere` problem can be decomposed into smaller problems by deciding if `item` is in the first position of the list or in the rest of the list. More formally

```
isThere(item) = return  (is item in the first list position?)
                        or (is item in the rest of the list?)
```

We can answer the first question just by comparing `item` to `list[0]`. But how do we know whether `item` is in the rest of the list? If only we had a method that would search the rest of the list. But we do have one! The `isThere` method searches for a value in a list. We simply need to start searching at position 1, instead of at position 0 (a smaller case). To do this, we need to pass the search starting place to `isThere` as a parameter. Each recursive call to `isThere` passes a starting location that is one location more than its own starting location. But how do we know when to stop? Within the code for the recursive `isThere` method we can use `numItems - 1` to mark the end of the list, so we can stop the recursive calls when our starting position is that value.

We use the following method specification:

```
public boolean isThere (Listable item, int startPosition);
// Effect:        Determines if element matching item is on this list between
//                startPosition and the end of the list
// Postcondition: Return value = (element with the same key as item is on this
//                list between startPosition and the end of the list)
```

To search the whole list, we would invoke the method with the statement

```
if (isThere(value, 0))
   .
   .
   .
```

Note that since Java allows us to overload method names (as long as the methods have unique signatures), we can still call our new method `isThere`. The general case of this approach is the part that searches the rest of the list. This case involves a recursive call to `isThere`, specifying a smaller part of the array to be searched:

```
return isThere(item, startPosition + 1);
```

By using the expression `startPosition + 1` as the argument, we have effectively diminished the size of the problem to be solved by the recursive call. That is, searching the list from `startPosition + 1` to `numItems - 1` is a smaller task than searching from `startPosition` to `numItems - 1`. Figure 7.2 shows the recursive method `isThere` frozen in mid-execution.

Finally, we need to know when to stop searching. In this problem, we have two base cases: (1) when the value is found (return true), and (2) when we have reached the end of the list without finding the value (return false). In either case, we can stop making recursive calls to `isThere`.

The code for the base case returns the appropriate Boolean value to the copy of the method that invoked it. This method, in turn, immediately returns the value to the copy of the method that invoked it, and so on, until the Boolean value is returned to whatever originally invoked the `isThere` method. Let's summarize what we have discussed and then write the `isThere` method.

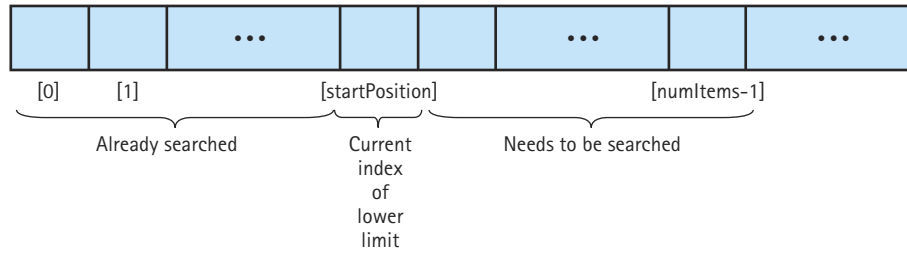


Figure 7.2 *Recursive method `isThere` in mid-execution*

Recursive isThere(item) method: returns boolean

Definition:	Searches list for item. Returns true if item is found; returns false otherwise.
Size:	The number of slots to search in list.info[startPosition]..list.info[list.numItems − 1].
Base Cases:	(1) If list.info[startPosition] equals value, return true. (2) If startPosition equals list.numItems − 1 and list.info[list.length − 1] does not equal item, return false.
General Case:	Search the rest of the list for item. This is a recursive invocation of isThere with a parameter startPosition + 1 (smaller caller).

The code for the recursive `isThere` method follows

```
public boolean isThere (Listable item, int startPosition)
// Returns true if item is on this list; otherwise, returns false
{
  if (item.compareTo(list[startPosition]) == 0)  // If they match
    return true;
  else if (startPosition == (numItems - 1))      // If end of list
    return false;
  else return isThere(item, startPosition + 1);
}
```

Note that it is the parameter `startPosition` that acts as an index through the array; it is initialized in the original invocation of `isThere` and incremented on each recursive call. The equivalent iterative solution would use a local counter, initialized inside the method above the loop and incremented inside the loop. The code for this method is included in the `UnsortedStringList3` class in the `ch07.stringLists` package on our web site.

Let's use the Three-Question Method to verify this method.

1. *The Base Case Question:* One base case occurs when the value is found on this call and the method is exited without any further calls to itself. A second base case occurs when the end of the list is reached without the value being found and the method is exited without any further recursive calls. The answer is yes.
2. *The Smaller Caller Question:* The recursive call in the general case increments the value of `startPosition`, making the part of the list left to be searched smaller. The answer is yes.

3. *The General Case Question:* Let's assume that the recursive call in the general case correctly tells us whether the value is found in the second through last elements in the list. Then Base Case 1 gives us the correct answer of `true` if the value is found in the first element in the list, and Base Case 2 gives us the correct answer of `false` if the value is not in the first element and the first element is the only element in the list. The only other possible case is that the value exists somewhere in the rest of the list. Assuming that the recursive call works correctly, the whole method works, so the answer to this question is also yes.

Debugging Recursive Methods

Because of their nested calls to themselves, recursive methods can be confusing to debug. The most serious problem is the possibility that the method recurses forever. A typical symptom of this problem is an error message telling us that the system has run out of space in the run-time stack, due to the level of recursive calls. (In Section 7.8 we look at how recursion uses the run-time stack.) Using the Three-Question Method to verify recursive methods should help us avoid the problem of never finishing. If we can answer yes to the base-case and smaller-caller questions, we should be able to guarantee that the method eventually ends—theoretically, at least.

That does not guarantee, however, that the program does not fail due to lack of space. In Section 7.8 we discuss the amount of space overhead required to support recursive method calls. And since a call to a recursive method may generate many, many levels of method calls to itself—it might be more than the system can handle.

One error that programmers often make when they first start writing recursive methods is to use a looping structure instead of a branching one. Because they tend to think of the problem in terms of a repetitive action, they inadvertently use a *while* statement rather than an *if* statement. The main body of the recursive method should always be a breakdown into base and recursive cases. Hence, we use a branching statement, not a looping statement. It's a good idea to double-check your recursive methods to make sure that you used an *if* or *switch* statement to get a branching effect.

Recursive methods are good places to put debug output statements during testing. Print out the parameters and local variables, if any, at the beginning and end of the method. Be sure to print the values of the parameters on the recursive call(s) to make sure that each call is trying to solve a problem smaller than the previous one.

7.5 Using Recursion to Simplify Solutions—Two Examples

So far, the examples we have looked at could just as easily (or more easily) been written as iterative methods. There are, however, many problems in which using recursion simplifies the solution. At the end of the chapter, we talk more about choosing between iterative and recursive solutions.

Combinations

Our first example of a problem that is easily solved with recursion is the evaluation of another mathematical function, Combinations, which tells us how many combinations of a certain size can be made out of a total group of elements. For instance, if we have 20 different books to pass out to four students, we can easily see that—to be equitable—we should give each student five books. But, how many combinations of five books can be made out of a group of 20 books?

There is a recursive mathematical formula that can be used for solving this problem. Given that *C* is the total number of combinations, *group* is the total size of the group to pick from, *members* is the size of each subgroup, and *group* >= *members*,

$$C(\text{group}, \text{members}) = \begin{cases} \text{group}, & \text{if members} = 1 \\ 1, & \text{if members} = \text{group} \\ C(\text{group} - 1, \text{members} - 1) + C(\text{group} - 1, \text{members}), & \text{if group} > \text{members} > 1 \end{cases}$$

Because this definition of *C* is recursive, it is easy to see how a recursive method can be used to solve the problem.

Let's summarize our problem.

Number of Combinations, returns int

Definition:	calculates how many combinations of members size can be made from the total group size.
Size:	Sizes of group, members.
Base Case:	(1) If members = 1, return group.
	(2) If members = group, return 1.
General Case:	If group > members > 1, return
	Combinations(group − 1, members − 1) +
	Combinations(group − 1, members)

The resulting recursive method, `combinations`, is listed here.

```
public static int combinations(int group, int members)
// Pre:  group and members are positive
// Post: Return value = number of combinations of members size
//       that can be constructed from the total group size
{
  if (members == 1)
    return group;              // Base case 1
```

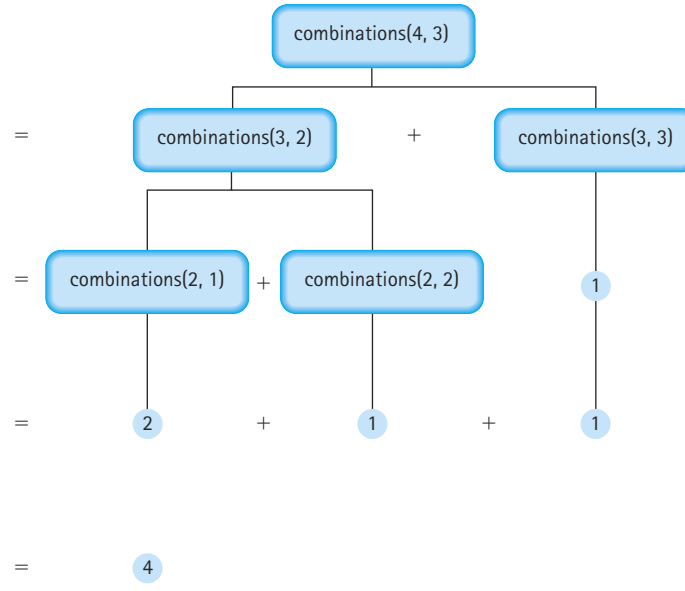


Figure 7.3 *Calculating* `combinations(4,3)`

```
  else if (members == group)
    return 1;                  // Base case 2
  else
    return (combinations(group - 1, members - 1) +
            combinations(group - 1, members));
}
```

The processing of this method to calculate the number of combinations of three elements that can be made from a set of four is shown in Figure 7.3.

Returning to our original problem, we can now find out how many combinations of five books can be made from the original set of 20 books with the statement

```
System.out.println("Number of combinations = " + combinations(20, 5));
```

which outputs "Number of combinations = 15504". Did you guess that it would be that large a number? Recursive definitions can be used to define functions that grow quickly.

The `tryComb.java` application program, available on our web site, can be used to test the recursive `combinations` method.

Beware of Inappropriate Recursion!

The Number of Combinations problem is a good example of a problem for which we can quickly create a recursive solution. Attempting to create an iterative solution based on the recursive definition of *C(group, members)* would be difficult. However, as discussed in Section 7.10, Deciding Whether to Use a Recursive Solution, our recursive approach to this problem results in inefficient multiple calculations of subsolutions; so much so that this approach can only be used for small argument values.

However, an easy (and efficient) iterative solution does exist to this problem, since mathematicians provide us an alternate definition of the function *C*:

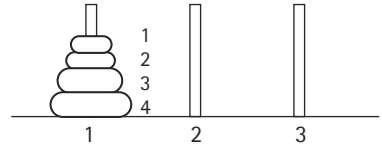
*C(group, members) = group! / (members!) * (group – members)!*

A carefully constructed iterative program based on this formula is much more efficient than the recursive solution we presented in this section.

Towers of Hanoi

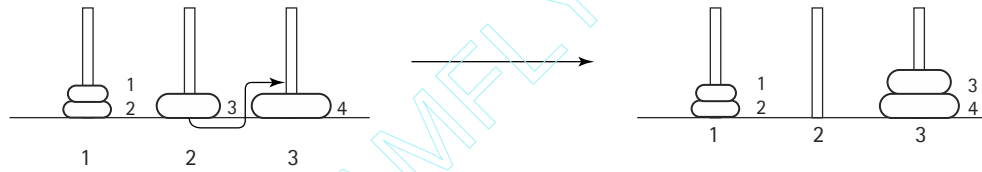
One of your first toys may have been a disk with three pegs with colored circles of different diameters. If so, you probably spent countless hours moving the circles from one peg to another. If we put some constraints on how the circles or discs can be moved, we have an adult game called the Towers of Hanoi. When the game begins, all the circles are on the first peg in order by size, with the smallest on the top. The object of the game is to move the circles, one at a time, to the third peg. The catch is that a circle cannot be placed on top of one that is smaller in diameter. The middle peg can be used as an auxiliary peg, but it must be empty at the beginning and at the end of the game. The circles can only be moved one at a time.

To get a feel for how this might be done, let's look at some sketches of what the configuration must be at certain points if a solution is possible. We use four circles or discs. The beginning configuration is:



To move the largest circle (circle 4) to peg 3, we must move the three smaller circles to peg 2 (this cannot be done with 1 move). Then circle 4 can be moved into its final place:

Let's assume we can do this. Now, to move the next largest circle (circle 3) into place, we must move the two circles on top of it onto an auxiliary peg (peg 1 in this case):



To get circle 2 into place, we must move circle 1 to another peg, freeing circle 2 to be moved to its place on peg 3:

The last circle (circle 1) can now be moved into its final place, and we are finished:

The general algorithm for moving *n* circles from peg 1 to peg 3 is:

Get n Circles Moved from Peg 1 to Peg 3

Get n − 1 circles moved from peg 1 to peg 2
Move the nth circle from peg 1 to peg 3
Get n − 1 circles moved from peg 2 to peg 3

This algorithm certainly seems simple; surely there must be more. But this really is all there is to it.

Let's write a recursive method that implements this algorithm. We can see that recursion works well because the first and third steps of the algorithm are essentially a repeat of the overall algorithm, albeit with a smaller number of disks. Notice however, that the beginning peg, the ending peg, and the auxiliary peg are different for the subproblems; they keep changing during the recursive execution of the algorithm. To make the algorithm easier to follow, we call the pegs `beginPeg`, `endPeg`, and `auxPeg`. These three pegs, along with the number of circles on the beginning peg, are the parameters of the method. We can't actually move discs, of course, but we can print out a message to do so.

We have the recursive or general case, but what about a base case? How do we know when to stop the recursive process? The clue is in the expression "Get *n* circles moved." If we don't have any circles to move, we don't have anything to do. We are finished with that stage. Therefore, when the number of circles equals 0, we do nothing (that is, we simply return). That is the base case.

```
public static void doTowers(
   int circleCount,     // Number of circles to move
   int beginPeg,        // Peg containing circles to move
   int auxPeg,          // Peg holding circles temporarily
   int endPeg       )   // Peg receiving circles being moved
// Moves are written on file outFile
{
  if (circleCount > 0)
  {
    // Move n - 1 circles from beginning peg to auxiliary peg
    doTowers(circleCount - 1, beginPeg, endPeg, auxPeg);

    outFile.println("Move circle from peg " + beginPeg
             + " to peg " + endPeg);

    // Move n - 1 circles from auxiliary peg to ending peg
    doTowers(circleCount - 1, auxPeg, beginPeg, endPeg);
  }
}
```

It's hard to believe that such a simple algorithm actually works, but we'll prove it to you. We enclose the method within a driver class `Towers` that invokes the `doTowers` method. Output statements have been added so you can see the values of the arguments with each recursive call. Because there are two recursive calls within the method, we have indicated which recursive statement issued the call. The program accepts two arguments through command line parameters. The first is the number of circles and the second is the name of an output file. For example, to generate the results for six circles to the file `HanoiInfo.dat`, enter

```
java Towers 6 HanoiInfo
```

on the command line.

```
//--------------------------------------------------------------------------
// Towers.java             by Dale/Joyce/Weems                    Chapter 7
//
// Driver class for doTowers method
// The number of circles is the first command-line parameter
// The output file name is the second command-line parameter
//--------------------------------------------------------------------------

import java.io.*;

public class Towers
{
  private static PrintWriter outFile;    // Output data file

  public static void main(String[] args) throws IOException
  {
    // Prepare outputfile
    String outFileName = args[1];
    outFile = new PrintWriter(new FileWriter(outFileName));

    // Get number of circles on starting peg
    int circleCount;
    circleCount = Integer.valueOf(args[0]).intValue();

    outFile.println("OUTPUT WITH " + circleCount + " CIRCLES");
    outFile.println("From original: ");
    doTowers(circleCount, 1, 2, 3);
    outFile.close();
  }

  public static void doTowers(
         int circleCount,    // Number of circles to move
         int beginPeg,       // Peg containing circles to move
         int auxPeg,         // Peg holding circles temporarily
         int endPeg      )   // Peg receiving circles being moved
  //  Moves are written on file outFile
  //  This recursive method moves circleCount circles from beginPeg
  //   to endPeg.  All but one of the circles are moved from beginPeg
  //   to auxPeg, then the last circle is moved from beginPeg to
  //   endPeg, and then the circles are moved from auxPeg to endPeg
  //  The subgoals of moving circles to and from auxPeg are what
  //   involve recursion
```

```
  {
    outFile.println("#circles: " + circleCount + " Begin: " +
      beginPeg + " Auxil: " + auxPeg + " End: " + endPeg);
    if (circleCount > 0)
    {
      // Move n - 1 circles from beginning peg to auxiliary peg
      outFile.println("From first:   ");
      doTowers(circleCount - 1, beginPeg, endPeg, auxPeg);

      outFile.println("Move circle from peg " + beginPeg
              + " to peg " + endPeg);

      // Move n - 1 circles from auxiliary peg to ending peg
      outFile.println("From Second:  ");
      doTowers(circleCount - 1, auxPeg, beginPeg, endPeg);
    }
  }
}
```

The output from a run with three circles follows. "`Original`" means that the parameters listed beside it are from the nonrecursive call, which is the first call to `doTowers`. "`From first`" means that the parameters listed are for a call issued from the first recursive statement. "`From second`" means that the parameters listed are for a call issued from the second recursive statement. Notice that a call cannot be issued from the second recursive statement until the preceding call from the first recursive statement has completed execution.

```
OUTPUT WITH 3 CIRCLES
From original: #circles: 3 Begin: 1 Auxil: 2 End: 3
From  first:   #circles: 2 Begin: 1 Auxil: 3 End: 2
From  first:   #circles: 1 Begin: 1 Auxil: 2 End: 3
From  first:   #circles: 1 Begin: 1 Auxil: 3 End: 2
Move circle 1 from 1 to 3
From second:   #circles: 0 Begin: 2 Auxil: 1 End: 3
Move circle 2 from 1 to 2
From second:   #circles: 1 Begin: 3 Auxil: 1 End: 2
From  first:   #circles: 0 Begin: 3 Auxil: 2 End: 1
Move circle 1 from 3 to 2
From second:   #circles: 0 Begin: 1 Auxil: 3 End: 2
Move circle 3 from 1 to 3
From second:   #circles: 2 Begin: 2 Auxil: 1 End: 3
From  first:   #circles: 1 Begin: 2 Auxil: 3 End: 1
From  first:   #circles: 0 Begin: 2 Auxil: 1 End: 3
Move circle 1 from 2 to 1
```

```
From second:   #circles: 0 Begin: 3 Auxil: 2 End: 1
Move circle 2 from 2 to 3
From second:   #circles: 1 Begin: 1 Auxil: 2 End: 3
From  first:   #circles: 0 Begin: 1 Auxil: 3 End: 2
Move circle 1 from 1 to 3
From second:   #circles: 0 Begin: 2 Auxil: 1 End: 3
```

Here is the output with all of the trace statements removed, which makes it easier to follow the solution:

```
OUTPUT WITH 3 CIRCLES
Move circle 1 from 1 to 3
Move circle 2 from 1 to 2
Move circle 1 from 3 to 2
Move circle 3 from 1 to 3
Move circle 1 from 2 to 1
Move circle 2 from 2 to 3
Move circle 1 from 1 to 3
```

Try the program for yourself. Be careful though—remember what we stated earlier about recursive functions growing quickly. In fact, every time you add one more circle to the starting peg you more than double the amount of output from the program. A run of `Towers` on the author's system, with an input parameter indicating 16 circles, generated an output file of size 10 megabytes.

7.6 A Recursive Version of Binary Search

In the chapter on array-based lists (Chapter 3), we developed the binary search algorithm for the public method `isThere` of our Sorted List ADT. Here is the description of the algorithm. "Divide the list in half (divide by 2—that's why it's called a binary search) and see if the middle element is the target item; if not, decide which half to look in next. Division of the selected portion of the list is repeated until the item is found or it is determined that the item is not in the list." There is something inherently recursive about this description.

Though the method that we wrote in Chapter 3 was iterative, this really is a recursive algorithm. The solution is expressed in smaller versions of the original problem: If the answer isn't found in the middle position, perform a binary search (a recursive call) to search the appropriate half of the list (a smaller problem). In the iterative version we kept track of the bounds of the current search area with two local variables, `first` and `last`. In the recursive version we call the method with these two values as parameters. Let's summarize the problem in terms of a `boolean` method `binarySearch` that returns `true` or `false` according to whether the desired item is found in the range indicated by the two parameters, `fromLocation` and `toLocation`.

binarySearch (item, fromLocation, toLocation), returns boolean

Definition:	Searches the list delimited by the parameters to see if item is present.
Size:	The number of elements in list[fromLocation] … list[toLocation].
Base Cases:	(1) If fromLocation > toLocation, return false. (2) If item.compareTo(list[midPoint]) = 0, return true.
General Case:	If item.compareTo(list[midPoint]) < 0, binarySearch the first half of the list. If item.compareTo(list[midPoint]) > 0, binarySearch the second half of the list.

The recursive `binarySearch` method follows. The method should not be a public method of class `SortedList`, but a private, auxiliary method of the class, since it takes array bounds information as arguments. Remember, following the principles of information hiding, the fact that an array is used to implement the list should not be used explicitly in the interface of the class. Therefore, `binarySearch` cannot be public. Besides, we really do not want to change our public list interface.

```
private boolean binarySearch (Listable item, int fromLocation, int toLocation)
// Returns true if item is on this list, between fromLocation and toLocation;
//  otherwise, returns false
{
  if (fromLocation > toLocation)          // Base case 1
    return false;
  else
  {
    int midPoint;
    int compareResult;
    midPoint = (fromLocation + toLocation) / 2;
    compareResult = item.compareTo(list[midPoint]);

    if (compareResult == 0)           // Item found
      return true;
    else if (compareResult < 0)
      // Item is less than element at location
      return binarySearch (item, fromLocation, midPoint - 1);
    else
      // Item is greater than element at location
      return binarySearch (item, midPoint + 1, toLocation);
  }
}
```

The client program's call to the `isThere` method has the same form as before; for example:

```
if myList.isThere(myItem) ...
```

The `isThere` method now consists of a single call to the recursive method, passing it the original values for `fromLocation` and `toLocation`:

```
public boolean isThere (Listable item)
// Returns true if item is on this list; otherwise, returns false
{
  return binarySearch(item, 0, numItems - 1);
}
```

The `SortedList2` class in the `ch07.genericLists` package on our web site uses the recursive `isThere` method.

7.7 Recursive Linked-List Processing

Next we look at some problems that use recursive approaches for manipulating a linked list. In the case of a singly linked list, a recursive approach can sometimes be useful for "backing up" in the list.

Reverse Printing

We start with a method that prints out the elements in our `SortedLinkedList` class. You might be protesting that this task is so simple to accomplish iteratively that it does not make any sense to write it recursively. So let's make the task more difficult: Print out the elements of the list in reverse order. This problem is much more easily and elegantly solved recursively than it is iteratively.

What is the task to be performed? The algorithm follows and is illustrated in Figure 7.4. To simplify our linked list figures in this chapter, we use a capital letter to stand for the object referenced by `info`, an arrow to represent the `next` reference (as always), and a slash to represent `null`.

revPrint (listRef)

Print out the second through last elements in the list referenced by listRef in reverse order.
Then print the first element in the list referenced by listRef

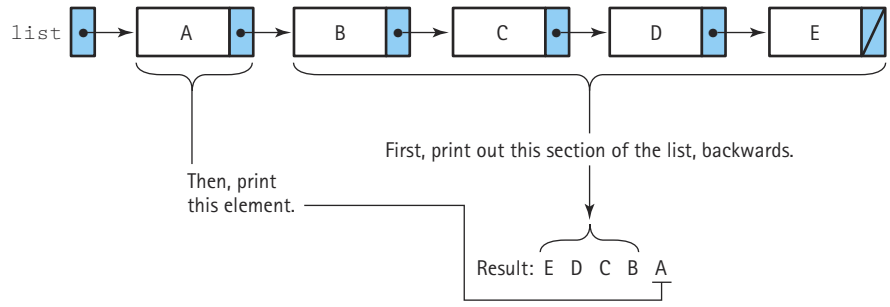


Figure 7.4 *Recursive* `revPrint`

The second part of the task is simple. If `listRef` references the first node in the list, we can print out its contents with the statement

```
System.out.println(" " + listRef.info);
```

The first part of the task—printing out all the other nodes in the list in reverse order—is also simple because we have a method that prints out lists in reverse order: We just invoke the method `revPrint` recursively. Of course, we have to adjust the parameter somewhat, so that the invocation is `revPrint(listRef.next)`. This call says "Print, in reverse order, the linked list pointed to by `listRef.next`." This task in turn is accomplished recursively in two steps:

revPrint the rest of the list (third through last elements).
Then print the second element in the list.

And, of course, the first part of this task is accomplished recursively.

Where does it all end? We need a base case. We can stop calling `revPrint` when we have completed its smallest case: reverse printing a list of one element. Then the value of `listRef.next` is `null`, and we can stop making recursive calls. Let's summarize the problem.

Reverse Print (listRef)

Definition:	Prints out the list referenced by listRef in reverse order.
Size:	Number of elements in the list referenced by listRef.
Base Case:	If the list is empty, do nothing.
General Case:	Reverse Print the list referenced by listRef.next, then print listRef.info.

The other recursive methods that we have written have been value-returning methods; `revPrint` is a `void` method. The `revPrint` method is a `void` method because each method call simply performs an action (printing the contents of a list node) without returning a value to the calling code. Here is the code for `revPrint` (yes, it is this simple!):

```
private void revPrint(ListNode listRef)
{
  if (listRef != null)
  {
    revPrint(listRef.next);
    System.out.println(" " + listRef.info);
  }
}
```

Notice that `revPrint` is a private method of the `SortedLinkedList` class. Could we make `revPrint` a public method instead? The answer is no, and here is the reason. To print the whole linked list, the client's initial call to `revPrint` must pass as an argument the reference to the first node in the list. But in our Linked List classes this reference (`list`) is a protected instance variable of the class, so the following client code is not permitted:

```
myList.revPrint(list);      // Not allowed--list is private
```

Therefore, we must create `revPrint` as an auxiliary, private method and define a public method, say, `PrintReversed`, which calls `revPrint`:

```
public void printReversed()
{
  revPrint(list);
}
```

This pattern of defining a private recursive method, with a public entry method, has been used in many of our examples. Given this design, the client can print the entire list with the following method invocation:

```
myList.PrintReversed();
```

Let's verify `revPrint` using the Three-Question Method.

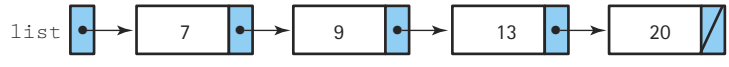
1. *The Base-Case Question:* The base case is implied. When `listRef` is equal to `null`, we return to the statement following the last recursive call to `revPrint`, and no further recursive calls are made. The answer is yes.
2. *The Smaller-Caller Question:* The recursive call passes the list referenced by `listRef.next`, which is one node smaller than the list referenced by `listRef`. The answer is yes.
3. *The General-Case Question:* We assume that `revPrint (listRef.next)` correctly prints out the rest of the list in reverse order; this call, followed by the statement printing the value of the first element, gives us the whole list, printed in reverse order. So the answer is yes.

How would you change the method `revPrint` (in addition to changing its name) to make it print out the list in forward rather than reverse order? How would you use the approach described in this section at the client level to print a list in reverse order? We leave these as exercises. Note that the recursive approach to printing a list backwards, described in this subsection, is used in the `SortedLinkedList2` class of the `ch07.genericLists` package, available on the web site.

The Insert Operation

Inserting an item into a linked implementation of a sorted list requires two references: one referencing the node being examined and one referencing the node behind it. We need this trailing reference because by the time we discover where to insert a node, we are beyond the node that needs to be changed. In the recursive version, we let the recursive process take care of the trailing reference.

Let's begin by looking at an example where the `item` type is `int`.



If we insert 11, we begin by comparing 11 to the value in the first node of the list, 7. Eleven is greater than 7, so we look for the insertion point in the sublist referenced by the `next` reference of the first node. This new list is one node shorter than the original list. We compare 11 to the value in the first node in this list, 9. Eleven is greater than 9, so we look for the insertion point in the sublist referenced by the `next` link of the first node. This new list is one node shorter than the current list. We compare 11 with the value in the first node of this new list, 13. Eleven is less than 13, so we have found the insertion point. We insert a new node with 11 as the value at the beginning of the list we are examining, at the beginning of the sublist consisting of 13 and 20. We then set the value of the `next` reference of the node containing 9 to reference this new list.

What if the value we are inserting is greater than the value in the last node of the list? In this case, we eventually examine an empty sublist and we insert the value into "the beginning" of this empty list.

Recursive Insert (subList, item) returns list

Definition:	Insert item into the sorted list referenced by subList.
Size:	The number of items in subList.
Base Cases:	(1) If subList is empty, insert item into the empty list, and return this new list. (2) If item is less than first element on subList insert item at the beginning of subList, and return this new list.
General Case:	Set subList to the list returned by recursive Insert(subList.next, item) and return subList.

The key to recursive insertion into a sorted linked list is to realize that a list minus its first node is still a list (we call this the `subList`) and that the easiest place to insert a node into a linked list is at its beginning. Combining these insights we develop the following recursive insertion algorithm:

recursiveInsert (subList, item): Returns List

```
if subList is empty
   return a list consisting of just item
else if item is less than the first item on subList
   insert item onto the front of subList
   return this new list
else
   subList.next = recursiveInsert(subList.next, item)
   return subList
```

Note that the algorithm returns a list. The idea is to start processing with a call such as

```
list = recursiveInsert(list, item);
```

so that our `list` becomes the result of inserting the `item` into our `list`.

Let's analyze the algorithm a little more closely. The first two *if* statements cover our base cases of inserting into an empty `subList` and inserting into a `subList` where

item belongs at its beginning (the "easiest place to insert"). Logically, these are two separate base cases, but it turns out that their implementation is identical. In both cases we create a new node, set its info attribute to item, and set its next reference to subList. (If subList is "empty" this effectively sets the next reference to null; if not this sets the next reference to the front of subList.) If we fall through to the general case we know that the current subList is not empty and that our item does not belong at the beginning of it. Therefore, we set the next reference of the current subList to the result returned by inserting item into the list consisting of the "rest" of the current subList, that is, we set it to the subList minus its first node. We also return a reference to the current subList to whatever called recursiveInsert.

The method is coded below, for our SortedLinkedList2 class. This class is found in the ch07.genericLists package. As we have done for previous examples, we keep the insert method's signature unchanged and create a private recursiveInsert method that is invoked by the public insert. The new version of insert is also listed below. Note that as the system works it way out of the series of recursive calls it repeatedly reassigns the references in the next links of the chain of nodes, to the same values that they already contain. This apparent redundancy is needed so that the placement of the new inserted node into the chain occurs properly.

```
private ListNode recursiveInsert(ListNode subList, Listable item)
{
  if (subList == null || item.compareTo(subList.info) < 0)
  {
    // Insert new node at the front of this sublist
    ListNode newNode = new ListNode();
    newNode.info = item;
    newNode.next = subList;

    // Return reference to new node
    return newNode;
  }
  else
  {
    // Recursively insert item into list referenced by next
    //  field of current sublist
    subList.next = recursiveInsert(subList.next, item);

    // Return reference to this sublist
    return subList;
  }
}

public void insert(Listable item)
// Adds a copy of item to list
{
  list = recursiveInsert(list, item);
}
```

Figure 7.5 shows visually what the code does in the case of our example of inserting 11 into the list shown above, a general case situation. We leave it to the reader to create similar figures for the base cases.

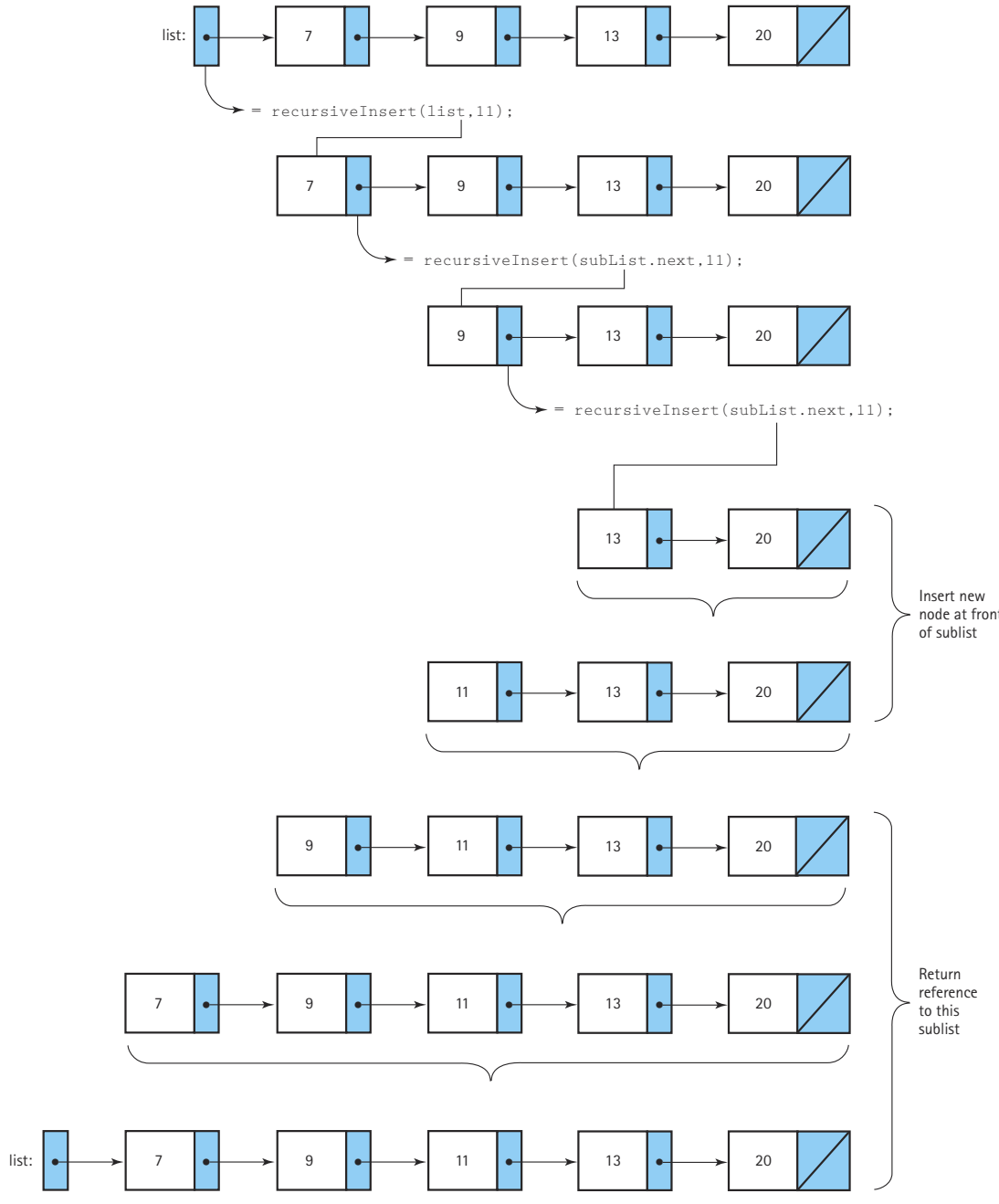


Figure 7.5 *Using* `recursiveInsert`

7.8 How Recursion Works

In order to understand how recursion works and why some programming languages allow it and some do not, we have to take a detour and look at how languages associate places in memory with variable names. The association of a memory address with a variable name is called *binding*. The point in the compile/execute cycle when binding occurs is called the *binding time*. We want to stress that binding time refers to a point of time in a process, not the amount of clock time that it takes to bind a variable.

As you know, Java is usually used as an interpreted language. When you compile a Java program it is translated into a language called Java bytecode. When you run a Java program, your machine's Java interpreter interprets the bytecode version of your program. The interpreter dynamically generates machine code based on the bytecode, and executes the machine code on your machine. You can also use a Java bytecode compiler to translate your bytecode files directly into machine code. In that case, you can run your programs directly on your machine, without having to use an interpreter. In either case, your Java programs must be transformed into the machine language of your machine, in order for you to run them. In this section, we discuss the machine language representation of programs. Programmers working with most other high-level languages typically use compilers that translate directly into machine language.

Static storage allocation associates variable names with memory locations at compile time; dynamic storage allocation associates variable names with memory locations at execution time. As we look at how static and dynamic storage allocation work, consider the following question: When are the parameters of a method bound to a particular address in memory? The answer to this question tells something about whether recursion can be supported.

Static Storage Allocation

As a program is being translated, the compiler creates a table called a *symbol table*. When the compiler reads a variable declaration, the variable is entered into the symbol table, and a memory location—an address—is assigned to it. For example, let's see how the compiler would translate the following Java global declarations:

```
int girlCount, boyCount, totalKids;
```

To simplify this discussion, we assume that integers take only one memory location. This statement causes three entries to be made in the symbol table. (The addresses we use are arbitrary.)

Symbol	Address
`girlCount`	0000
`boyCount`	0001
`totalKids`	0002

That is, at compile time,

`girlCount` is bound to address 0000.

`boyCount` is bound to address 0001.

`totalKids` is bound to address 0002.

Whenever a variable is used later in the program, the compiler searches the symbol table for its actual address and substitutes that address for the variable name. After all, meaningful variable names are for the convenience of the human reader; addresses, however, are meaningful to computers. For example, the assignment statement

```
totalKids = girlCount + boyCount;
```

is translated into machine instructions that execute the following actions:

- Get the contents of address 0000.
- Add it to the contents of address 0001.
- Put the result into address 0002.

The object code itself can then be stored in a different part of memory. Let's say that the translated instructions begin at address 1000. At the beginning of execution of the program, control is transferred to address 1000. The instruction stored there is executed, then the instruction in 1001 is executed, and so on.

Where are the parameters of methods stored? With static storage allocation, the parameters of a method are assumed to be in a particular place; for instance, the compiler might set aside space for the parameter values immediately preceding the code for each method. Consider a method `countKids`, with two `int` parameters, `girlCount` and `boyCount`, as well as a local variable `totalKids`. Let's assume that the method's code begins at an address we call `CountKids`. The compiler leaves room for the two parameters and the local variable at addresses `CountKids - 1`, `CountKids - 2`, and `CountKids - 3`, respectively. Given this method heading and declaration:

```
void countKids(int girlCount, int boyCount)
{
  int totalKids;
  .
  .
}
```

the statement

```
totalKids = girlCount + boyCount;
```

in the body of the method would generate the following actions:

- Get the contents of address `CountKids - 1`.
- Add it to the contents of address `CountKids - 2`.
- Store the result in address `CountKids - 3`.

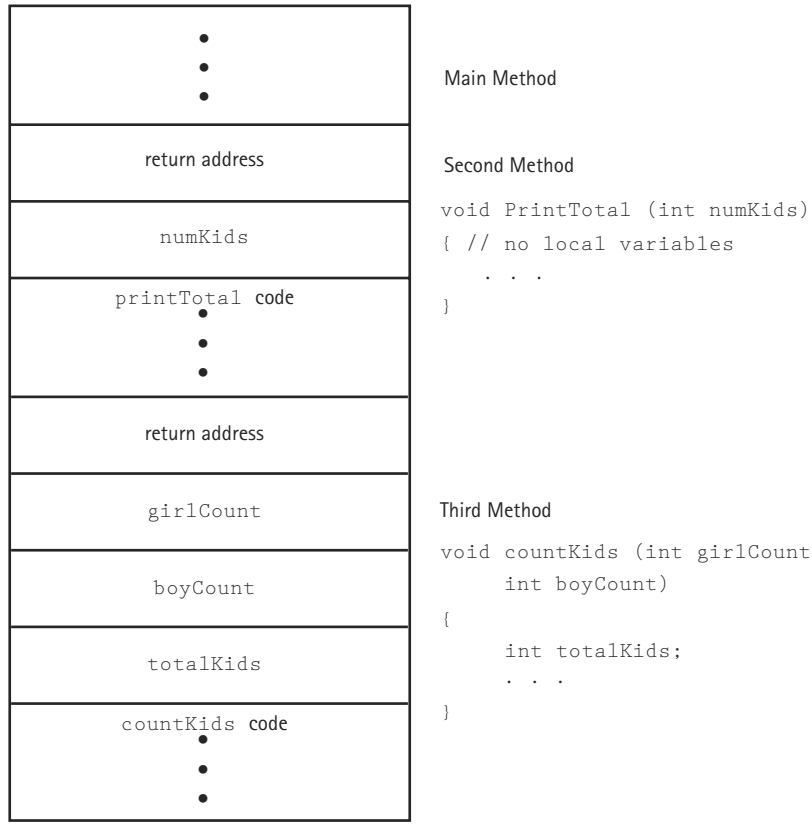


Figure 7.6 *Static allocation of space for a program with three methods*

Figure 7.6 shows how a program with three methods might be arranged in memory.

This discussion has been simplified somewhat, because the compiler sets aside space not only for the parameters and local variables, but also for the return address (the location in the calling code of the next instruction to process, following the completion of the method) and the computer's current register values. However, we have illustrated the main point: The method's parameters and local variables are bound to actual addresses in memory at compile time in this scheme.

We can compare the static allocation scheme to one way of allocating seats in an auditorium where a lecture is to be held. A finite number of invitations are issued for the event, and the exact number of chairs needed are set up before the lecture. Each invited guest has a reserved seat. If anyone brings friends, however, there is nowhere for them to sit.

What is the implication of binding variable names to memory locations before the program executes? Each parameter and local variable has but a single location assigned to it at compile time. (They are like invited guests with reserved seats.) If each call to a method is an independent event, there is no problem. But in the case of recursion, each

recursive call requires space to hold its own values. Where is the storage for the multiple versions of the parameters and local variables generated by recursive calls? Because the intermediate values of the parameters and local variables must be retained, the recursive call cannot store its arguments in the fixed number of locations that were set up at compile time. The values from the previous recursive call would be overwritten and lost. Thus, a language that uses only static storage allocation cannot support recursion.

Dynamic Storage Allocation

The situation we have described is like a class of students that must share one copy of a workbook. Joe writes his exercise answers in the space provided in the workbook, then Mary erases his answers, and writes hers in the same space. This process continues until each student in the class writes his or her answers into the workbook, obliterating all the answers that came before. Obviously, this situation is not practical. Clearly, what is needed is for each student to read from the single copy of the workbook, then to write his or her answers on a separate piece of paper. In computer terms, what each invocation of a method needs is its own work space. Dynamic storage allocation provides this solution.

With dynamic storage allocation, variable names are not bound to actual addresses in memory until run time. The compiler references variables not by their actual addresses, but by relative addresses. Of particular interest to us, the compiler references the parameters and local variables of a method relative to some address known at run time, not relative to the location of the method's code.

Let's look at a simplified version of how this might work in Java. (The actual implementation depends on the particular machine/compiler/interpreter/bytecode-compiler.) When a method is invoked, it needs space to keep its parameters, its local variables, and the return address (the address in the calling code to which the computer returns when the method completes its execution). Just like students sharing one copy of a workbook, each invocation of a method needs its own work space. This work space is called an activation record or stack frame. Consider our recursive `factorial` method:

Activation record (stack frame) A record used at run time to store information about a method call, including the parameters, local variables, register values, and return address

```
public static int factorial(int number)
{
  if (number == 0)
    return 1;           // Base case
  else
    return (number * factorial (number - 1));      // General case
}
```

A simplified version of an activation record for method `factorial` might have the following "declaration":

```
class ActivationRecordType
{
  AddressType returnAddr;     // Return address
  int result;                 // Returned value
  int number;                 // Parameter
  .
  .
  .
};
```

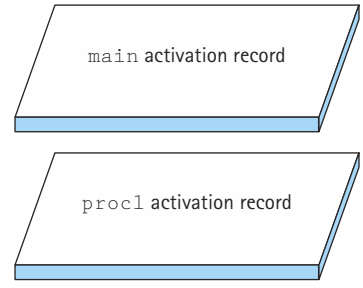
Each call to a method, including recursive calls, generates a new activation record. Within the method, references to the parameters and local variables use the values in the activation record. When the method ends, the activation record is released. How does this happen? Your source code doesn't need to allocate and free activation records; the compiler adds a "prologue" to the beginning of each method and an "epilogue" to the end of each method. Table 7.1 compares the source code for `factorial` with a simplified version of the "code" executed at run time. (Of course, the code executed at run

Table 7.1 *Run-time Version of `Factorial` (Simplified)*

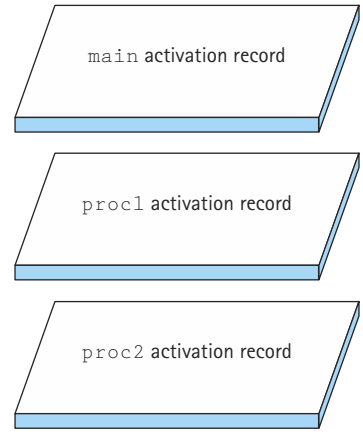
What Your Source Code Says	What the Run-time System Does
`int factorial(int number)`	`// Method prologue`
`{`	`actRec = new ActivationRecordType;`
	`actRec.returnAddr = retAddr;`
	`actRec.number = number;`
`  if (number == 0)`	`// actRec->result is undefined`
`    return 1;`	`if (actRec.number == 0)`
`  else`	`  actRec.result = 1;`
`    return number *`	`else`
`     factorial(number - 1);`	`  actRec.result =`
`}`	`    actRec.number *`
	`    factorial(actRec.number-1);`
	`// Method epilogue`
	`returnValue = actRec.result;`
	`retAddr = actRec.returnAddr;`
	`delete actRec;`
	`Jump (goto) retAddr`

time is machine code, but we are listing the source-code equivalent so that it makes sense to the reader.)

What happens to the activation record of one method when a second method is invoked? Consider a program whose `main` method calls `proc1`, which then calls `proc2`. When the program begins executing, the "main" activation record is generated (the `main` method's activation record exists for the entire execution of the program). At the first method call, an activation record is generated for `proc1`.[1]



When `proc2` is called from within `proc1`, its activation record is generated. Because `proc1` has not finished executing, its activation record is still around; just like the mathematicians with telephones, one waits "on hold" until the next call is finished:



When `proc2` finishes executing, its activation record is released. But which of the other two activation records becomes the active one: `proc1`'s or `main's`? `proc1`'s activation record should now be active, of course. The order of activation follows the Last-In-First-Out rule. We know of a structure that supports LIFO access—the stack—so it

[1]The drawings in this chapter that represent the run-time stack have the top of the stack at the bottom of the picture because we generally think of memory being allocated in increasing address order.

should come as no surprise that the structure that keeps track of the activation records at run time is called the run-time stack.

Run-time stack A data structure that keeps track of activation records during the execution of a program

When a method is invoked, its activation record is pushed onto the run-time stack. Each nested level of method calls adds another activation record to the stack. As each method completes its execution, its activation record is popped from the stack. Recursive method calls, like calls to any other methods, cause a new activation record to be generated. The level of recursive calls in a program determines how many activation records for this method are pushed onto the run-time stack at any one time.

Using dynamic allocation might be compared to another way of allocating seats in an auditorium where a lecture has been scheduled. A finite number of invitations is issued, but each guest is asked to bring his or her own chair. In addition, each guest can invite an unlimited number of friends, as long as they all bring their own chairs. Of course, if the number of extra guests gets out of hand, the space in the auditorium runs out, and there may not be enough room for any more friends or chairs. Similarly, the level of recursion in a program is limited by the amount of memory available in the run-time stack.

Let's walk through method `factorial` again to see how its execution affects the run-time stack. Let's say that the `main` method is loaded in memory beginning at location 5000, and that the initial call to `factorial` is made in a statement at memory location 5200. Suppose also that the `factorial` method is loaded in memory at location 1000, with the recursive call made in the statement at location 1010. Figure 7.7 shows a

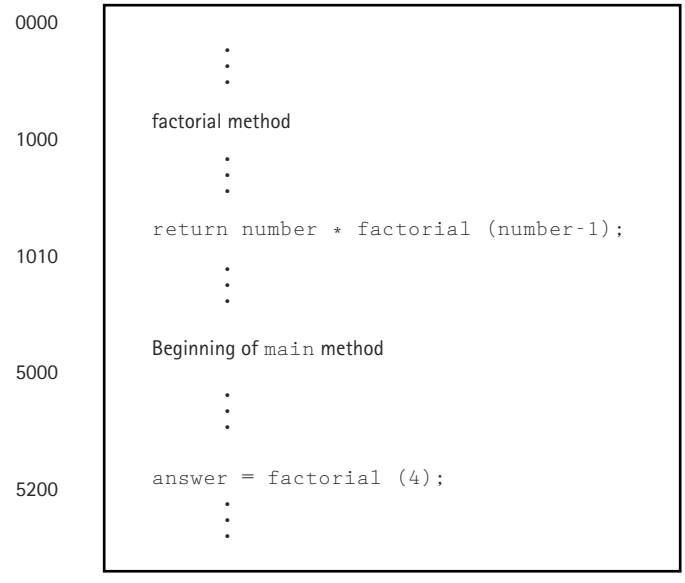


Figure 7.7 *The sample program loaded in memory*

simplified version of how this example program is loaded in memory. (These numbers have been picked arbitrarily, so that we have actual numbers to show in the return address of the activation record.)

When `factorial` is called the first time from the statement in the `main` method at address 5400:

```
answer = factorial(4);
```

an activation record is pushed onto the run-time stack to hold three pieces of data: the return address (5400), the parameter `number` (4), and the value returned from the method (`result`), which has not yet been evaluated. Rather than showing our activation records as pictures, we show it as a table. Each new activation record is a new row of the table. This activation record in the last row of the table is now on the top of the run-time stack. We have added a column on the left that identifies which call it is.

Call	*number*	*result*	*returnAddr*	
1	4	?	5200	⟵ top

The code is now executed. Is `number` (the `number` value in the top activation record) equal to 0? No, it is 4, so the *else* branch is taken:

```
return number * factorial(number - 1);
```

This time the method `factorial` is called from a different place. It is called recursively from within the method itself, from the statement at location 1010. After the value of `factorial(number - 1)` is calculated, we return to this location to multiply the `result` times `number`. A new activation record is pushed onto the run-time stack:

Call	*number*	*result*	*returnAddr*	
1	4	?	5200	
2	3	?	1010	⟵ top

The code for the new invocation of `factorial` begins executing. Is `number` (the `number` value in the top activation record) equal to 0? No, it is 3, so the *else* branch is taken

```
return number * factorial(number - 1);
```

So the method `factorial` is again called recursively from the instruction at location 1010. This process continues until the situation looks as shown below with the fifth call.

Call	*number*	*result*	*returnAddr*	
1	4	?	5200	
2	3	?	1010	
3	2	?	1010	
4	1	?	1010	
5	0	?	1010	⟵ top

Now, as the fifth call is executed, we again ask the question: Is `number` (the `number` value in the top activation record) equal to 0? Yes. This time we perform the *then* clause, storing the value 1 into `result` (the `result` in the top activation record, that is). The fifth invocation of the method has executed to completion, and the value of `result` in the top activation record is returned from the method. The run-time stack is popped to release the top activation record, leaving the activation record of the fourth call to `factorial` at the top of the run-time stack. We don't restart the fourth method call from the beginning, however. As when we return from any method call, we return to the place where the method was called. This place was recorded as the return address (location 1010) stored in the activation record.

Next, the returned value (1) is multiplied by the value of `number` in the top activation record (1) and the result (1) is stored into `result` (the instance of `result` in the top activation record, that is). Now the fourth invocation of the method is complete, and the value of `result` in the top activation record is returned from the method. Again the run-time stack is popped to release the top activation record and a multiplication occurs, leaving the activation record of the third call to `factorial` at the top of the run-time stack.

Call	*number*	*result*	*returnAddr*	
1	4	?	5200	
2	3	?	1010	
3	2	2	1010	⟵ top

This process continues until we are back to the first call:

Call	*number*	*result*	*returnAddr*	
1	4	?	5200	⟵ top

and 6 has just been returned as the value of `factorial(number - 1)`. This value is multiplied by the value of `number` in the top activation record (that is, 4) and the result,

24, is stored into the `result` field of the top activation record. This assignment completes the execution of the initial call to method `factorial`. The value of `result` in the top activation record (24) is returned to the place of the original call (address 5200), and the activation record is popped. This leaves the `main` activation record at the top of the run-time stack. The final value of `result` is stored into the variable `answer`, and the statement following the original call is executed.

Depth of recursion The number of recursive calls used to complete an original call of a recursive method

The number of recursive calls is the depth of the recursion. Notice the relationship between the complexity of the iterative version of `factorial` in terms of Big-O notation and the depth of recursion for the recursive version. Both are based on the parameter `number`. Is it a coincidence that the depth of recursion is the same as the complexity of the iterative version? No. Recursion is another way of doing repetition, so you would expect that the depth of recursion would be approximately the same as the number of iterations for the iterative version of the same problem. In addition, they are both based on the size of the problem.

7.9 Removing Recursion

In cases where a recursive solution is not desired, either because the language doesn't support recursion or because the recursive solution is deemed too costly in terms of space or time, a recursive algorithm can be implemented as a nonrecursive method. There are two general techniques that are often substituted for recursion: iteration and stacking.

Iteration

When the recursive call is the last action executed in a recursive method, an interesting situation occurs. The recursive call causes an activation record to be put on the run-time stack to contain the method's arguments and local variables. When this recursive call finishes executing, the run-time stack is popped and the previous values of the variables are restored. But, because the recursive call is the last statement in the method, the method terminates without using these values. The pushing and popping of activation records is superfluous. All we really need to do is to change the "smaller-caller" variable(s) on the recursive call's parameter list, and "jump" back to the beginning of the method. In other words, we really need a loop.

For instance, it is not necessary to use recursion for our array-based Unsorted List `isThere` operation, as we did in Section 7.4. However, it is simple to remove the recursion from this method. (Let's assume we have our recursive solution and want to use it to generate an iterative solution.) The last statement executed in the general case is the recursive call to itself. Let's see how to replace the recursion with a loop. First we revisit the code for the recursive approach:

```
public boolean isThere (Listable item, int startPosition)
// Returns true if item is on this list; otherwise, returns false
{
  if (item.compareTo(list[startPosition]) == 0)  // If they match
    return true;
  else if (startPosition == (numItems - 1))      // If end of list
    return false;
  else return isThere(item, startPosition + 1);
}
```

The recursive solution has two base cases; one occurs if we find the value and the other occurs if we reach the end of the list without finding the value. The base cases solve the problem without further executions of the method. In the iterative solution, the base cases become the terminating conditions of the loop:

```
while (moreToSearch && !found)
```

When the terminating conditions are met, the problem is solved without further executions of the loop body.

In the general case of the recursive solution, `isThere` is called to search the remaining, unsearched part of the list. Each recursive execution of the method processes a smaller version of the problem. The smaller-caller question is answered affirmatively because `startPosition` is incremented, shrinking the unsearched part of the list on every recursive call. Similarly, in an iterative solution, each subsequent execution of the loop body processes a smaller version of the problem. The unsearched part of the list is shrunk on each execution of the loop body by incrementing `location`. Here is the iterative version of the method:

```
public boolean isThere (Listable item)
// Returns true if item is on this list; otherwise, returns false
{
  boolean moreToSearch;
  int location = 0;
  boolean found = false;

  moreToSearch = (location < numItems);

  while (moreToSearch && !found)
  {
    if (item.compareTo(list[location]) == 0)  // If they match
      found = true;
    else
    {
      location++;
```

```
      moreToSearch = (location < numItems);
    }
  }

 return found;
}
```

Tail recursion The case in which a method contains only a single recursive invocation and it is the last statement to be executed in the method

Cases in which the recursive call is the last statement executed are called tail recursion. Note that the recursive call is not necessarily the last statement in the method. For instance, the recursive call in the following version of `isThere` is still tail recursion, even though it is not the last statement in the method:

```
public boolean isThere(Listable item, int startPosition)
{
  if (item.compareTo(list[startPosition]) == 0)  // If they match
    return true;
  else if (startPosition != (numItems - 1))      // If not end of list
    return isThere(item, startPosition + 1);
  else return false;
}
```

The recursive call is the last statement executed in the general case—thus it is tail recursion. Tail recursion is usually replaced by iteration to remove recursion from the solution. In fact, many compilers catch tail recursion and automatically replace it with iteration.

Stacking

When the recursive call is not the last action executed in a recursive method, we cannot simply substitute a loop for the recursion. For instance, consider method `revPrint`, developed in Section 7.7 for printing a linked list in reverse order:

```
private void revPrint(ListNode listRef)
{
  if (listRef != null)
  {
    revPrint(listRef.next);
    System.out.println(" " + listRef.info);
  }
}
```

Here we make the recursive call and then print the value in the current node. In cases like this, we must replace the stacking that was done by the system with stacking that is done by the programmer, in order to remove the recursion.

How would we write method `revPrint` nonrecursively? As we traverse the list, we must keep track of the reference to each node, until we reach the end of the list (when our traversing reference equals `null`). When we reach the end of the list, we print the `info` value of the last node. Then we back up and print again, back up and print, and so on, until we have printed the first list element.

We know of a data structure in which we can store references and retrieve them in reverse order: the stack. The general task for `revPrint` becomes:

RevPrint (iterative)

```
Create an empty stack of node references.
Set listRef to reference the first node in list
while listRef is not null
    Push listRef onto the stack
    Advance listRef
while the stack is not empty
    Pop the stack to get listRef (to previous node)
    Print listRef.info
```

A nonrecursive `revPrint` method may be coded as follows. Note that we can now make `revPrint` a public method of class `SortedLinkedList` instead of a helper method. Because `revPrint` no longer has a parameter, we don't have to deal with the problem of having the client pass the (inaccessible) pointer to the beginning of the linked list. We use the `ArrayStack` stack developed in Chapter 4 to hold the stacked references.

```
public void revPrint()
// Prints this list in reverse order
{
  ArrayStack stack = new ArrayStack();
  ListNode listRef;

  listRef = list;

  while (listRef != null)    // Put references onto the stack
  {
    stack.push(listRef);
```

```
      listRef = listRef.next;
    }

    // Retrieve references in reverse order and print elements
    while (!stack.isEmpty())
    {
      listRef = (ListNode)stack.top();
      stack.pop();
      System.out.println(" " + listRef.info);
    }
  }
```

Notice that the nonrecursive version of `revPrint` is quite a bit longer than its recursive counterpart, especially if we add in the code for the stack methods `push`, `pop`, `top`, and `isEmpty`. This verbosity is caused by our need to stack and unstack the references explicitly. In the recursive version, we just called `revPrint` recursively and let the run-time stack keep track of the references. The nonrecursive version of `revPrint` appears in our `SortedLinkedList2` class in the `ch07.genericLists` package.

7.10 Deciding Whether to Use a Recursive Solution

There are several factors to consider in deciding whether or not to use a recursive solution to a problem. The main issues are the clarity and the efficiency of the solution. Let's talk about efficiency first. In general, a recursive solution is more costly in terms of both computer time and space. (This is not an absolute decree; it really depends on the problem, the computer, and the compiler.) A recursive solution usually requires more "overhead" because of the nested recursive method calls, in terms of both time (the method prologues and epilogues must be run for each recursive call) and space (activation records must be created). A call to a recursive method may hide many layers of internal recursive calls. For instance, the call to an iterative solution to `factorial` involves a single method invocation, causing one activation record to be put on the run-time stack. Invoking the recursive version of `factorial`, however, requires $n + 1$ method calls and $n + 1$ activation records to be pushed onto the run-time stack, where n represents the parameter `number`. That is, the depth of recursion is $O(n)$. For some problems, the system just may not have enough space in the run-time stack to run a recursive solution.

Another problem to look for is the possibility that a particular recursive solution might just be inherently inefficient. Such inefficiency is not a reflection of how we choose to implement the algorithm; rather, it is an indictment of the algorithm itself. For instance, look back at method `combinations`, which we discussed in Section 7.5. The example of this method illustrated in Figure 7.3, `combinations(4,3)`, seems straightforward enough. But consider the execution of `combinations(6,4)`, as illustrated in Figure 7.8. The inherent problem with this method is that the same

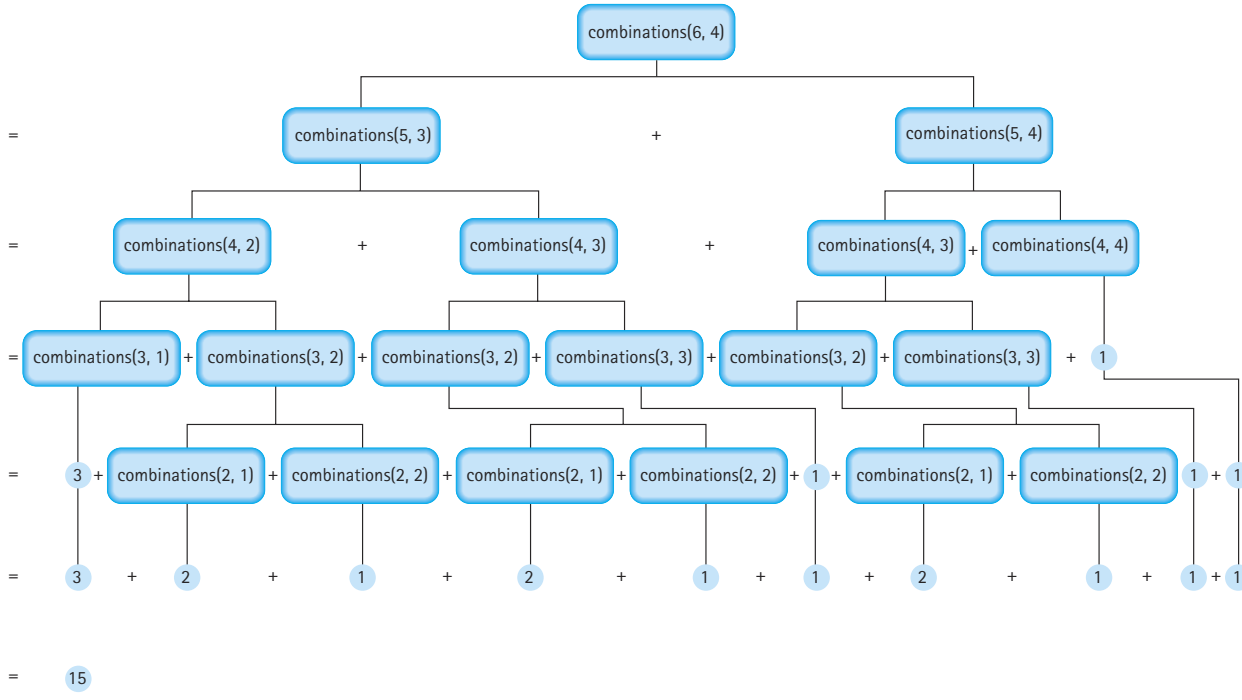


Figure 7.8 *Calculating* `combinations(6,4)`

values are calculated over and over. `combinations(4,3)` is calculated in two different places, and `combinations(3,2)` is calculated in three places, as are `combinations(2,1)` and `combinations(2,2)`. It is unlikely that we could solve a combinatorial problem of any large size using this method. The problem is that the program runs "forever"—or until it exhausts the capacity of the computer; it is an exponential-time, $O(2^N)$ solution to a linear-time, $O(N)$, problem. Although our recursive method is very easy to understand, it was not a practical solution. In such cases, you should seek an alternate solution. A programming approach called *dynamic programming*, where solutions to subproblems that are needed repeatedly are saved in a data structure instead of being recalculated, can often be used. Or even better, discover an iterative solution, as we did for combinations in the feature section in Section 7.5.

The issue of the clarity of the solution is still an important factor, however. For many problems, a recursive solution is simpler and more natural for the programmer to write. The total amount of work required to solve a problem can be envisioned as an iceberg. By using recursive programming, the applications programmer may limit his or her view to the tip of the iceberg. The system takes care of the great bulk of the work below the surface. Compare, for example, the recursive and nonrecursive versions of method `revPrint`. In the recursive version, we let the system take care of the stacking

that we had to do explicitly in the nonrecursive method. Thus, recursion is a tool that can help reduce the complexity of a program by hiding some of the implementation details. With the cost of computer time and memory decreasing and the cost of a programmer's time rising, it is worthwhile to use recursive solutions to such problems.

To summarize, it is good to use recursion when:

- The depth of recursive calls is relatively "shallow," some fraction of the size of the problem. For instance, the level of recursive calls in the `binarySearch` method is $O(\log_2 N)$; this is a good candidate for recursion. The depth of recursive calls in the `factorial` and `isThere` methods, however, is $O(N)$.
- The recursive version does about the same amount of work as the nonrecursive version. You can compare the Big-O approximations to determine this. For instance, we have determined that the $O(2^N)$ recursive version of `combinations` is a poor use of recursion, compared to an $O(N)$ iterative version. Both the recursive and iterative versions of `binarySearch`, however, are $O(\log_2 N)$. `binarySearch` is a good example of a recursive method.
- The recursive version is shorter and simpler than the nonrecursive solution. By this rule, `factorial` and `isThere` are not good uses of recursive programming. They illustrate how to understand and write recursive methods, but they could more efficiently be written iteratively—without any loss of clarity in the solution. `revPrint` is a better use of recursion. Its recursive solution is very simple to understand, and the nonrecursive equivalent is much less elegant.

Summary

Recursion is a very powerful computing tool. Used appropriately, it can simplify the solution of a problem, often resulting in shorter, more easily understood source code. As usual in computing, there are tradeoffs: Recursive methods are often less efficient in terms of both time and space, due to the overhead of many levels of method calls. How expensive this cost is depends on the problem, the computer system, and the compiler.

A recursive solution to a problem must have at least one base case—that is, a case in which the solution is derived nonrecursively. Without a base case, the method recurses forever (or at least until the computer runs out of memory). The recursive solution also has one or more general cases that include recursive calls to the method. The recursive calls must involve a "smaller caller." One (or more) of the actual parameter values must change in each recursive call to redefine the problem to be smaller than it was on the previous call. Thus, each recursive call leads the solution of the problem toward the base case(s).

A typical implementation of recursion involves the use of a stack. Each call to a method generates an activation record to contain its return address, parameters, and local variables. The activation records are accessed in a Last-In-First-Out manner. Thus a stack is the choice of data structure. Recursion can be supported by systems and languages that use dynamic storage allocation. The method parameters and local variables are not bound to addresses until an activation record is created at run time.

Thus multiple copies of the intermediate values of recursive calls to the method can be supported, as new activation records are created for them.

With static storage allocation, in contrast, a single location is reserved at compile time for each parameter and local variable of a method. There is no place to store intermediate values calculated by repeated nested calls to the same method. Therefore, systems and languages with only static storage allocation cannot support recursion.

When recursion is not possible or appropriate, a recursive algorithm can be implemented nonrecursively by using a looping structure and, in some cases, by pushing and popping relevant values onto a stack. This programmer-controlled stack explicitly replaces the system's run-time stack. Although such nonrecursive solutions are often more efficient in terms of time and space, there is usually a tradeoff in terms of the elegance of the solution.

Summary of Classes and Support Files

The classes and files are listed in the order in which they appear in the text. The package a class belongs to, if any, is listed in parentheses under Notes. The class and support files are available on our web site. They can be found in the `ch07` subdirectory of the `bookFiles` directory.

Classes, Interfaces, and Support Files Defined in Chapter 7

File	1st Ref.	Notes
`tryFact.java`	page 483	This class contains the recursive and iterative versions of `factorial`
`UnsortedStringList3.java`	page 487	(`ch07.stringLists`) This class contains the recursive version of `isThere`
`tryComb.java`	page 490	This class contains the recursive version of `combinations`
`Towers.java`	page 494	The Towers of Hanoi program
`SortedList2.java`	page 498	(`ch07.genericLists`) This class contains the recursive version of Binary Search, used to implement the `isThere` method
`SortedLinkedList2.java`	page 501	(`ch07.genericLists`) This class contains both the recursive and iterative (stack based) versions of Reverse Print. The recursive version is invoked through `printReversed` and the iterative version is invoked through `revPrint`. The recursive version of `insert` is also implemented here.

There were no new library classes used in this chapter.

Exercises

The exercises for this chapter are divided into three units: Basics (Sections 1, 2, 3), Examples (Sections 4, 5, 6, 7), and Advanced (Sections 8, 9, 10).

Basics (Sections 1, 2, 3)

1. Explain what is meant by
   a. base case.
   b. general (or recursive) case.
   c. indirect recursion.
2. Use the Three-Question Method to verify the `isThere` method in this chapter.
3. Describe the Three-Question Method of verifying recursive methods in relation to an inductive proof.

Examples (Sections 4, 5, 6, 7)

Use the following method in answering Exercises 4 and 5:

```
int puzzle(int base, int limit)

{
  if (base > limit)
    return -1;
  else
    if (base == limit)
      return 1;
    else
      return base * puzzle(base + 1, limit);
}
```

4. Identify
   a. the base case(s) of method `puzzle`.
   b. the general case(s) of method `puzzle`.
5. Show what would be written by the following calls to the recursive method `puzzle`.
   a. `System.out.println(puzzle (14, 10));`
   b. `System.out.println(puzzle (4, 7));`
   c. `System.out.println(puzzle (0, 0));`
6. Given the following method:

```
int exer(int num)
{
  if (num == 0)
    return 0;
  else
    return num + exer(num + 1);
}
```

a. Is there a constraint on the values that can be passed as a parameter in order for this method to pass the smaller-caller test?

b. Is `exer(7)` a good call? If so, what is returned from the method?

c. Is `exer(0)` a good call? If so, what is returned from the method?

d. Is `exer(-5)` a good call? If so, what is returned from the method?

7. For each of the following recursive methods, identify the base and general cases and the legal initial argument values, and explain what the method does.

a.
```
int power(int base, int exponent)
{
  if (exponent == 0)
    return 1;
  else
    return (base * power(base, exponent-1));
}
```

b.
```
int factorial (int num)
{
  if (num > 0)
    return (num * factorial (num - 1));
  else
    if (num == 0)
      return 1;
}
```

c.
```
int recur(int n)
{
  if (n < 0)
    return -1;
  else if (n < 10)
    return 1;
  else
    return (1 + recur(n / 10);
}
```

d.
```
int recur2(int n)
{
  if (n < 0)
     return -1;
  else if (n < 10)
     return n;
  else
     return (n % 10) + recur2(n / 10);
}
```

8. You must assign the grades for a programming class. Right now the class is studying recursion, and they have been given this simple assignment: Write a recursive method `sumSquares` that takes a reference to a linked list of integer

elements and returns the sum of the squares of the elements. The list nodes contain an `info` variable of primitive type `int`, and a `next` reference of type `ListNode`.

Example:

`sumSquares(list)` yields (5 * 5) + (2 * 2) + (3 * 3) + (1 * 1) = 39

Assume that the list is not empty.

You have received quite a variety of solutions. Grade the methods below, marking errors where you see them.

a.
```
int sumSquares(ListNode list)
{
  return 0;
  if (list != null)
    return (list.info * list.info) + sumSquares(list.next));
}
```

b.
```
int sumSquares(ListNode list)
{
  int sum = 0;
  while (list != null)
  {
    sum = list.info + sum;
    list = list.next;
  }
  return sum;
}
```

c.
```
int sumSquares(ListNode list)
{
  if (list == null)
    return 0;
  else
    return list.info * list.info + sumSquares(list.next);
}
```

d.
```
int sumSquares(ListNode list)
{
  if (list.next == null)
    return list.info * list.info;
  else
    return list.info * list.info + sumSquares(list.next);
}
```

e.
```
int sumSquares(ListNode list)
{
  if (list == null)
    return 0;
  else
    return (sumSquares(list.next) *
            sumSquares(list.next));
}
```

9. The Fibonacci sequence is the series of integers

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89 ...

See the pattern? Each element in the series is the sum of the preceding two items. There is a recursive formula for calculating the *n*th number of the sequence:

$$\text{Fib}(N) = \begin{cases} N, & \text{if } N = 0 \text{ or } 1 \\ \text{Fib}(N-2) + \text{Fib}(N-1), & \text{if } N > 1 \end{cases}$$

a. Write a recursive method `fibonacci` that returns the *n*th Fibonacci number when passed the argument `n`.
b. Write a nonrecursive version of method `fibonacci`.
c. Write a driver to test your two versions of method fibonacci.
d. Compare the recursive and iterative versions for efficiency. (Use words, not Big-O notation.)
e. Can you think of a way to make the recursive version more efficient?

10. The following defines a function that calculates an approximation of the square root of a number, starting with an approximate answer (`approx`), within the specified tolerance (`tol`).

SqrRoot(number, approx, tol)=

$$\begin{cases} \text{approx}, & \text{if } |\ \text{approx}^2 - \text{number}\ | <= \text{tol} \\ \text{SqrRoot(number, (approx}^2 + \text{number)/(2*approx), tol)}, & \text{if } |\ \text{approx}^2 - \text{number}\ | > \text{tol} \end{cases}$$

a. What limitations must be made on the values of the parameters if this function is to work correctly?
b. Write a recursive method `sqrRoot` that implements the function `SqrRoot`.
c. Write a nonrecursive version of method `sqrRoot`.
d. Write a driver to test the recursive and iterative versions of method `sqrRoot`.

11. Given the following method[2]:

```
int ulam(int num)
{
  if (num < 2)
    return 1;
  else
    if (num % 2 == 0)
      return ulam(num / 2);
    else
      return ulam (3 * num + 1);
}
```

   a. What problems come up in verifying this method?

   b. How many recursive calls are made by the following initial calls:

   - ulam (7)
   - ulam (8)
   - ulam (15)

12. In Section 7.7 we implement a recursive version of the list insert operation, for sorted linked lists. Using that as a model, design and implement a recursive version of the list delete operation, for sorted linked lists. Note that the code for our recursive `insert` method is included in the `SortedLinkedList2` class.

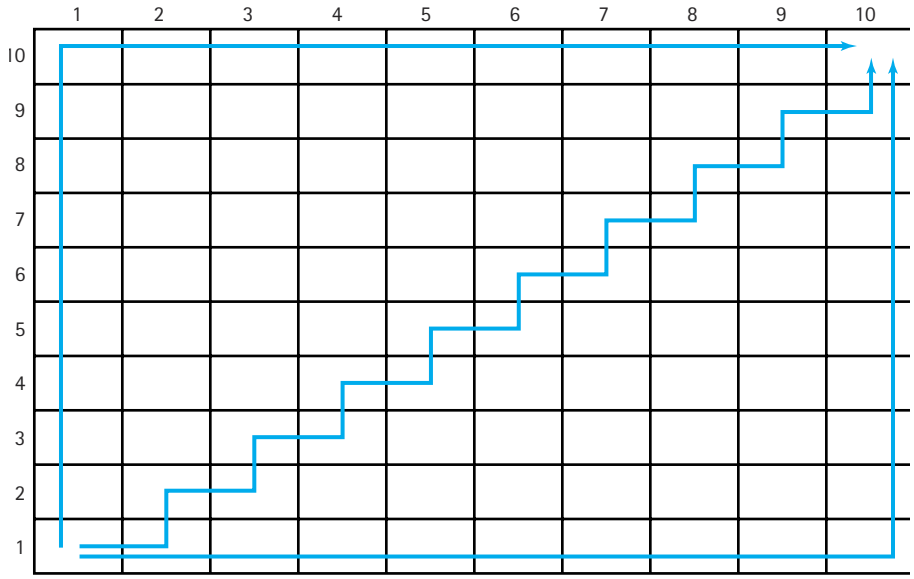
13. Using the recursive method `revPrint` as a model, write the recursive method `printList`, which traverses the elements in a linked list in forward order. Does one of these methods constitute a better use of recursion? If so, which one?

14. The recursive method `revPrint` prints the elements of a linked list in reverse order. It was written as a private method of the `SortedLinkedList2` class, which is called from the public method `printReversed`. A client program can print a list in reverse order by calling `printReversed`. Suppose the linked list class did not export a Print Reverse operation. Design a recursive client level method `clientRevPrint`, using the public list iteration methods, to print a linked list in reverse order.

15. We want to count the number of possible paths to move from row 1, column 1 to row *N*, column *N* in a two-dimensional grid. Steps are restricted to going up or to the right, but not diagonally. The illustration shows three of many paths, if *N* = 10:

---

[2]One of our reviewers pointed out that the proof of termination of this algorithm is a celebrated open question in mathematics. See *Programming Pearls* by Jon Bently for a discussion and further references.



a. The following method, `numPaths`, is supposed to count the number of paths, but it has some problems. Debug the method.

```
int numPaths(int row, int col, int n)
{
  if (row == n)
    return 1;
  else
    if (col == n)
      return (numPaths + 1);
    else
      return (numPaths(row + 1, col) * numPaths(row, col + 1))
}
```

b. After you have corrected the method, trace the execution of `numPaths` with n = 4 by hand. Why is this algorithm inefficient?

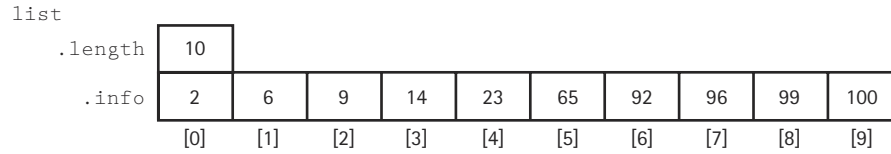
c. The efficiency of this operation can be improved by keeping intermediate values of `numPaths` in a two-dimensional array of integer values. This approach keeps the method from having to recalculate values that it has already done. Design and code a version of `numPaths` that uses this approach.

d. Show an invocation of the version of `numPaths` in part (c), including any array initialization necessary.

e. How do the two versions of `numPaths` compare in terms of time efficiency? Space efficiency?

Advanced (Sections 8, 9, 10)

16. Explain what is meant by:
    a. run-time stack.
    b. binding time.
    c. tail recursion.
17. True or False? Explain your answers. Recursive methods:
    a. often have fewer local variables than the equivalent nonrecursive methods.
    b. generally use *while* or *for* statements as their main control structure.
    c. are possible only in languages with static storage allocation.
    d. should be used whenever execution speed is critical.
    e. are always shorter and clearer than the equivalent nonrecursive methods.
    f. must always contain a path that does not contain a recursive call.
    g. are always less efficient, in terms of Big-O.
18. What data structure would you most likely see in a nonrecursive implementation of a recursive algorithm?
19. Explain the relationship between dynamic storage allocation and recursion.
20. What do we mean by binding time, and what does it have to do with recursion?
21. Given the following values in `list`,



show the contents of the run-time stack during the execution of this call to `binarySearch`:

```
binarySearch(99, 0, 9);
```

22. True or False? Explain your answers. A recursive solution should be used when:
    a. computing time is critical.
    b. the nonrecursive solution would be longer and more difficult to write.
    c. computing space is critical.
    d. your instructor says to use recursion.

Chapter 8

Binary Search Trees

Goals

Measurable goals for this chapter include that you should be able to

- define and use the following tree terminology:
  - binary tree
  - binary search tree
  - root
  - parent
  - child
  - ancestor
  - descendant
  - level
  - height
  - subtree
  - full
  - complete
- given a binary tree, identify the order the nodes would be visited for preorder, inorder, and postorder traversals
- define a binary search tree at the logical level
- show what a binary search tree would look like after a series of insertions and deletions
- implement the following binary search tree algorithms in Java:
  - finding an element
  - counting the number of nodes
  - inserting an element
  - deleting an element
  - retrieving an element
  - traversing a tree in preorder, inorder, and postorder
- discuss the Big-O efficiency of a given binary search tree operation
- describe an algorithm for balancing a binary search tree
- show how a binary tree can be represented as an array, with implicit positional links between the elements

We have described some of the advantages of using a linear linked list to store sorted information. One of the drawbacks of using a linear linked list is the time it takes to search a long list. A sequential or linear search of (possibly) all the nodes in the whole list is a O(N) operation. In Chapter 3, we saw how a binary search could find an element in a sorted list stored sequentially in an array. The binary search is a O($\log_2 N$) operation. It would be nice if we could use a binary search with a linked list, but there is no practical way to find the midpoint of a linked list of nodes. We can, however, reorganize the list's elements into a linked structure that is just perfect for binary searching: the binary search tree. The binary search tree provides us with a data structure that retains the flexibility of a linked list while allowing quicker [O($\log_2 N$) in the average case] access to any node in the list.

This chapter introduces some basic tree vocabulary and then develops the algorithms and implementations of the operations needed to use a binary search tree. The case study uses a binary search tree to calculate the frequency of words in a text file.

8.1 Trees

Each node in a singly linked list may point to one other node: the one that follows it. Thus, a singly linked list is a *linear* structure; each node in the list (except the last) has a unique successor. A tree is a nonlinear structure in which each node is capable of having many successor nodes, called *children*. Each of the children, being nodes in a tree, can also have many child nodes, and these children can also have many children, and so on, giving the tree its branching structure. The "beginning" of the tree is a unique starting node called the root. Trees are useful for representing hierarchical relationships among data items. Figure 8.1 shows three example trees; the first represents the chapters, sections, and subsections of this textbook, the second represents the hierarchical inheritance relationship among a set of Java classes, and the third represents a scientific classification of butterflies.

Tree A structure with a unique starting node (the root), in which each node is capable of having many child nodes, and in which a unique path exists from the root to every other node

Root The top node of a tree structure; a node with no parent

Trees are recursive structures. You can view any tree node as being the root of its own tree; such a tree is called a *subtree* of the original tree. For example, in Figure 8.1(a) the node labeled Chapter 1 is the root of a subtree containing all of the Chapter 1 material. There is one more defining quality of a tree: Its subtrees are disjoint; that is, they do not share any nodes. Another way of expressing this property is to say that there is a unique path from the root of a tree to any other node of the tree. This means that every node (except the root) has a unique parent. In the structure at the top of page 532, this rule is violated any way you look at it: the subtrees of A are not disjoint; there are two paths from the root to the node containing D; D has two parents. Therefore, this structure is not a tree.

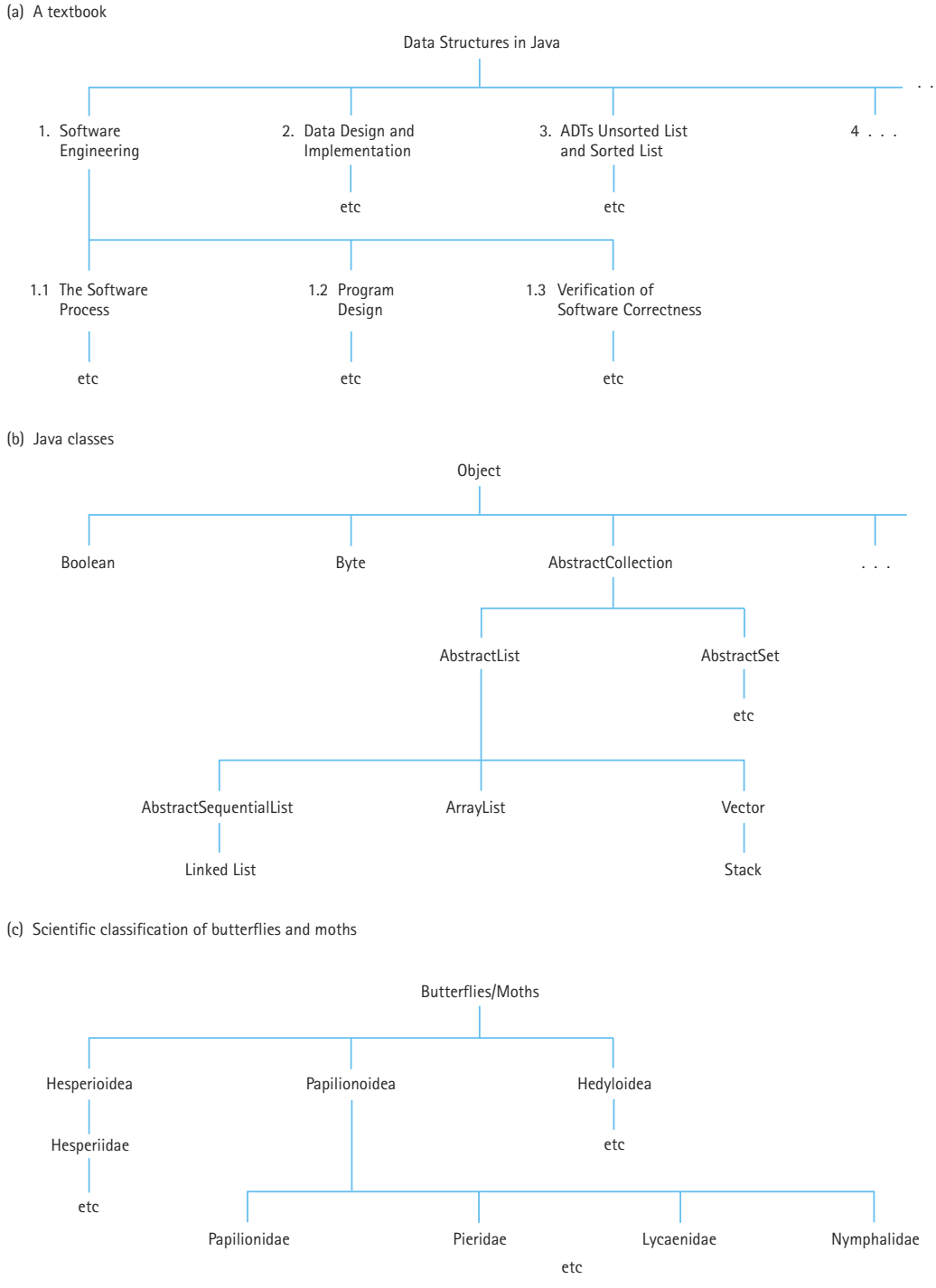
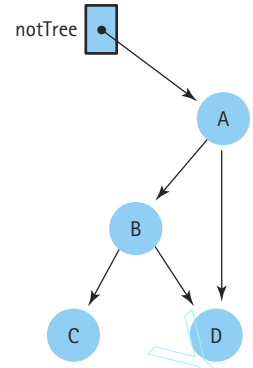


Figure 8.1 *Trees model hierarchical relationships*



Trees are useful structures. In this chapter, we concentrate on a particular form of tree: the binary tree. In fact, we concentrate on a particular type of binary tree: the binary search tree.

Binary Trees

A binary tree is a tree where each node is capable of having two children. Figure 8.2 depicts a binary tree. The root node of this binary tree contains the value A. Each node in the tree may have 0, 1, or 2 children. The node to the left of a node, if it exists, is called its *left child*. For instance, the left child of the root node contains the value B. The node to the right of a node, if it exists, is its *right child*. The right child of the root node contains the value C. The root node is the parent of the nodes containing B and C. If a node in the tree has no children, it is called a leaf. For instance, the nodes containing G, H, E, I, and J are leaf nodes.

Binary tree A tree in which each node is capable of having two child nodes, a left child node and a right child node

Leaf A tree node that has no children

Note, in Figure 8.2, that each of the root node's children is itself the root of a smaller binary tree, or subtree. The root node's left child, containing B, is the root of its *left subtree*, while the right child, containing C, is the root of its *right subtree*. In fact, any node in the tree can be considered the root node of a binary subtree. The subtree whose root node has the value B also includes the nodes with values D, G, H, and E. These nodes are the *descendants* of the node containing B. The descendants of the node containing C are the nodes with the values F, I, and J. The leaf nodes have no descendants. A node is the *ancestor* of another node if it is the parent of the node, or the parent of some other ancestor of that node. (Yes, this is a recursive definition.) The ancestors of the node with the value G are the nodes containing D, B, and A. Obviously, the root of the tree is the ancestor of every other node in the tree, but the root node has no ancestors itself.

The level of a node refers to its distance from the root. Therefore, in Figure 8.2 the level of the node containing A (the root node) is 0 (zero), the level of the nodes contain-

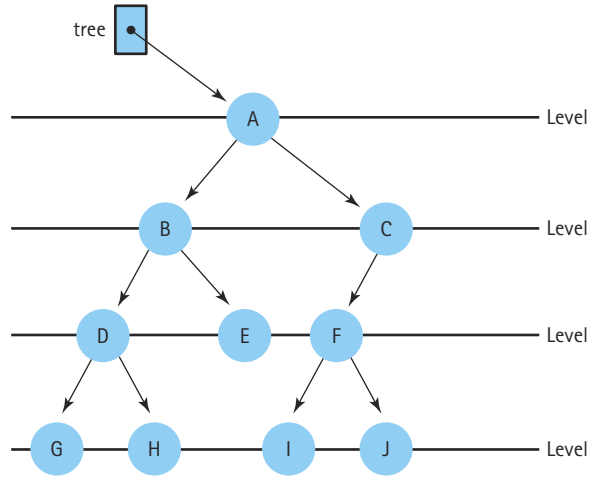


Figure 8.2 *A binary tree*

ing B and C is 1, the level of the nodes containing D, E, and F is 2, and the level of the nodes containing G, H, I, and J is 3.

The maximum level in a tree determines its height. The maximum number of nodes at any level N is 2^N. Often, however, levels do not contain the maximum number of nodes. For instance, in Figure 8.2, Level 2 could contain four nodes, but because the node containing C in Level 1 has only one child, Level 2 contains only three nodes. Level 3, which could contain eight nodes, has only four. We could make many differently shaped binary trees out of the ten nodes in this tree. A few variations are illustrated in Figure 8.3. It is easy to see that the maximum number of levels in a binary tree with N nodes is N (counting Level 0 as one of the levels). What is the minimum number of levels? If we fill the tree by giving every node in each level two children until we run out of nodes, the tree has $\log_2 N + 1$ levels (Figure 8.3a). Demonstrate this to yourself by drawing "full" trees with 8 [$\log_2$ (8) = 3] and 16 [$\log_2$ (16) = 4] nodes. What if there are 7, 12, or 18 nodes?

The height of a tree is the critical factor in determining how efficiently we can search for elements. Consider the maximum-height tree in Figure 8.3(c). If we begin searching at the root node and follow the references from one node to the next, accessing the node with the value J (the farthest from the root) is a O(N) operation—no better than searching a linear list! On the other hand, given the minimum-height tree depicted in Figure 8.3(a), to access the node containing J, we only have to look at three other nodes—the ones containing E, A, and G—before we find J. Thus, if the tree is of minimum height, its structure supports O($\log_2 N$) access to any element.

However, the arrangement of the values in the tree pictured in Figure 8.3(a) does not lend itself to quick searching. Let's say that we want to find the value G. We begin searching at the root of the tree. This node contains E, not G, so we need to keep

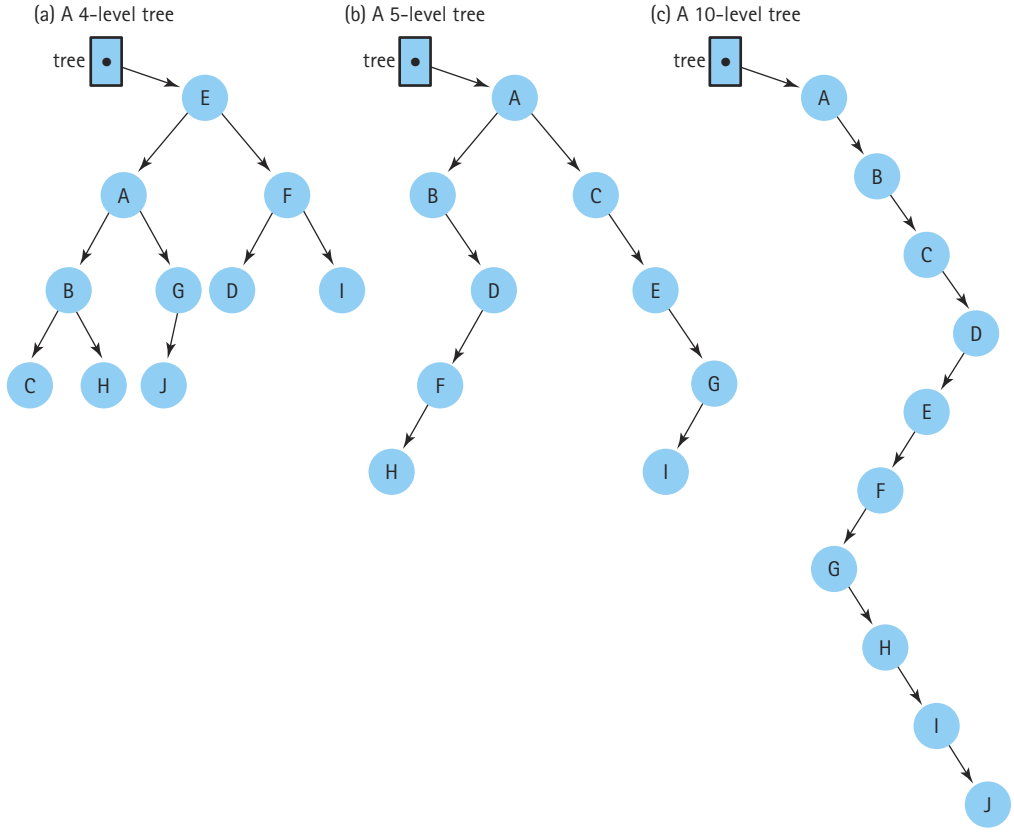


Figure 8.3 *Binary trees*

searching. But which of its children should we look at next, the right or the left? There is no special order to the nodes, so we have to check both subtrees. We could search the tree, level by level, until we come across the value we are searching for. But that is a O(N) search operation, which is no better than searching a linked list!

Binary Search Trees

To support O($\log_2 N$) searching, we add a special property based on the relationship among the keys of the items in the binary tree. We put all the nodes with values smaller than the value in the root into its left subtree, and all the nodes with values larger than the value in the root into its right subtree. Figure 8.4 shows the nodes from Figure 8.3(a) rearranged to satisfy this property. The root node, which contains E, references two sub-

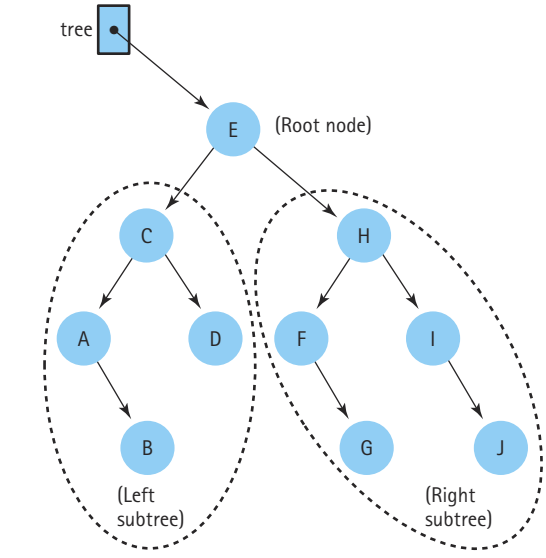


Figure 8.4 *A binary tree*

trees. The left subtree contains all the values smaller than E and the right subtree contains all the values larger than E.

Searching for the value G, we look first in the root node. G is larger than E, so we know that G must be in the root node's right subtree. The right child of the root node contains H. Now what? Do we go to the right or to the left? This subtree is also arranged according to the binary search property: The nodes with smaller values are to the left and the nodes with larger values are to the right. The value of this node, H, is greater than G, so we search to its left. The left child of this node contains the value F, which is smaller than G, so we reapply the rule and move to the right. The node to the right contains G; we have found the node we were searching for.

A binary tree with this special property is called a binary search tree. Like any binary tree, it gets its branching structure by allowing each node to have up to two child nodes. It gets its easy-to-search structure by maintaining the *binary search property*: The left child of any node (if there is one) is the root of a subtree that contains only values smaller

Binary search tree A binary tree in which the key value in any node is greater than the key value in its left child and any of its descendants (the nodes in the left subtree) and less than the key value in its right child and any of its descendants (the nodes in the right subtree)

than the node. The right child of any node (if there is one) is the root of a subtree that contains only values that are larger than the node.

Four comparisons instead of up to ten doesn't sound like such a big deal, but as the number of elements in the structure increases, the difference becomes impressive. In the worst case—searching for the last node in a linear linked list—you must look at every node in the list; on the average you must search half the list. If the list contains 1,000 nodes, you must make 1,000 comparisons to find the last node! If the 1,000 nodes were arranged in a binary search tree of minimum height, you would never make more than 10 comparisons ($\log_2(1000) < 10$), no matter which node you were seeking!

Binary Tree Traversals

To traverse a linear linked list, we set a temporary reference equal to the start of the list and then follow the list references from one node to the other until we reach a node whose reference value is `null`. Similarly, to traverse a binary tree, we initialize our reference to the root of the tree. But where do we go from there—to the left or to the right? Do we visit[1] the root or the leaves first? The answer is "all of these." There are only two standard ways to traverse a list: forward and backward. There are many ways to traverse a tree. We define three common ones in this subsection.

Preorder traversal A systematic way of visiting all the nodes in a binary tree by visiting a node, then visiting the nodes in the left subtree of the node, and then visiting the nodes in the right subtree of the node

Inorder traversal A systematic way of visiting all the nodes in a binary tree by visiting the nodes in the left subtree of a node, then visiting the node, and then visiting the nodes in the right subtree of the node

Postorder traversal A systematic way of visiting all the nodes in a binary tree by visiting the nodes in the left subtree of a node, then visiting the nodes in the right subtree of the node, and then visiting the node

Our traversal definitions depend upon the relative order in which we visit a root and its subtrees. We define three possibilities here:

- Preorder traversal: Visit the root, visit the left subtree, visit the right subtree
- Inorder traversal: Visit the left subtree, visit the root, visit the right subtree
- Postorder traversal: Visit the left subtree, visit the right subtree, visit the root

Notice that the name given to each traversal specifies where the root itself is processed in relation to its subtrees. Also note that these are recursive definitions.

We can visualize each of these traversal orders by drawing a "loop" around a binary tree as in Figure 8.5. Before drawing the loop, extend the nodes of the tree that have less than two children with short lines so that every node has two "edges." Then draw the loop from the root of the tree, down the left subtree, and back up again, hugging the shape of the tree as you go. Each node of the tree is "touched" three times by the loop (the touches are numbered in the figure): once on the way down

[1] When we say "visit," we mean that the algorithm does whatever it needs to do with the values in the node: print them, sum certain values, or delete them, for example. For this section we assume that a visit means to print out the value of the node.

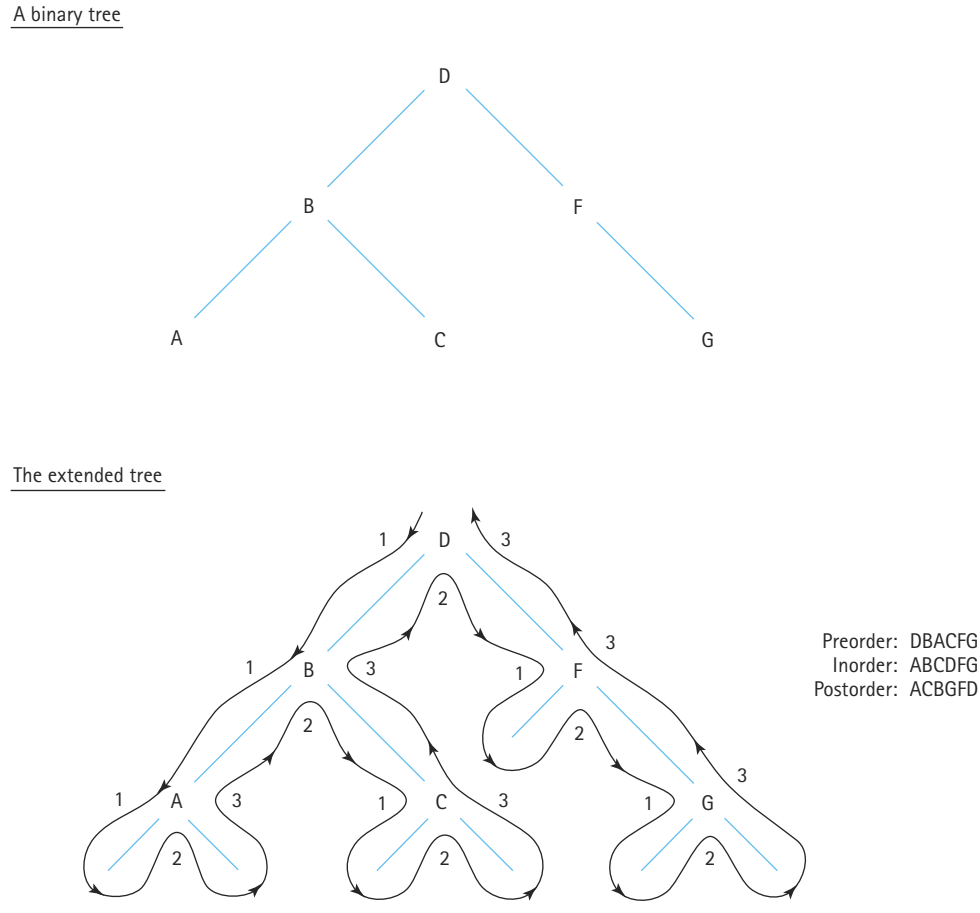
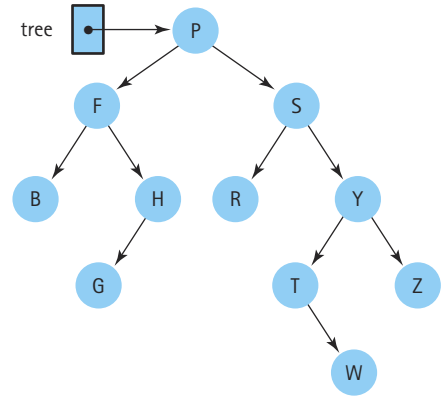


Figure 8.5 *Visualizing binary tree traversals*

before the left subtree is reached, once after finishing the left subtree but before starting the right subtree, and once on the way up, after finishing the right subtree. To generate a preorder traversal, follow the loop and visit each node the first time it is touched (before visiting the left subtree). To generate an inorder traversal, follow the loop and visit each node the second time it is touched (in between visiting the two subtrees). To generate a postorder traversal, follow the loop and visit each node the third time it is touched (after visiting the right subtree). Use this method on the tree in Figure 8.6 and see if you agree with the listed traversal orders.

You may have noticed that an inorder traversal of a binary search tree visits the nodes in order from the smallest to the largest. Obviously, this would be useful when we



Inorder: B F G H P R S T W Y Z
Preorder: P F B H G S R Y T W Z
Postorder: B G H F R W T Z Y S P

Figure 8.6 *Three binary tree traversals.*

need to access the elements in ascending key order; for example, to print a sorted list of the elements. There are also useful applications of the other traversal orders. For example, the preorder and postorder traversals can be used to translate infix arithmetic expressions into their prefix and postfix counterparts.

8.2 The Logical Level

In this section, we specify our Binary Search Tree ADT. As we have done for lists, stacks, and queues, we use the Java *interface* construct to specify our ADT. However, before proceeding with the specification, we have to decide what kinds of elements we are going to store on our trees.

The Comparable Interface

In order to define public operations on binary search trees, we must know what kinds of elements make up the trees. Remember that these elements are parameters to many of our operations. For example, we cannot specify a tree `insert` operation without including the type of the element being inserted. Let's review some of our previous ADTs and the types of elements used with them:

- In Chapter 3 we started with an Unsorted List ADT that used strings as elements. We needed to be able to copy list elements, since we stored the elements "by copy." We needed to be able to compare list elements; for example, to support the `isThere` operation. Since we already were able to copy and compare strings, we could use them without any special specifications.

- Later in Chapter 3 we decided to implement a list of more generic elements. However, we still needed to copy and compare our list elements. Therefore, we created the `Listable` interface, consisting of the abstract methods `compareTo` and `copy`, and insisted that our list elements implement the `Listable` interface. We had lists of `Listable` elements.
- In Chapter 4 we defined and implemented Stack and Queue ADTs. We implemented these structures "by reference" rather than "by copy," eliminating the need for elements that support the copy operation. Additionally, since we never search stacks or queues for elements based on a key, we did not need our elements to support a compare operation. So, we simply defined and implemented our stacks and queues to work directly with Java objects (of class `object`).

What about trees? As we did for stacks and queues, we implement our trees "by reference." Recall that this means that a client who uses our tree ADT must be careful not to change the key value of a tree node after it has been inserted into the tree. If a client does, the structure of the tree could be corrupted.

Our decision to use the "by reference" approach means we do not need our tree elements to support a copy operation. How about the compare operation? If we are going to use trees in ways similar to the way we use lists, as implied in the introduction to this chapter, we certainly need to support the comparison of tree elements. Besides, if we are building binary search trees, we need to compare the current tree elements to the elements we are inserting in order to determine insertion locations that are consistent with the tree's binary search property.

So, we do not need elements to support a copy operation, but we do need them to support comparison. As we did for lists, when we defined the `Listable` interface, we could define our own interface (Treeable?!) that consists of just the abstract method `compareTo`, and insist that our tree elements implement this interface. However, we do not need to do that. The Java library already provides such an interface. It is called the `Comparable` interface. It consists of exactly one abstract method:

```
public int compareTo(Object o);
// Returns a negative integer, zero, or a positive integer as this object
//  is less than, equal to, or greater than the parameter object
```

This definition is consistent with the way we have been using `compareTo` throughout the textbook. In fact, you may recall that in Chapter 3, when we first defined `compareTo`, we stated, "we follow the Java convention used in the `String` class by naming the method `compareTo` and by having it return integer values to indicate the result of the comparison." In actuality, the definition of `compareTo` in Java's `String` class is based on the `compareTo` specification in the `Comparable` interface. The `String` class is just one of many Java classes that implement the `Comparable` interface, and therefore provide a concrete implementation of its abstract method. This highlights one of the benefits of using a predefined interface for our tree elements—our trees are able to store

any class that already implements the `Comparable` interface. For example, our trees are able to store `String` and `Integer` objects.

The Binary Search Tree Specification

Our binary search tree specification is very similar to our sorted list specification. This is not surprising, since both are typically used to store and retrieve sorted data. We now know that our binary search trees hold `Comparable` elements and store references to the elements, rather than copies of the elements. Are there other decisions to make before we can formally specify our ADT?

First, we must decide if duplicate elements are allowed in our trees. In one sense, we have already made this decision. Our definition of a binary search tree precludes duplicate elements. Do you see why? If two identical elements are in different subtrees, then the root of the subtrees could not possibly be greater than one of the elements and less than the other. Therefore, they must be in the same subtree. However, that means that one of them must be a descendant of the other. The definition of a binary search tree does not allow the key value of a node to be equal to the key value of one of its descendants. Therefore, they cannot be in the same subtree. Clearly, this is an impossible situation.

Of course, we could change the definition of binary search tree to state that the key value of a node is greater than "or equal to" the key values of the nodes in its left subtree. But there is no need to do this; instead, we do assume that our trees consist of unique elements. This is consistent with our list approach.

We also follow the conventions established with our list approach in terms of operation preconditions. For example, we assume that when the `retrieve` operation is invoked that an element in the tree has a key value that matches the key value of the parameter item. As with lists, alternative approaches are possible, and in some cases preferable. But we want to concentrate here on the tree operations themselves.

In the previous section we defined three binary tree traversals. Which one should we use to iterate through our tree? Why not support all three? We define the `reset` and `getNextItem` operations with a parameter that indicates which of the three traversals to use. As we did with our Specialized List in Chapter 6, we allow more than one traversal to be in progress at a time. Within our interface definition we define the three constants:

```
public static final int INORDER = 1;
public static final int PREORDER = 2;
public static final int POSTORDER = 3;
```

for use as parameters to `reset` and `getNextItem`. These constants are available to any class that implements the interface. Their use is demonstrated in the next section.

We make one other modification to the definition of the `reset` operation, as compared to the `reset` operation for lists. The Binary Search Tree `reset` method, in addition to setting up an iteration, returns the current number of nodes in the tree. We explain the reason for this change in the subsection that discusses the implementation of the iteration methods.

Here is the specification of our Binary Search Tree ADT.

```
//--------------------------------------------------------------------------
// BSTInterface.java           by Dale/Joyce/Weems                 Chapter 8
//
// Interface for a class that implements a Binary Search Tree of unique
//  elements, i.e., no duplicate elements as defined by the key of the tree
//  elements
//--------------------------------------------------------------------------

package ch08.trees;

public interface BSTInterface
{
  // Used to specify traversal order
  public static final int INORDER = 1;
  public static final int PREORDER = 2;
  public static final int POSTORDER = 3;

  public boolean isEmpty();
  // Effect:         Determines whether this BST is empty
  // Postcondition:  Return value = (this BST is empty)

  public boolean isFull();
  // Effect:         Determines whether this BST is full
  // Postcondition:  Return value = (this BST is full)

  public int numberOfNodes();
  // Effect:         Determines the number of nodes in this BST
  // Postcondition:  Return value = number of nodes in this BST

  public Comparable isThere (Comparable item);
  // Effect:         Determines whether element matching item is in this BST
  // Postcondition:  Return value = (element with the same key as item is in
  //                 this BST)

  public Comparable retrieve(Comparable item);
  // Effect:         Returns the BST element with the same key as item
  // Precondition:   An element with the same key as item is in this BST
  // Postcondition:  Return value = (reference to BST element that matches
  //                 item)

  public void insert (Comparable item);
  // Effect:         Adds item to this BST
  // Preconditions:  This BST is not full
```

```
  //                  Element matching item is not in this BST
  // Postcondition:   Item is in this BST

  public void delete (Comparable item);
  // Effect:          Delete the element of this BST whose key matches item's
  //                  key
  // Precondition:    Exactly one element in this BST has a key matching item's
  //                  key
  // Postcondition:   No element has a key matching the argument item's key

  public int reset(int orderType);
  // Effect:          Initializes current position for an iteration through this
  //                  BST
  //                  in orderType order
  // Postconditions:  Current position is first element for the orderType order
  //                  Returns current number of nodes in the tree

  public Comparable getNextItem (int orderType);
  // Effect:          Returns a copy of the element at the current position in
  //                  this BST and advances the value of the current position
  //                  based on the orderType
  // Preconditions:   Current position for this orderType is defined
  //                  There exists a BST element at current position
  //                  No BST transformers were called since most recent call to
  //                  reset
  // Postcondition:   Return value = (a copy of element at current position)
}
```

8.3 The Application Level

As we have already pointed out, our Binary Search Tree ADT is very similar to our Sorted List ADT. A comparison of the `BSTInterface` listed in the previous section to the `ListInterface` presented in Section 4.1 only reveals a few minor changes. Instead of a `length` operation, we have a `numberOfNodes` operation. (The number of nodes in a tree is analogous to the length of a list.) In addition, we support three iteration paths through the tree rather than just one, as we do with the list. The biggest difference between our lists and our trees is that our lists use copies of the client's information, whereas our trees use references to the client's information. This difference is a problem only if the client updates information related to an item's key after inserting an item into the structure.

Another important difference between our sorted lists and our binary search trees is in the efficiency of some of the operations; we highlight these differences later in this chapter. The similarity between our ADTs means that we can use the binary search tree in many of the same applications where we use lists.

For example, we used the Sorted List ADT in the Real Estate program for the Chapter 3 case study. In place of the `SortedList` class we could use a class that implements the `BSTInterface`. For the Real Estate program, the differences between storing copies of elements and references to elements do not affect anything. The application does not support a "change house information" operation. The only way for the user to change the information about a house is to delete the house from the list and then reinsert it with the new information in place. This works equally well whether the information is stored by copy or by reference. Of course, at those places in the program where the list iterator is used, the appropriate tree iterator would have to be chosen.

We look next at an example of how a client performs tree iteration.

A printTree Operation

As with our lists, we have not included a print operation for our binary search trees. And the reason is the same. We don't include it because to write a good print routine, we must know what the elements represent. The application programmer (who does know what the elements represent) can use the `reset` and `getNextItem` operations to iterate through the tree, printing each element in a form that makes sense within the application. In the code that follows, we assume our tree elements are strings and the desired form of output is a simple numbered list of elements. We use the inorder traversal, so the values are printed from smallest to largest. Finally, we assume that the class `BinarySearchTree` implements the `BSTInterface`.

```
void printTree(PrintWriter outFile, BinarySearchTree tree)
// Effect:         Prints contents of tree to outFile
// Preconditions:  Tree has been instantiated
//                 OutFile is open for writing
// Postconditions: Each component in tree has been written to outFile
//                 OutFile is still open
{
  String theString;
  int treeSize;
  treeSize = tree.reset(BinarySearchTree.INORDER);
  outFile.println("The tree elements in Inorder order are:");
  for (int count = 1; count <= treeSize; count++)
  {
    theString = (String) tree.getNextItem(BinarySearchTree.INORDER);
    outFile.println(count + ". " + theString);
  }
}
```

Note the use of the constant `INORDER` as a parameter to the `reset` and `getNextItem` methods. It is defined in the `BSTInterface` interface. We access the constant through the `BinarySearchTree` class, which implements the interface. Also note how we make use of the new functionality of the `reset` operation; it tells us the number of

nodes in the tree and we use that value to control the number of iterations through the *for* loop.

8.4 The Implementation Level—Declarations and Simple Operations

We represent a tree as a linked structure whose nodes are allocated dynamically. Before we go on, we need to decide just what a node in the tree is going to look like. In our discussion of binary trees, we talked about right and left children. These are the structural references in the tree; they hold the tree together. We also need a place to store the user's data in the node. We might as well continue to call it `info`. Figure 8.7 shows a picture of a node.

Here is the definition of `BSTNode` that corresponds to the picture in Figure 8.7:

```
protected class BSTNode
{
  // Used to hold references to BST nodes for the linked implementation
  protected Comparable info;       // The info in a BST node
  protected BSTNode left;          // A link to the left child node
  protected BSTNode right;         // A link to the right child node
}
```

Notice the type of information held by a tree node: `Comparable`. Our trees can hold any kind of object, as long as it is of a class that implements `Comparable`.

We declare `BSTNode` as an inner class of our Binary Search Tree implementation. In this way, it is available to all of the methods of the implementation, but it is hidden from the client programmer, as it should be.

We call our implementation class `BinarySearchTree`. It implements the `BSTInterface`. The instance variable `root` references the root node of the tree. It is set to `null` by the constructor. The beginning of the class definition looks like this:

```
//-------------------------------------------------------------------------
// BinarySearchTree.java          by Dale/Joyce/Weems                Chapter 8
//
// Defines all constructs for a reference-based BST
//-------------------------------------------------------------------------
```

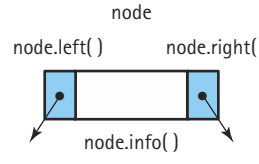


Figure 8.7 *Node terminology for a binary tree node*

```
package ch08.trees;

import ch04.queues.*;
import ch05.stacks.*;
public class BinarySearchTree implements BSTInterface
{
  protected class BSTNode
  {
    // Used to hold references to BST nodes for the linked implementation
    protected Comparable info;       // The info in a BST node
    protected BSTNode left;          // A link to the left child node
    protected BSTNode right;         // A link to the right child node
  }

  protected BSTNode root;            // Reference to the root of this BST

  public BinarySearchTree()
  // Creates an empty BST object
  {
    root = null;
  }
  . . .
}
```

Next let's look at the simple observer methods `isFull` and `isEmpty`. As we did for our dynamically allocated lists, stacks, and queues, we implement `isFull` to always return `false`. We include `isFull` in the `BSTInterface`, because an implementation that depends upon static allocation of node space may become full. However, with the dynamic allocation approach we use here, our tree is not full unless we completely run out of memory. In that case, the Java system throws an exception anyway. So we ignore the possibility and assume that our tree is never full.

```
public boolean isFull()
// Determines whether this BST is full
{
  return false;
}
```

The `isEmpty` operation is different from its list counterpart. We are not keeping a running count of the number of nodes. Therefore, we cannot simply check an instance variable for the current number of nodes for this operation. We could use the `numberOfNodes` method—if it returns a 0, `isEmpty` returns `true`; otherwise, it returns `false`. But the `numberOfNodes` method has to count the nodes on the tree each time it is called. This takes at least O(N) steps, where N is the number of nodes. Is there a

cheaper way to see if the list is empty? Yes, just see whether the root of the tree is currently `null`. This only takes O(1) steps.

```
public boolean isEmpty()
// Determines whether this BST is empty
{
  return (root == null);
}
```

We next look at methods that are more complicated. We start with the `numberOfNodes` method and use it to investigate the differences between iterative and recursive approaches.

8.5 Iterative Versus Recursive Method Implementations

Binary search trees provide a good chance to compare iterative and recursive approaches to a problem. Notice that trees are inherently recursive. Any node in a tree can be considered the root of a subtree. We even use recursive definitions when talking about properties of trees; for example: "a node is the *ancestor* of another node if it is the parent of the node, or the parent of some other *ancestor* of that node." And, of course, the formal definition of a binary tree node, embodied in the class `BSTNode`, is itself recursive. So it seems probable that recursive solutions work well when dealing with trees. In this section we address that hypothesis.

First, we develop recursive and iterative implementations of the `numberOfNodes` method. Of course, the method could be implemented by maintaining a running count of tree nodes (incrementing it for every `insert` and decrementing it for every `delete`). We used that approach for lists. The alternate approach of counting the nodes each time the number is needed is also viable, and we use it here.

After we look at the two implementations of the `numberOfNodes` method, we discuss the benefits of recursion versus iteration for this problem.

Recursive numberOfNodes

As was the case with the recursive linked list methods we developed in Chapter 7, we must use a public method to access the Number of Nodes operation, and a private recursive method to do all the work. The recursive method requires a reference to a tree node as a parameter; since tree nodes are hidden from the client, the client cannot directly invoke the recursive method. Thus the use of the public/private pattern.

The public method, `numberOfNodes`, calls the private recursive method, `recNumberOfNodes` and passes it a reference to the root of the tree. We design the recursive method to return the number of nodes in the subtree referenced by the argument passed to it. Therefore, it returns the number of nodes in the entire tree to the `numberOfNodes` method, which in turn returns it to the client program. The code for `numberOfNodes` is, of course, very simple:

```
public int numberOfNodes()
// Determines the number of nodes in this BST
{
  return recNumberOfNodes(root);
}
```

Remember for function Factorial we said that we could determine the factorial of N if we knew the factorial of $N - 1$. The analogous statement here is that we can determine the number of nodes in the tree if we know the number of nodes in its left subtree and the number of nodes in its right subtree. That is, the number of nodes in a tree is:

number of nodes in left subtree + *number of nodes in right subtree* + 1

This is easy. Given a method `recNumberOfNodes` and a reference to a tree node, we know how to calculate the number of nodes in a subtree: We call `recNumberOfNodes` recursively with the reference to the subtree as the argument. Thus we know how to write the general case. What about the base case? Well, a leaf node has no subtrees, so the number of nodes is 1. How do we determine that a node has no subtrees? The references to its children are `null`. Let's try summarizing these observations into an algorithm, where `tree` is a reference to a node.

recNumberOfNodes(tree): returns int *Version 1*

```
if (tree.left( ) is null) AND (tree.right( ) is null)
   return 1
else
   return recNumberOfNodes(tree.left( )) + recNumberOfNodes(tree.right( )) + 1
```

Let's try this algorithm on a couple of examples to be sure that it works (see Figure 8.8).

We call `recNumberOfNodes` with the tree in Figure 8.8(a). The left and right children of the root node (M) are not `null`, so we call `recNumberOfNodes` with the node containing A as the root. Because both the left and right children are `null` on this call, we send back the answer 1. Now we call `recNumberOfNodes` with the tree containing Q as the root. Both of its children are `null`, so we send back the answer 1. Now we can calculate the number of nodes in the tree with M in the root: it is

1 + 1 + 1 = 3

This seems okay.

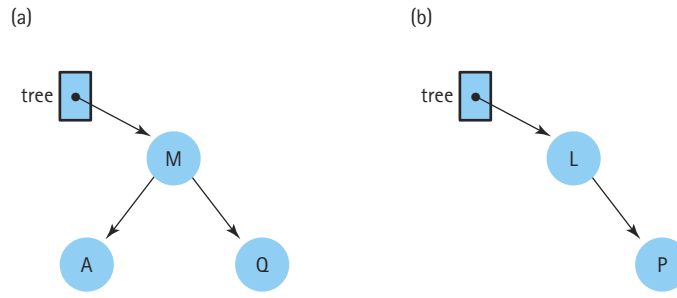


Figure 8.8 *Two binary search trees*

The left subtree of the root of the tree in Figure 8.8(b) is empty; let's see if this proves to be a problem. It is not true that both children of the root (L) are `null`, so `recNumberOfNodes` is called with the left child as the argument. OOPS! We do have a problem. The left child of L is `null`, so we just called `recNumberOfNodes` with a `null` parameter. The first statement checks to see if the children of the tree referenced by the argument are `null`, but the value of tree itself is `null`. The method crashes when we try to access `tree.left` when `tree` is `null`. To prevent this, we can check to see if a child is `null`, and not call `recNumberOfNodes` if it is.

recNumberOfNodes(tree): returns int *Version 2*

```
if (tree.left( ) is null) AND (tree.right( ) is null)
      return 1
else if tree.left( ) is null
      return recNumberOfNodes(tree.right( )) + 1
else if tree.right( ) is null
      return recNumberOfNodes(tree.left( )) + 1
else return recNumberOfNodes(tree.left( )) + recNumberOfNodes(tree.right( )) + 1
```

Version 2 works correctly if method `recNumberOfNodes` has a precondition that the tree is not empty. However, an initially empty tree causes a crash. We must check to see if the tree is empty as the first statement in the algorithm, and if it is, return zero.

recNumberOfNodes(tree): returns int *Version 3*

```
if tree is null
      return 0
else if (tree.left( ) is null) AND (tree.right( ) is null)
      return 1
else if tree.left( ) is null
      return recNumberOfNodes(tree.right( )) + 1
else if tree.right( ) is null
      return recNumberOfNodes(tree.left( )) + 1
else return recNumberOfNodes(tree.left( )) + recNumberOfNodes(tree.right( )) + 1
```

This certainly looks complicated; there must be a simpler solution—and there is. We can collapse the two base cases into one. There is no need to make the leaf node a special case. We can simply have one base case: an empty tree returns zero. Now we do not have to check the left and right subtrees; if they are `null` and we process them, they just contribute a value of zero.

recNumberOfNodes(tree): returns int *Version 4*

```
if tree is null
      return 0
else
      return recNumberOfNodes(tree.left( )) + recNumberOfNodes(tree.right( )) + 1
```

We have taken the time to work through the versions containing errors because they illustrate two important points about recursion with trees: (1) Always check for the empty tree first, and (2) leaf nodes do not need to be treated as separate cases. Table 8.1 reviews the design notation and the corresponding Java code.

Table 8.1 *Comparing Node Design Notation to Java Code*

Design Notation	Java Code
location.info( )	location.info
location.right( )	location.right
location.left( )	location.left
Set location.info( ) to value	location.info = value

Here is the method specification:

Method recNumberOfNodes(tree)

Definition:	Counts and returns the number of nodes in tree
Size:	Number of nodes in tree
Base Case:	If tree is null, return 0
General Case:	Return recNumberOfNodes(tree.left) + recNumberOfNodes(tree.right) + 1

And the code:

```
private int recNumberOfNodes(BSTNode tree)
// Determines the number of nodes in tree
{
  if (tree == null)
    return 0;
  else
    return recNumberOfNodes(tree.left) +
           recNumberOfNodes(tree.right) + 1;
}
```

Iterative numberOfNodes

An iterative method to count the nodes on a linked list is easy:

```
count = 0;
while (list != null)
{
  count++;
  list = list.next;
}
return count;
```

A similar approach for an iterative method to count the nodes in a binary tree quickly runs into trouble. We start at the root and increment the count. Now what? Should we count the nodes in the left subtree or the right subtree? Suppose we decide to

count the nodes in the left subtree. We must remember to come back later and count the nodes in the right subtree. In fact, every time we make a decision on which subtree to count we must remember to return to that node and count the nodes of its other subtree. How can we remember all of this?

In the recursive version, we did not have to explicitly remember which subtrees we still needed to process. The trail of unfinished business was maintained on the system stack for us automatically. For the iterative version, we must maintain the information explicitly, on our own stack. Whenever we postpone processing a subtree, we can push a reference to that subtree on a stack of references. Then, when we are finished with our current processing, we can remove the reference that is on the top of the stack and continue our processing with it.

We must be careful that we process each node in the tree exactly once. To ensure that we do not process a node twice, we follow these rules:

1. Process a node immediately after removing it from the stack.
2. Do not process nodes at any other time.
3. Once a node is removed from the stack, do not push it back onto the stack.

To ensure that we do not miss any nodes, we begin execution by pushing the root onto the stack. As part of the processing of every node, we push its children onto the stack. This guarantees that all descendants of the root are eventually pushed onto the stack; in other words, that we do not miss any nodes.

Finally, we only push references to actual tree nodes. We do not push any `null` references. This way, when we remove a reference from the stack we can increment the count of nodes and access the left and right links of the referenced node without worrying about null reference errors. Here is an algorithm for the iterative `numberOfNodes`:

numberOfNodes returns int

```
Set count to 0
if the tree is not empty
    Instantiate a stack
    Push the root of the tree onto the stack
    while the stack is not empty
        Set currNode to top of stack
        Pop the stack
        Increment count
        if currNode has a left child
            Push currNode's left child onto the stack
        if currNode has a right child
            Push currNode's right child onto the stack
return count
```

The corresponding code, using a stack named `hold`, is as follows:

```
public int numberOfNodes()
// Determines the number of nodes in this BST
{
  int count = 0;
  if (root != null)
  {
    LinkedStack hold = new LinkedStack();
    BSTNode currNode;
    hold.push(root);
    while (!hold.isEmpty())
    {
      currNode = (BSTNode) hold.top();
      hold.pop();
      count++;
      if (currNode.left != null)
        hold.push(currNode.left);
      if (currNode.right != null)
        hold.push(currNode.right);
    }
  }
  return count;
}
```


Recursion or Iteration?

Now that we have looked at both the recursive and the iterative versions of counting nodes, can we determine which is better? In the last chapter, we gave some guidelines for determining when recursion is appropriate. Let's apply these to the use of recursion for counting nodes.

Is the depth of recursion relatively shallow?

Yes. The depth of recursion is dependent on the height of the tree. If the tree is well-balanced (relatively short and bushy, not tall and stringy), the depth of recursion is closer to $O(\log_2 N)$ than to $O(N)$.

Is the recursive solution shorter or clearer than the nonrecursive version?

Yes. The recursive solution is shorter than the iterative method, especially if you count the code for implementing the stack against the iterative approach. Is the recursive solution clearer? Although we spent more space discussing the recursive solution, we do believe it is clearer. We used extra space in order to teach you a little more about recursive design. We believe the recursive version is intuitively obvious. It is very easy to see

that the number of nodes in a binary tree that has a root is 1 plus the number of nodes in its two subtrees. The iterative version is not as clear. We need to worry that we did not count any node twice, and that we did not miss any nodes. Compare the code for the two approaches and see what you think.

Is the recursive version much less efficient than the nonrecursive version?

No. Both the recursive and the nonrecursive versions of `numberOfNodes` are O(N) operations. They both have to count every node.

We give the recursive version of the method an 'A'; it is a good use of recursion.

8.6 The Implementation Level—More Operations

In this section, we use recursion to implement the remaining Binary Search Tree operations.

The isThere and retrieve Operations

At the beginning of this chapter, we demonstrated how to search for an element in a binary search tree. First check to see if the item searched for is in the root. If it is not, compare the element with the root and look in either the left or the right subtree. This is a recursive algorithm.

We implement `isThere` using a private recursive method called `recIsThere`. This method is passed the item we are searching for and a reference to a subtree in which to search. It follows the algorithm described above in a straightforward manner. The only remaining question is how to determine that there is no item with the same key in the tree. If the subtree we are searching is empty, then there cannot be an item with the same key as the item's key. We summarize these observations in a table, which is followed by the code.

Method recIsThere (item, tree)

Definition:	Searches for an element in tree with the same key as item's key If found, return true; otherwise, return false
Size:	Number of nodes in tree (or number of nodes in the path)
Base Cases:	(1) If item's key matches key in tree.info( ), return true. (2) If tree = null, return false.
General Case:	If item's key is less than key in tree.info( ), return recIsThere(item, tree.left( )); else return recIsThere(item, tree.right( ))

```
private boolean recIsThere(Comparable item, BSTNode tree)
// Returns true if item is in tree; false otherwise
{
  if (tree == null)
    return false;                              // Item is not found
  else if (item.compareTo(tree.info) < 0)
    return recIsThere(item, tree.left);        // Search left subtree
  else if (item.compareTo(tree.info) > 0)
    return recIsThere(item, tree.right);       // Search right subtree
  else
    return true;                               // Item is found
}

public boolean isThere (Comparable item)
// Determines if element matching item is in this BST
{
  return recIsThere(item, root);
}
```

Let's trace this operation using the tree in Figure 8.9. In our trace we substitute actual arguments for the method parameters. We assume we can work with integers. We want to search for the element with the key 18 in a tree `myTree`, so the call to the public method is

```
myTree.isThere(18);
```

The `isThere` method, in turn, immediately calls the recursive method:

```
return recIsThere(18, root);
```

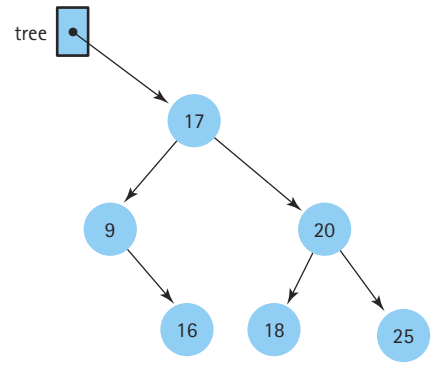


Figure 8.9 *Tracing the `isThere` operation*

Since `root` is not `null`, and `18 > tree.info`, we issue the first recursive call:

```
return recIsThere(18, tree.right);
```

Now `tree` references the node whose key is 20 so `18 < tree.info`. The next recursive call is

```
return recIsThere(18, tree.left);
```

Now `tree` references the node with the key 18 so processing falls through to the last *else* statement

```
return true;
```

This halts the recursive descent, and the value `true` is passed back up the line of recursive calls until it is returned to the original `isThere` method and then to the client program.

Next, let's look at an example where the key is not found in the tree. We want to find the element with the key 7. The public method call is

```
myTree.isThere(7);
```

followed immediately by

```
recIsThere(7, root)
```

`tree` is not `null` and `7 < tree.info`, so the first recursive call is

```
recIsThere(7, tree.left)
```

`tree` is pointing to the node that contains 9. `tree` is not `null` and we issue the second recursive call

```
recIsThere(7, tree.left)
```

Now tree is `null`, and the return value of `false` makes its way back to the original caller.

The `retrieve` method is very similar to the `isThere` operation. In both cases we search the tree recursively to locate the tree element that matches the parameter `item`. There are two differences however. First, the precondition of `retrieve` states that the item being retrieved is definitely in the tree, so we do not have to worry about the base case of reaching a `null` subtree. Second, instead of returning a `boolean` we must return a reference to the tree element that matches `item`. Remember that the actual tree element is the `info` of the tree node, so we must return a reference to the `info` object. The `info` variable references an object of type `Comparable`.

```
private Comparable recRetrieve(Comparable item, BSTNode tree)
// Returns the element in tree with the same key as item
{
  if (item.compareTo(tree.info) < 0)
    return recRetrieve(item, tree.left);        // Retrieve from left subtree
  else
  if (item.compareTo(tree.info) > 0)
    return recRetrieve(item, tree.right);       // Retrieve from right subtree
  else
    return tree.info;
}

public Comparable retrieve(Comparable item)
// Returns the BST element with the same key as item
{
  return recRetrieve(item, root);
}
```

The insert Operation

To create and maintain the information stored in a binary search tree, we must have an operation that inserts new nodes into the tree. We use the following insertion approach. A new node is always inserted into its appropriate position in the tree as a leaf. Figure 8.10 shows a series of insertions into a binary tree.

We use a public method, `insert`, that is passed the `item` for insertion. The `insert` method invokes the recursive method, `recInsert`, and passes it the `item` plus a reference to the `root` of the tree.

```
public void insert (Comparable item)
// Adds item to this BST
{
  root = recInsert(item, root);
}
```

Note that the call to `recInsert` returns a `BSTNode`. It returns a reference to the new tree, that is, to the tree that includes `item`. The statement

```
root = recInsert(item, root);
```

can be interpreted as "Set the reference of the root of this tree to the root of the tree that is generated when item is inserted into this tree." At first this might seem inefficient. Since we always perform insertions as leaves, why do we have to change the root of the tree? Look again at the sequence of insertions in Figure 8.10. Do any of the insertions affect the value of the root of the tree? Yes, the original insertion into the empty tree changes the value held

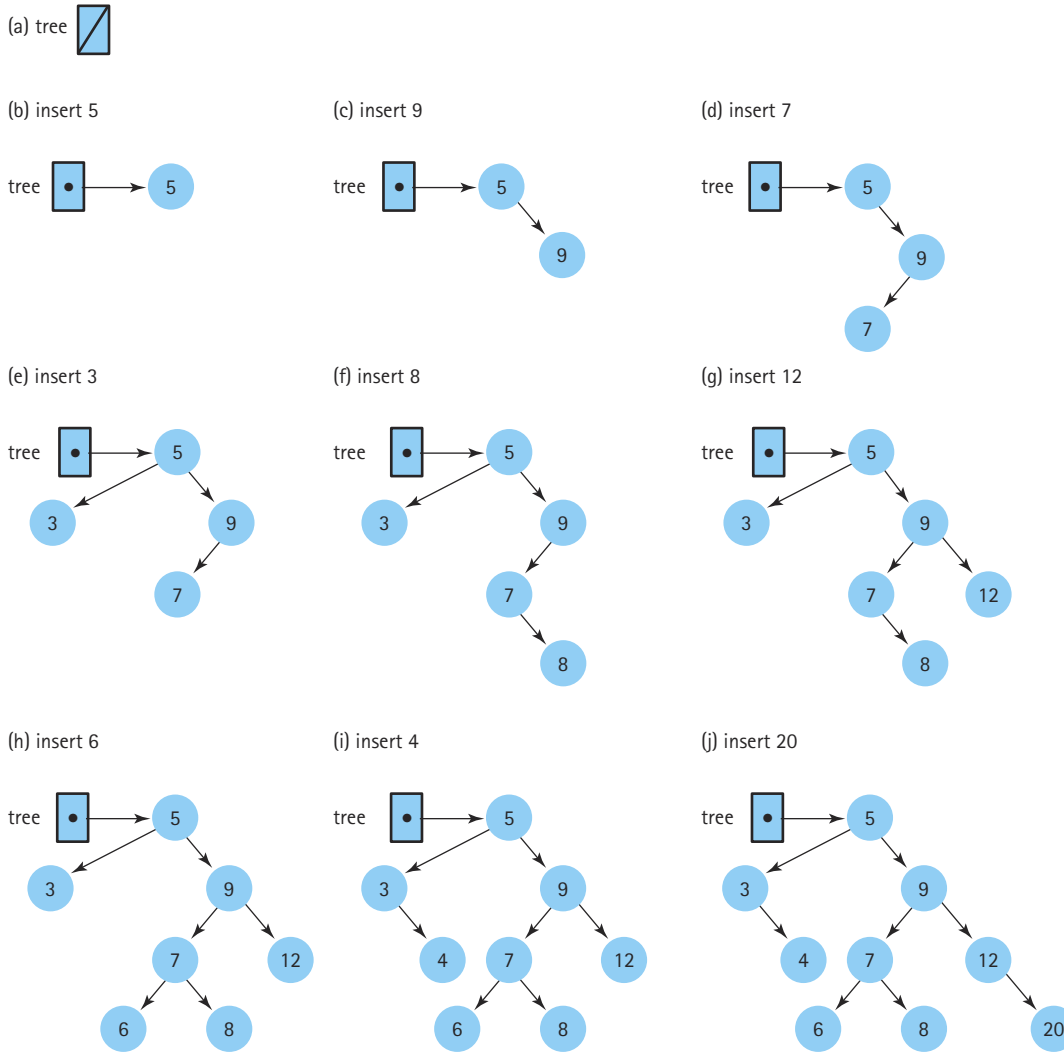


Figure 8.10 *Insertions into a binary search tree*

in root. In the case of all the other insertions the statement in the `insert` method just copies the current value of root onto itself; but we still need the assignment statement to handle the degenerative case of insertion into an empty tree. When does the assignment statement occur? After all the recursive calls to `recInsert` have been processed and have returned.

Before we go into the development of `recInsert`, we want to reiterate that every node in a binary search tree is the root node of a binary search tree. In Figure 8.11(a), we want to insert a node with the key value 13 into our tree whose root is the node

Figure 8.11 *The recursive* `insert` *operation*

containing 7. Because 13 is greater than 7, we know that the new node belongs in the root node's right subtree. We now have redefined a smaller version of our original problem. We want to insert a node with the key value 13 into the tree whose root is `tree.right` (Figure 8.11b). Of course, we have a method to insert elements into a binary search tree: `recInsert`. The `recInsert` method is called recursively:

```
tree.right = recInsert(item, tree.right);
```

`recInsert` still returns a reference to a `BSTNode`; it is the same `recInsert` method that was originally called from `insert` so it must behave in the same way. The above statement says "Set the reference of the right subtree of the tree to the root of the tree that is generated when item is inserted into the right subtree of tree." Again, the actual assignment statement does not occur until after the remaining recursive calls to `recInsert` have finished processing and have returned.

`recInsert` begins its execution, looking for the place to insert item in the tree whose root is the node with the key value 15. We compare the key of `item` (13) to the key of the root node; 13 is less than 15, so we know that the new item belongs in the tree's left subtree. Again, we have obtained a smaller version of the problem. We want to insert a node with the key value 13 into the tree whose root is `tree.left` (Figure 8.11c). We call `recInsert` recursively to perform this task. Remember that in this (recursive) execution of `recInsert`, `tree` points to the node whose key is 15, not the original tree root:

```
tree.left = recInsert(item, tree.left);
```

Again we recursively execute `recInsert`. We compare the key of `item` to the key of the (current) root node and then call `recInsert` to insert `item` into the correct subtree—the left subtree if `item`'s key is less than the key of the root node, the right subtree if `item`'s key is greater than the key of the root node.

Where does it all end? There must be a base case, in which space for the new element is allocated and the value of `item` copied into it. This case occurs when `tree` is `null`, that is, when the subtree we wish to insert into is empty. (Remember, we are going to add `item` as a leaf node.) Figure 8.11(d) illustrates the base case. We create the new node and return a reference to it to the most recent invocation of `recInsert` where the reference is assigned to the `right` link of the node containing 10 (see Figure 8.11e). That invocation of `recInsert` is then finished, and it returns a reference to its subtree to the previous invocation (see Figure 8.11f) where the reference is assigned to the `left` link of the node containing 15. And so on; we continue until a reference to the entire tree is returned to the original `insert` method, which assigns it to `root`, as shown in Figure 8.11(g, h).

Note that while backing out of the recursive calls, the only assignment statement that actually changes a value is the one at the deepest nested level; the one that changes the right subtree of the node containing 10 from `null` to a reference to the new node. All of the other assignment statements simply assign a reference to the variable that held that

reference previously. This is a typical recursive approach. We do not know ahead of time at what level the crucial assignment takes place, so we perform the assignment at every level.

This technique should sound familiar. We used it in the last chapter for the recursive version of insertion into a sorted linked list. The recursive method for insertion into a binary search tree is summarized as follows:

Method recInsert(item, tree) returns tree reference

Definition:	Inserts item into binary search tree.
Size:	The number of elements in path from root to insertion place.
Base Case:	If tree is null, then allocate a new leaf to contain item.
General Cases:	(1) If item < tree.info(), then recInsert(item, tree.left())
	(2) If item > tree.info(), then recInsert(item, tree.right())

Here is the code that implements this recursive algorithm.

```
private BSTNode recInsert(Comparable item, BSTNode tree)
// Inserts item into the tree
{
  if (tree == null)
  {// Insertion place found
    tree = new BSTNode();
    tree.right = null;
    tree.left = null;
    tree.info = item;
  }
  else if (item.compareTo(tree.info) < 0)
    tree.left = recInsert(item, tree.left);     // Insert in left
                                                //  subtree
  else
    tree.right = recInsert(item, tree.right);    // Insert in right
                                                 //  subtree
  return tree;
}
```

Insertion Order and Tree Shape

Because nodes are always added as leaves, the order in which nodes are inserted determines the shape of the tree. Figure 8.12 illustrates how the same data, inserted in differ-

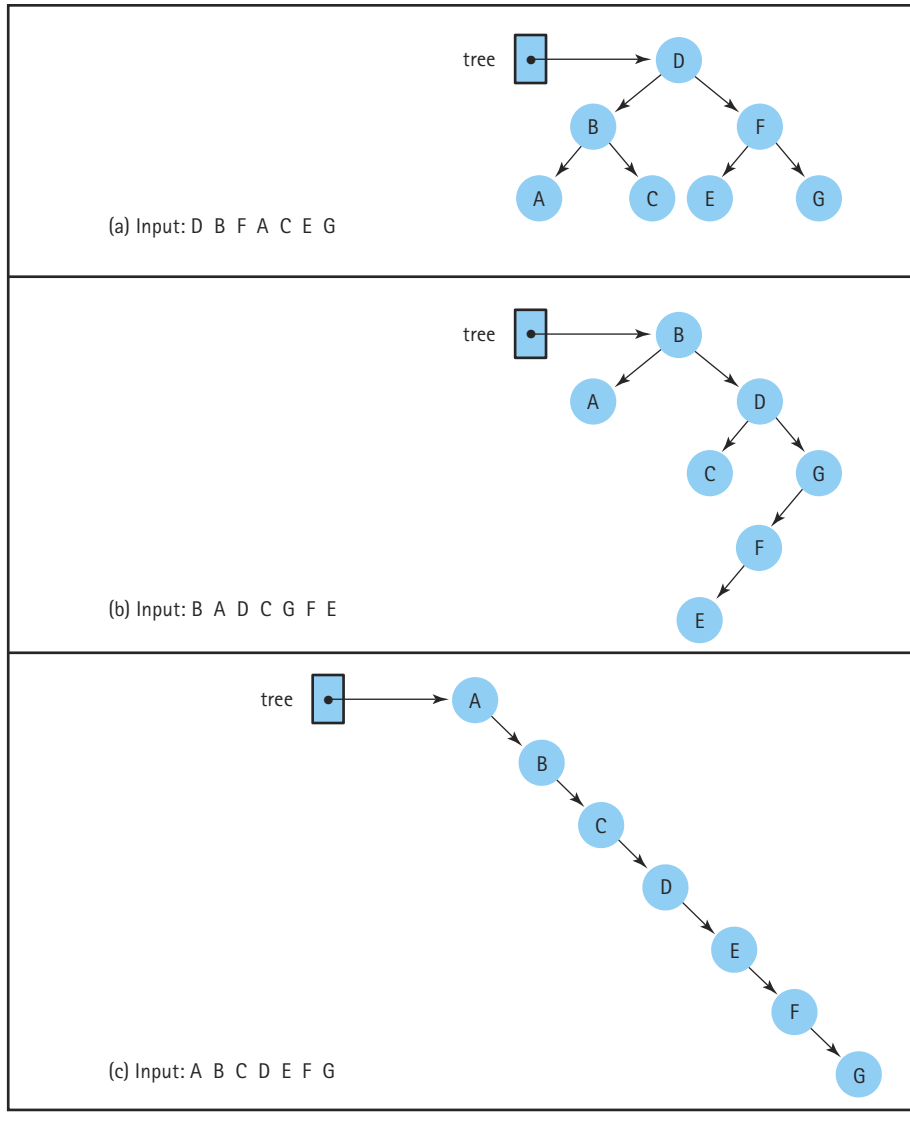


Figure 8.12 *The insert order determines the shape of the tree*

ent orders, produce very differently shaped trees. If the values are inserted in order (or in reverse order), the tree is completely *skewed* (a long "narrow" tree shape). A random mix of the elements produces a shorter, "bushy" tree. Because the height of the tree determines the maximum number of comparisons in a binary search, the tree's shape is very important. Obviously, minimizing the height of the tree maximizes the efficiency of the search. There are algorithms to adjust a tree to make its shape more desirable; one such scheme is presented in Section 8.8.

The delete Operation

The delete operation is the most complicated of the binary search tree operations. It is not difficult to locate the item to be deleted; but we must ensure when we delete it that we maintain the binary search tree property.

The set up for the delete operation is the same as that for the insert operation. The private `recDelete` method is invoked from the public `delete` method with parameters equal to the `item` to be deleted and the subtree to delete it from. The recursive method returns a reference to the revised tree, just as it did for `insert`. Here is the code for `delete`:

```
public void delete (Comparable item)
// Delete the element of this BST whose key matches item's key
{
  root = recDelete(item, root);
}
```


Again, in most cases the root of the tree is not affected by the `recDelete` call, in which case the assignment statement is somewhat superfluous, since it is reassigning the current value of `root` to itself. But, if the node being deleted happens to be the root node, then this assignment statement is crucial.

The `recDelete` method receives an item and the external reference to the binary search tree, finds and deletes the node matching the item's key from the tree, and returns a reference to the newly created tree. According to the specifications of the operation, an item with the same key exists in the tree. These specifications suggest a three-part operation:

recDelete: returns BSTNode

Find the node in the tree
Delete the node from the tree
return a reference to the new tree

We know how to find the node; we did it for `retrieve`. As with that operation, we use recursive calls to `recDelete` to progressively decrease the size of the tree where the target node could be, until we actually locate the node. Now we must delete the node and return a reference to the new subtree—this is somewhat complicated. This task varies according to the position of the target node in the tree. Obviously it is simpler to delete a leaf node than to delete a non-leaf node. In fact, we can break down the deletion algorithm into three cases, depending on the number of children linked to the node we want to delete:

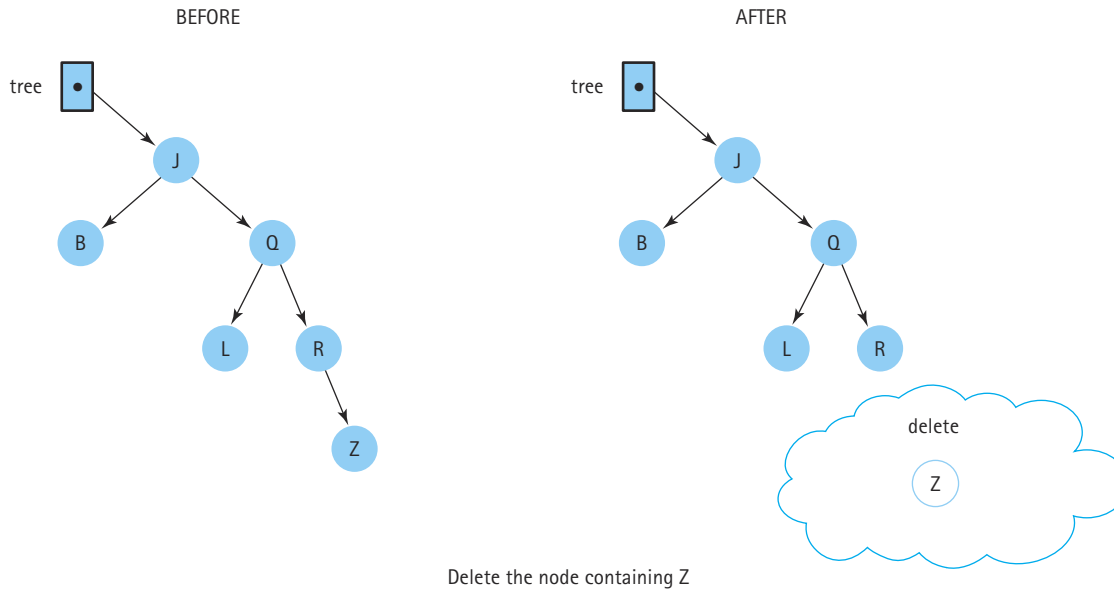


Figure 8.13 *Deleting a leaf node*

1. Deleting a leaf (no children): As shown in Figure 8.13, deleting a leaf is simply a matter of setting the appropriate link of its parent to `null`.
2. Deleting a node with only one child: The simple solution for deleting a leaf does not suffice for deleting a node with a child, because we don't want to lose all of its descendants from the tree. We want to make the reference from the parent skip over the deleted node and point instead to the child of the node we intend to delete (see Figure 8.14).
3. Deleting a node with two children: This case is the most complicated because we cannot make the parent of the deleted node point to both of the deleted node's children. The tree must remain a binary tree and the search property must remain intact. There are several ways to accomplish this deletion. The method we use does not delete the node but replaces its `info` with the `info` from another node in the tree so that the search property is retained. We then delete this other node. Hmmm. That also sounds like a candidate for recursion. Let's see how this turns out.

What element could we replace the deleted `item` with that would maintain the search property? The elements whose keys immediately precede or follow the key of `item`: the logical predecessor or successor of `item`. We replace the `info` of the node we wish to delete with the `info` of its logical predecessor—the node whose key is closest in value to, but less than, the key of the node to be deleted. Look back at Figure 8.10(j) and locate the logical predecessor of nodes 5, 9, and 7. Do you see the pattern? The logical predecessor of 5 is 4, the largest value in 5's left subtree. The logical predecessor of 9 is 8, the largest value in 9's left subtree. The logical predecessor of 7 is 6, the largest value

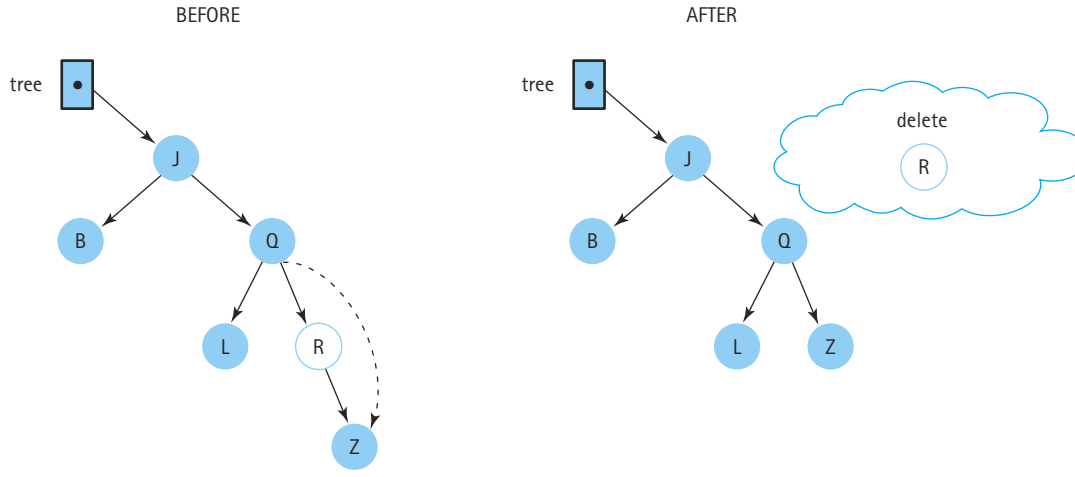


Figure 8.14 *Deleting a node with one child*

in 7's left subtree. This replacement value is in a node with either 0 or 1 children. We then delete the node the replacement value was in by changing one of its parent's references (see Figure 8.15).

Examples of all of these types of deletions are shown in Figure 8.16.

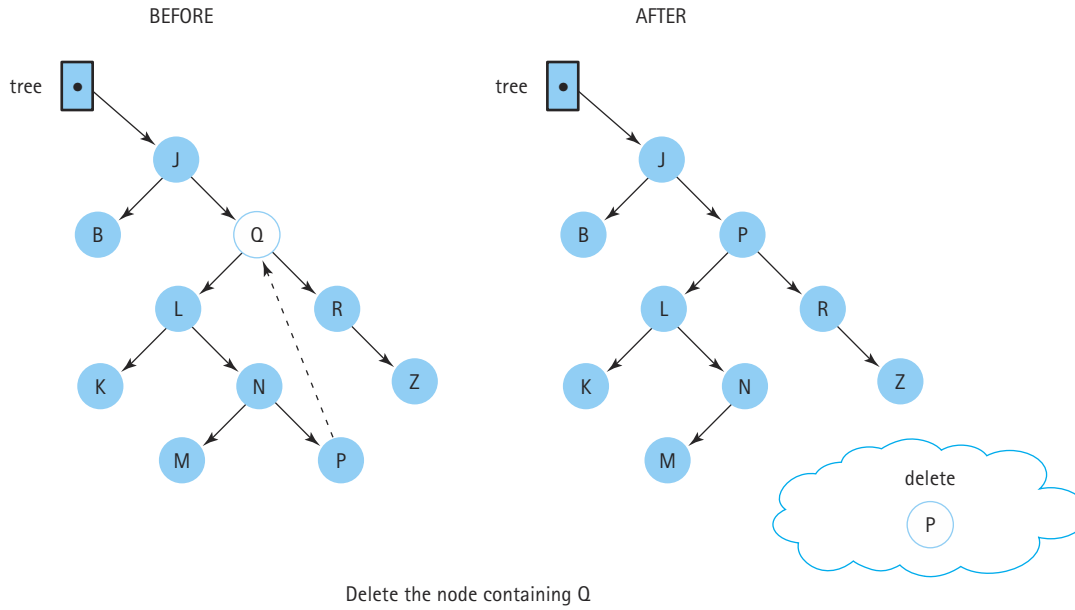


Figure 8.15 *Deleting a node with two children*

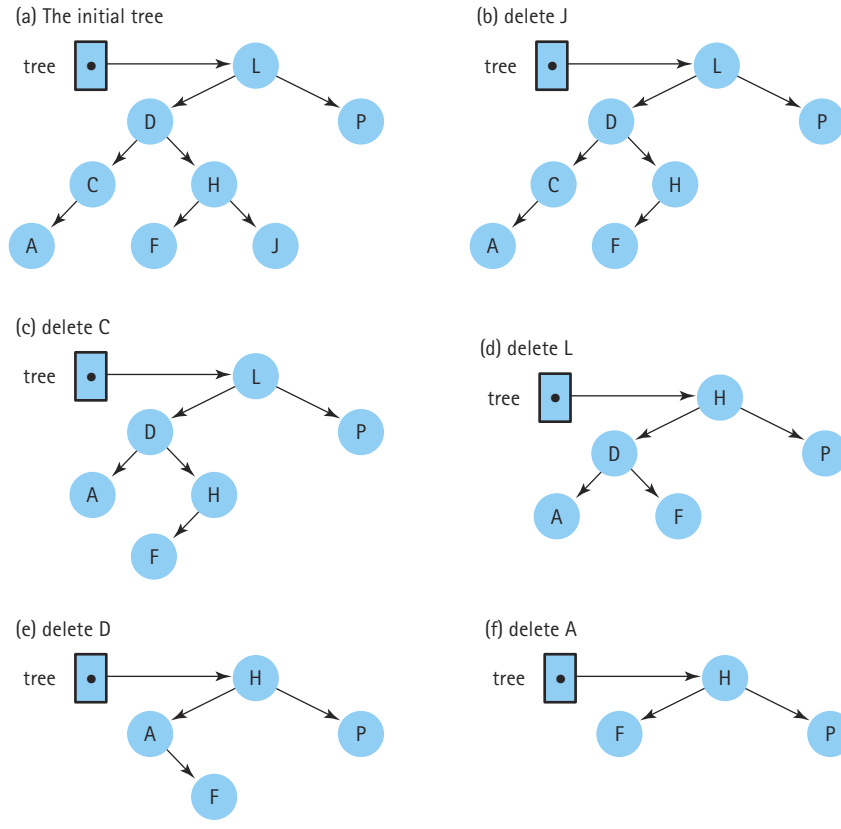


Figure 8.16 *Deletions from a binary search tree*

It is clear that the delete task involves changing references in the parent of the node to be deleted. This is why the `recDelete` method must return a reference to a `BSTNode`. Let's look at the three cases in terms of our implementation.

If both child references of the node to be deleted are `null`, the node is a leaf and we just return `null`; so the previous reference to this leaf node is replaced by `null` in the calling method, effectively removing the leaf node from the tree. If one child reference is `null`, we return the other child reference; the previous reference to this node is replaced by a reference to the node's only child, effectively jumping over the node and removing it from the tree (similar to the way we deleted a node from a singly linked list). If neither child reference is `null`, we replace the `info` of the node with the `info` of the node's logical predecessor and delete the node containing the predecessor. The node containing the predecessor came from the left subtree of the current node; so we delete it from that subtree. We then return the original reference to the node (we have not created a new node with a new reference; we have just changed the node's `info` reference).

Let's summarize all of this in algorithmic form as `deleteNode`. Within the algorithm and the code the reference to the node to be deleted is `tree`.

deleteNode (tree): returns BSTNode

```
if (tree.left( ) is null) AND (tree.right( ) is null)
      return null
else if tree.left( ) is null
      return tree.right( )
else if tree.right( ) is null
      return tree.left( )
else
      Find predecessor
      Set tree.info( ) to predecessor.info( )
      Set tree.left( ) to recDelete(predecessor.info( ), tree.left( ))
      return tree
```

Now we can write the recursive definition and code for `recDelete`.

Method recDelete (item, tree) returns BSTNode

Definition:	Deletes item from tree.
Size:	The number of nodes in the path from the root to the node to be deleted.
Base Case:	If item's key matches key in tree.info, delete node pointed to by tree.
General Case:	If item < tree.info(), recDelete(item, tree.left()); else recDelete(item, tree.right()).

```
private BSTNode recDelete(Comparable item, BSTNode tree)
// Deletes item from the tree
{
  if (item.compareTo(tree.info) < 0)
    tree.left = recDelete(item, tree.left);
  else if (item.compareTo(tree.info) > 0)
    tree.right = recDelete(item, tree.right);
  else
    tree = deleteNode(tree);    // Item is found
  return tree;
}
```

Before we code `deleteNode`, let's look at its algorithm again. We can remove one of the tests if we notice that the action taken when the left child reference is `null` also takes care of the case in which both child references are `null`. When the left child reference is `null`, the right child reference is returned. If the right child reference is also `null`, then `null` is returned, which is what we want if they are both `null`.

In good top-down fashion, let's now write the code for `deleteNode` using `getPredecessor` as the name of an operation that returns the `info` reference of the predecessor of the node with two children.

```
private BSTNode deleteNode(BSTNode tree)
// Deletes the node referenced by tree
// Post: The user's data in the node referenced to by tree is no
//       longer in the tree.  If tree is a leaf node or has only
//       a nonnull child pointer, the node pointed to by tree is
//       deleted; otherwise, the user's data is replaced by its
//       logical predecessor and the predecessor's node is deleted
{
  Comparable data;

  if (tree.left == null)
    return tree.right;
  else if (tree.right == null)
    return tree.left;
  else
  {
    data = getPredecessor(tree.left);
    tree.info = data;
    tree.left = recDelete(data, tree.left);  // Delete predecessor node.
    return tree;
  }
}
```

Now we must look at the operation for finding the logical predecessor. We know that the logical predecessor is the maximum value in `tree`'s left subtree. Where is this node? The maximum value in a binary search tree is in its rightmost node. Therefore, given `tree`'s left subtree, we just keep moving right until the right child is `null`. When this occurs, we return the `info` reference of the node. There is no reason to look for the predecessor recursively in this case. A simple iteration until `tree.right` is `null` suffices.

```
private Comparable getPredecessor(BSTNode tree)
// Returns the info member of the rightmost node in tree
{
  while (tree.right != null)
    tree = tree.right;
  return tree.info;
}
```

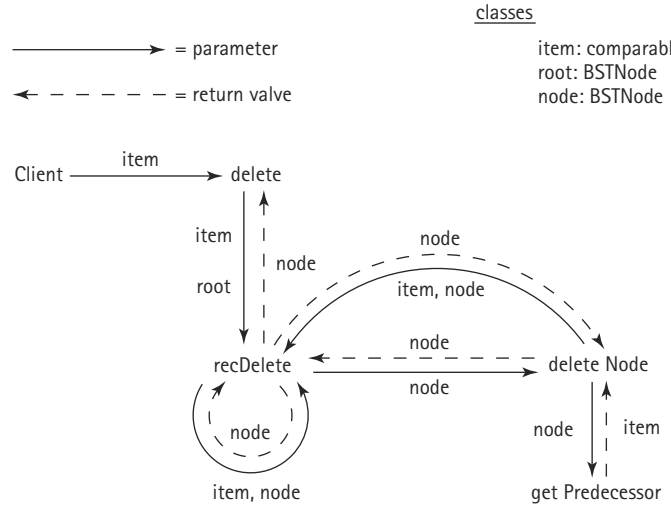


Figure 8.17 *The methods used to delete a node*

That's it. We have used four methods to implement the Binary Search Tree delete operation! See Figure 8.17 for a view of the calling relationships among the four methods. Did you notice that we used both types of recursion in our solution—we used both direct recursion (`recDelete` invokes itself) and indirect recursion (`recDelete` invokes `deleteNode`, which in turn may invoke `recDelete`). Due to the nature of our approach we are guaranteed that the indirect recursion never proceeds deeper than one level. Whenever `deleteNode` invokes `recDelete`, it passes an `item` and a reference to a subtree such that the `item` matches the largest element in the subtree. Therefore, the item matches the rightmost element of the subtree, which does not have a right child. This represents one of the base cases for the `recDelete` method and the recursion stops there.

Iteration

In the Print Tree operation developed in Section 8.3, we developed a client method to print the contents of a Binary Search Tree in order. The method printed the value of a node in between printing the values in its left subtree and the values in its right subtree. Using the inorder traversal resulted in a listing of the values of the binary search tree in ascending key order.

Let's review our traversal definitions:

- **Preorder traversal**: Visit the root, visit the left subtree, visit the right subtree.
- **Inorder traversal**: Visit the left subtree, visit the root, visit the right subtree.
- **Postorder traversal**: Visit the left subtree, visit the right subtree, visit the root.

Remember that the name given to each traversal specifies where the root itself is processed in relation to its subtrees.

Our Binary Search Tree ADT supports all three traversal orders. How can it do this? As you saw for the Print Tree operation, the client program passes the `reset` and `getNextItem` methods a parameter indicating which of the three traversal orders to use for that particular invocation of the method. Imagine, for example, that `reset` is called with an `INORDER` argument, followed by several calls to `getNextItem` with `INORDER` arguments. How does `getNextItem` keep track of which tree node to return next? It is not as simple as maintaining an instance variable that references the next item, as we did for linked lists. A simple reference to a node does not capture the status of the traversal. Sure, `getNextItem` could return the referenced item, but then how does it update the reference in preparation for the next call? Does it go down the left subtree, or down the right subtree, or back up to the parent? The program could save more information about the current traversal, enough to let it find the next item; due to the recursive nature of the traversals it would have to save this information in a stack.

However, there is a simpler way: We let `reset` generate a queue of node contents in the indicated order and let `getNextItem` process the node contents from the queue. Each of the traversal orders is supported by a separate queue. Therefore, the instance variables of our `BinarySearchTree` class must include three queues:

```
protected ArrayQueue inOrderQueue;    // Queue of info
protected ArrayQueue preOrderQueue;   // Queue of info
protected ArrayQueue postOrderQueue;  // Queue of info
```

The `reset` method instantiates one of the queues, based on its parameter, and initializes it to a size equal to the current size of the tree. It then calls one of three recursive methods, depending on the value of its parameter. Each of these methods implements a recursive traversal, enqueing the node contents onto the corresponding queue in the appropriate order. Below is the code for `reset` and `getNextItem`: Note that `reset` calls the `numberOfNodes` method in order to determine how large to make the required queue. In most cases, a client program that invokes `reset` immediately requires the same information, to control how many times they call the `getNextItem` method to iterate through the tree. See the code for `printTree` in Section 8.3, for example. We make it easy for the client to obtain the number of nodes, by providing it as the return value of `reset`. Since `reset` needs to call `numberOfNodes` anyway, this requires minimal extra cost.

What happens when `getNextItem` reaches the end of the collection of items? At that point, the corresponding queue is empty, and another call to `getNextItem` results in a run time exception being thrown. Unlike with our List ADT, iterations on Binary Search Trees do not "wrap around." The client must be sure not to call `getNextItem` inappropriately.

```
public int reset(int orderType)
// Initializes current position for an iteration through this BST in orderType
//  order
{
  int numNodes = numberOfNodes();
```

```
  if (orderType == INORDER)
  {
    inOrderQueue = new ArrayQueue(numNodes);
    inOrder(root);
  }
  else
  if (orderType == PREORDER)
  {
    preOrderQueue = new ArrayQueue(numNodes);
    preOrder(root);
  }
  if (orderType == POSTORDER)
  {
    postOrderQueue = new ArrayQueue(numNodes);
    postOrder(root);
  }
  return numNodes;
}

public Comparable getNextItem (int orderType)
// Returns a copy of the element at the current position in this BST and
//  advances the value of the current position based on the orderType set
//  through reset
{
  if (orderType == INORDER)
    return (Comparable)inOrderQueue.dequeue();
  else
  if (orderType == PREORDER)
    return (Comparable)preOrderQueue.dequeue();
  else
  if (orderType == POSTORDER)
    return (Comparable)postOrderQueue.dequeue();
  else return null;
}
```

All that is left to do is to define the three traversal methods to store the required information into the queues in the correct order.

We start with the inorder traversal. We first need to visit the root's left subtree, all the values in the tree that are smaller than the value in the root node. Then we visit the root node by enqueing its `info` reference in our `inOrderQueue`. Finally, we visit the root's right subtree, all the values that are larger than the value in the root node (see Figure 8.18).

Let's describe this problem again, developing our algorithm as we proceed. We assume our method is named `inOrder` and is passed a parameter `tree`. We want to visit the elements in the binary search tree rooted at `tree` in order; that is, first we visit the left subtree in order, then we visit the root, and finally we visit the right subtree in order.

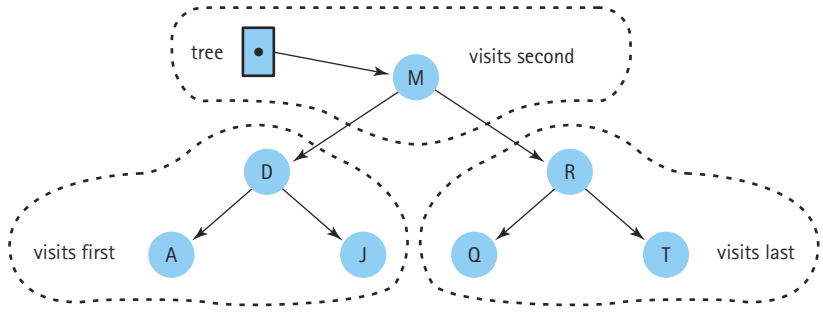


Figure 8.18 *Visiting all the nodes in order*

`tree.left` references the root of the left subtree. Because the left subtree is also a binary search tree, we can call method `inOrder` to visit it, using `tree.left` as the argument. When `inOrder` finishes visiting the left subtree, we enqueue the `info` reference of the root node. Then we call method `inOrder` to visit the right subtree with `tree.right` as the argument.

Of course, each of the two calls to method `inOrder` uses the same approach to visit the subtree: They visit the left subtree with a call to `inOrder`, visit the root, and then visit the right subtree with another call to `inOrder`. What happens if the incoming parameter is `null` on one of the recursive calls? This situation means that the parameter is the root of an empty tree. In this case, we just want to exit the method—clearly there's no point to visiting an empty subtree. That is our base case.

Method inOrder (tree)

Definition:	Enqueues the items in the binary search tree in order from smallest to largest.
Size:	The number of nodes in the tree whose root is tree
Base Case:	If tree = null, do nothing.
General Case:	Traverse the left subtree in order. Then enqueue tree.info(). Then traverse the right subtree in order.

This description is coded as the following recursive method:

```
private void inOrder(BSTNode tree)
// Initializes inOrderQueue with tree elements in inOrder order
{
  if (tree != null)
```

```
  {
    inOrder(tree.left);
    inOrderQueue.enqueue(tree.info);
    inOrder(tree.right);
  }
}
```

The remaining two traversals are approached exactly the same, except the relative order in which they visit the root and the subtrees is changed. Recursion certainly allows for an elegant solution to the binary tree traversal problem.

```
private void preOrder(BSTNode tree)
// Initializes preOrderQueue with tree elements in preOrder order
{
  if (tree != null)
  {
    preOrderQueue.enqueue(tree.info);
    preOrder(tree.left);
    preOrder(tree.right);
  }
}

private void postOrder(BSTNode tree)
// Initializes postOrderQueue with tree elements in postOrder order
{
  if (tree != null)
  {
    postOrder(tree.left);
    postOrder(tree.right);
    postOrderQueue.enqueue(tree.info);
  }
}
```


Testing Binary Search Tree Operations

Now that we've finished the implementation of the Binary Search Tree ADT we must address testing our implementation. The code for the entire `BinarySearchTree` class is contained on our web site, including both the recursive `numberOfNodes` method, and the iterative version (`numberOfNodes2`). We have also included a test driver for the ADT called `TDBinarySearchTree`. In addition to directly supporting testing of all the ADT operations, the test driver also supports a `printTree` operation that accepts one of the order constants as a parameter and "prints" the contents of the tree, in that order, to an output file. See Figure 8.19 for an example of a test input file, the resulting output file, and the screen information generated during the test. Note that the `printTree` results can be used to help verify the shape of the tree.

```
Test Case for  TextBook
BinarySearchTree
numberOfNodes
numberOfNodes2
isEmpty
insert
L
insert
D
insert
P
insert
C
insert
H
insert
J
numberOfNodes
numberOfNodes2
isThere
J
delete
J
printTree
INORDER
printTree
PREORDER
quit
```

File: tree1.dat

```
Results Test Case for  TextBook

The tree is instantiated
The number of nodes in the tree is    0
The (iterative) number of nodes in the tree is    0
The tree is empty is true
L was inserted into the tree
D was inserted into the tree
P was inserted into the tree
C was inserted into the tree
H was inserted into the tree
J was inserted into the tree
The number of nodes in the tree is    6
The (iterative) number of nodes in the tree is    6
J is on the tree: true
J was deleted from the tree
The tree in Inorder is:
1. C
2. D
3. H
4. L
5. P
The tree in Preorder is:
1. L
2. D
3. C
4. H
5. P
```

File: tree1.out

Screen Output

Command: *java TDBinarySearchTree tree1.dat tree1.out*

Figure 8.19 *Example of a test input file and the resulting output file*

You are invited to use the test driver to test the various tree operations. Be sure to test all of the operations, in many combinations. In particular, you should test both skewed and balanced trees.

8.7 Comparing Binary Search Trees to Linear Lists

A binary search tree is an appropriate structure for many of the same applications discussed previously in conjunction with other sorted list structures. The special advantage of using a binary search tree is that it facilitates searching while conferring the benefits of linking the elements. It provides the best features of both the sorted array-based list and the linked list. Like a sorted array-based list, it can be searched quickly, using a binary search. Like a linked list, it allows insertions and deletions without having to move data. Thus, it is particularly suitable for applications in which search time must be minimized or in which the nodes are not necessarily processed in sequential order.

As usual, there is a tradeoff. The binary search tree, with its extra reference in each node, takes up more memory space than a singly linked list. In addition, the algorithms for manipulating the tree are somewhat more complicated. If all of the list's uses involve sequential rather than random processing of the elements, the tree may not be as good a choice.

Suppose we have 100,000 customer records in a list. If the main activity in the application is to send out updated monthly statements to the customers and if the order in which the statements are printed is the same as the order in which the information appears on the list, a linked list would be suitable. But suppose we decide to keep a terminal available to give out account information to the customers whenever they ask. If the data are kept in a linked list, the first customer on the list can be given information almost instantly, but the last customer has to wait while the other 99,999 nodes are examined. When direct access to the nodes is a requirement, a binary search tree is a more appropriate structure.

Big-O Comparisons

Finding a node (`isThere`), as we would expect in a structure dedicated to searching, is the most interesting operation to analyze. In the best case—if the order that the elements were inserted results in a short and bushy tree—we can find any node in the tree with at most $\log_2 N + 1$ comparisons. We would expect to be able to locate a random element in such a tree much faster than finding an element in a sorted linked list. In the worst case—if the elements were inserted in order from smallest to largest or vice versa—the tree won't really be a tree at all; it would be a linear list, linked through either the `left` or `right` references. This is called a "degenerate" tree. In this case, the tree operations should perform much the same as the operations on a linked list. Therefore, if we were doing a worst-case analysis, we would have to say that the complexity of the tree operations is identical to the comparable linked-list operations. However, in the following analysis, we assume that the items are inserted into the tree in random order giving a balanced tree.

The `insert` and `retrieve` operations are basically finding the node [$O(\log_2 N)$] plus tasks that are $O(1)$—for instance, creating a node or resetting references. Thus these operations are all described as $O(\log_2 N)$. The `delete` operation consists of finding the node plus invoking `deleteNode`. In the worst case (deleting a node with two children), `deleteNode` must find the replacement value, a $O(\log_2 N)$ operation. (Actually, the two tasks together add up to $\log_2 N$ comparisons, because if the target node is higher in the tree, fewer comparisons are needed to find it, but more comparisons may be needed to find its replacement node; and vice versa.) Otherwise, if the deleted node has 0 or 1 child, `deleteNode` is a $O(1)$ operation. So `delete` too may be described as $O(\log_2 N)$.

The `numberOfNodes` and `reset` operations require the tree to be traversed, processing each element once. Thus these are $O(N)$ operations. Of course, iterating through an entire collection of elements takes $O(N)$ steps even if both `reset` and `getNextItem` are $O(1)$, since `getNextItem` must be called N times by the client.

The orders of magnitude for the tree and list operations as we have coded them are compared in Table 8.2. The binary search tree operations are based on a random inser-

Table 8.2 *Big-O Comparison of List Operations*

	Binary Search Tree	Array-Based Linear List	Linked List
Class constructor	$O(1)$	$O(1)$	$O(1)$
`isFull`	$O(1)$	$O(1)$	$O(1)$
`isEmpty`	$O(1)$	$O(1)$	$O(1)$
`reset`	$O(N)$	$O(1)$	$O(1)$
`getNextItem`	$O(1)$	$O(1)$	$O(1)$
`isThere`	$O(\log_2 N)$	$O(\log_2 N)$	$O(N)$
`retrieve`			
Find	$O(\log_2 N)$	$O(\log_2 N)$	$O(N)$
Process	$O(1)$	$O(1)$	$O(1)$
Total	$O(\log_2 N)$	$O(\log_2 N)$	$O(N)$
`insert`			
Find	$O(\log_2 N)$	$O(\log_2 N)$	$O(N)$
Process	$O(1)$	$O(N)$	$O(N)$
Total	$O(\log_2 N)$	$O(N)$	$O(N)$
`delete`			
Find	$O(\log_2 N)$	$O(\log_2 N)$	$O(N)$
Process	$O(1)$	$O(N)$	$O(1)$
Total	$O(\log_2 N)$	$O(N)$	$O(N)$

tion order of the items; the Find operation in the array-based implementation is based on using a binary search. We do not list the list's `lengthIs` method or the tree's `numberOfNodes` method. These methods can be implemented with a simple *return* statement if the object maintains an instance variable holding the size of the structure. Of course, this instance variable would have to be updated every time an item is inserted or deleted from the structure, so the cost really depends on how often those operations occur.

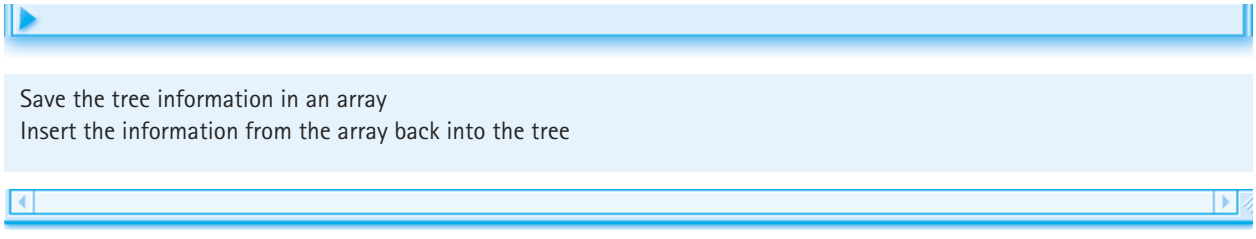
8.8 Balancing a Binary Search Tree

In our Big-O analysis of binary search tree operations we assumed our tree was balanced. If this assumption is dropped and if we perform a worst-case analysis assuming a completely skewed tree, the efficiency benefits of the binary search tree disappear. The time required to perform the `isThere`, `retrieve`, `insert`, and `delete` operations is now O(N), just as it is for the linked list. Therefore, a beneficial addition to our Binary Search Tree ADT operations is a `balance` operation that balances the tree. The specification of the operation is:

```
public balance();
// Effect:          Restructures this BST to be optimally balanced
// Postcondition:   This BST has a minimum number of levels
//                  The information in the tree is unchanged
```

Of course, it is up to the client program to use the `balance` method appropriately. It should not be invoked too often, since it also has an execution cost associated with it.

There are several ways to restructure a binary search tree. We use a simple approach:



The structure of the new tree depends on the order that we save the information into the array, or the order in which we insert the information back into the tree, or both. Let's start by assuming we insert the array elements back into the tree in "index" order, that is, starting at index 0 and working through the array. We use the following algorithm:

```
Set ORDER to one of the tree traversal orders.
Set count to tree.reset(ORDER).
for (int index = 0; index < count; index++)
      Set array[index] = tree.getNextItem(ORDER).
tree = new BinarySearchTree().
for (int index = 0; index < count; index++)
      tree.insert(array[index]).
```

Does that balance the tree? Its impossible to tell what it does without knowing the order of the tree traversal.

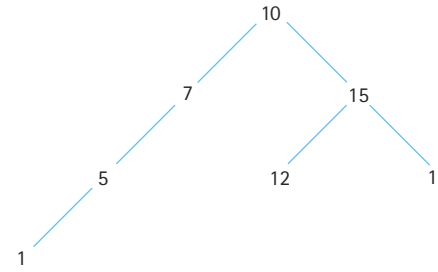
Let's consider what happens if we use an inorder traversal. See Figure 8.20 for an example. The results are not very satisfactory are they? We have taken a perfectly nice bushy tree and turned it into a degenerate skewed tree that does not efficiently support our tree operations. That is the opposite of what we hope to accomplish. There must be a better approach.

Next we try a preorder traversal. See Figure 8.21. We end up with an exact copy of our tree. Will this always be the case? Yes. Recall our rule for inserting items into a tree—we always insert at a leaf position. Do you see what this means? Consider the tree in Figure 8.21(a). Assuming it has had no deletions, what was the first node inserted into that tree when it was created? It had to be the root node, the node containing the value 10, since the only time the root node is also a leaf is when the tree contains a single element. What was the second element inserted? It was either the 7 or the 15—they are the roots of the left subtree and right subtree. If we recreate the tree based on a preorder traversal (root first!) of the original tree, we obtain an exact copy of the tree. Since we always insert a node before any of its descendants, we maintain the ancestor-descendant relationships of the original tree, and obtain an exact copy. This is an interesting discovery, and could be useful if we wanted to duplicate a tree, but it doesn't help us balance a tree.

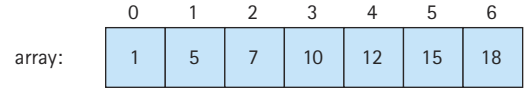
Using a postfix traversal doesn't help either. Try it out.

How can we do better? One way to ensure a balanced tree is to even out, as much as possible, the number of descendants in each node's left and right subtrees. Since we insert items "root first" this means that we should first insert the "middle" item. (If we list the items from smallest to largest, the "middle" item is the one in the middle of the list—it has as many items less than it as it has greater than it, or at least as close as possible.) The middle item becomes the root of the tree. It has about the same number of descendants in its left subtree as it has in its right subtree. Good. What item do we insert next? Let's work on the left subtree. The root of the left subtree should be the "middle" item of all the items that are less than the root. That item is inserted next. Now, when the remaining items that are less than the root are inserted, about half of

(a) The original tree



(b) The inorder traversal



(c) The resultant tree if linear traversal of array is used

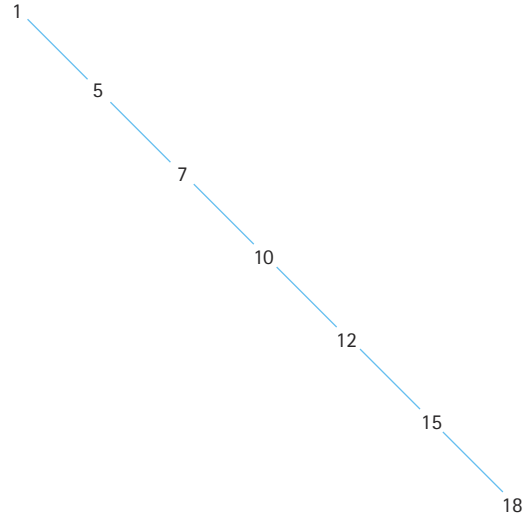
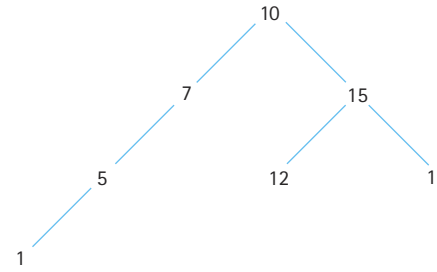
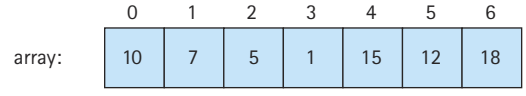


Figure 8.20 *A skewed tree is produced*

(a) The original tree



(b) The preorder traversal



(c) The resultant tree if linear traversal of array is used

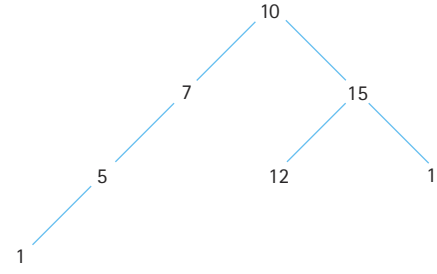


Figure 8.21 *An exact copy is produced*

them will be in the left subtree of the item, and about half will be in its right subtree. And so on. Sounds recursive, doesn't it?

Here is an algorithm for balancing a tree based on the approach described above. The algorithm consists of two parts, one iterative and one recursive. The iterative part, Balance, creates the array and invokes the recursive part, InsertTree, that rebuilds the tree. We first store the nodes of the tree into our array using an inorder traversal. Therefore, they are stored, in order, from smallest to largest. The algorithm continues by

invoking the recursive algorithm InsertTree, passing it the bounds of the array. The InsertTree algorithm checks the array bounds it is passed. If the low and high bounds are the same (base case 1) it simply inserts the corresponding array element into the tree. If the bounds only differ by one location (base case 2) the algorithm inserts both elements into the tree. Otherwise, it computes the "middle" element of the subarray, inserts it into the tree, and then makes two recursive calls to itself: one to process the elements less than the middle element, and one to process the elements greater than the element.

Balance

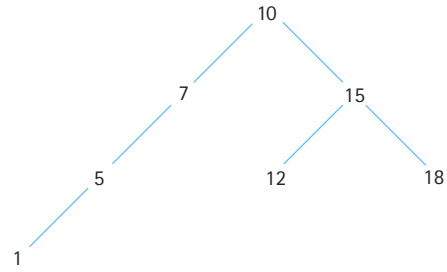
```
Set count to tree.reset(INORDER).
For (int index = 0; index < count; index++)
      Set array[index] = tree.getNextItem(ORDER).
tree = new BinarySearchTree().
tree.InsertTree(0, count − 1)
```

InsertTree(low, high)

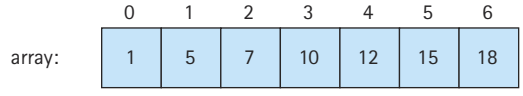
```
if (low == high)                    // Base case 1
      tree.insert(nodes[low]).
else if ((low + 1) == high)         // Base case 2
      tree.insert(nodes[low]).
      tree.insert(nodes[high]).
else
      mid = (low + high) / 2.
      tree.insert(mid).
      tree.InsertTree(low, mid – 1).
      tree.InsertTree(mid + 1, high).
```

Trace the InsertTree algorithm using sorted arrays of both even and odd length to convince yourself that it works. The code for `balance` and a helper method `insertTree` follows directly from the algorithm and is left as an exercise. Figure 8.22 shows the results of using this approach on the previous example.

(a) The original tree



(b) The inorder traversal



(c) The resultant tree if InsertTree (0,6) is used

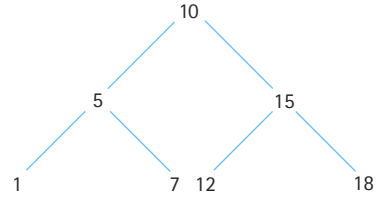


Figure 8.22 *An optimal transformation*

8.9 A Nonlinked Representation of Binary Trees

Our discussion of the implementation of binary trees has so far been limited to a scheme in which the links from parent to children are *explicit* in the implementation structure. An instance variable was declared in each node for the reference to the left child and the reference to the right child.

A binary tree can be stored in an array in such a way that the relationships in the tree are not physically represented by link members, but are *implicit* in the algorithms that manipulate the tree stored in the array. The code is, of course, much less self-documenting, but we might save memory space because there are no references.

Let's take a binary tree and store it in an array in such a way that the parent–child relationships are not lost. We store the tree elements in the array, level by level, left-to-right. If the number of nodes in the tree is `numElements`, we can package the array and `numElements` into an object as illustrated in Figure 8.23. The tree elements are stored with the root in `tree.nodes[0]` and the last node in `tree.nodes[numElements - 1]`.

To implement the algorithms that manipulate the tree, we must be able to find the left and right child of a node in the tree. Comparing the tree and the array in Figure 8.23, we see that

`tree.nodes[0]`'s children are in `tree.nodes[1]` and `tree.nodes[2]`.
`tree.nodes[1]`'s children are in `tree.nodes[3]` and `tree.nodes[4]`.
`tree.nodes[2]`'s children are in `tree.nodes[5]` and `tree.nodes[6]`.

Do you see the pattern? For any node `tree.nodes[index]`, its left child is in `tree.nodes[index*2 + 1]` and its right child is in `tree.nodes[index*2 + 2]` (provided that these child nodes exist). Notice that the nodes in the array from `tree.nodes[tree.numElements/2]` to `tree.nodes[tree.numElements - 1]` are leaf nodes.

Not only can we easily calculate the location of a node's children, we also can determine the location of its parent node. This task is not an easy one in a binary tree linked together with references from parent to child nodes, but it is very simple in our implicit link implementation: `tree.nodes[index]`'s parent is in `tree.nodes[(index - 1)/2]`.

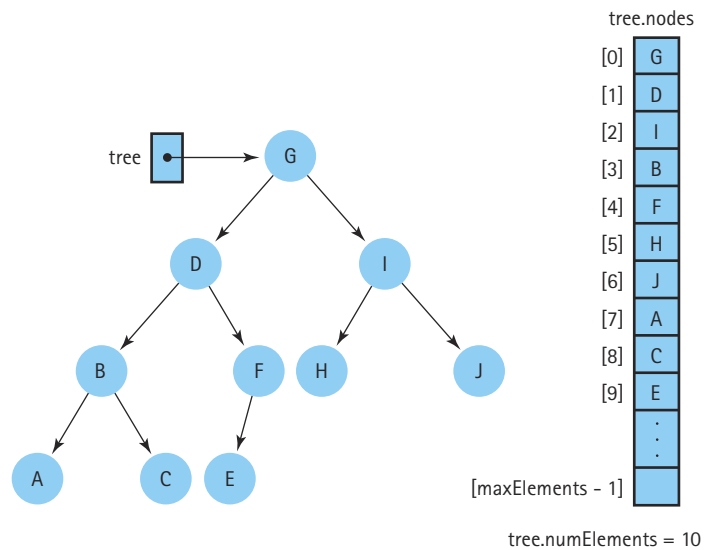
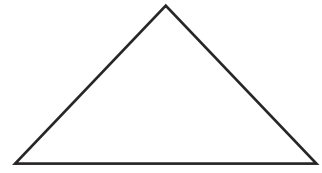


Figure 8.23 *A binary tree and its array representation*

Because integer division truncates any remainder, `(index - 1)/2` is the correct parent index for either a left or right child. Thus, this implementation of a binary tree is linked in both directions: from parent to child, and from child to parent. We take advantage of this fact in the next chapter when we study heaps.

This tree representation works well for any binary tree that is full or complete. A full binary tree is a binary tree in which all of the leaves are on the same level and every nonleaf node has two children. The basic shape of a full binary tree is triangular:

Full binary tree A binary tree in which all of the leaves are on the same level and every nonleaf node has two children



A complete binary tree is a binary tree that is either full or full through the next-to-last level, with the leaves on the last level as far to the left as possible. The shape of a complete binary tree is either triangular (if the tree is full) or something like the following:

Complete binary tree A binary tree that is either full or full through the next-to-last level, with the leaves on the last level as far to the left as possible

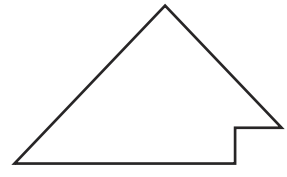


Figure 8.24 shows some examples of different types of binary trees.

The array-based representation is simple to implement for trees that are full or complete, because the elements occupy contiguous array slots. If a tree is not full or complete, however, we must account for the gaps where nodes are missing. To use the array representation, we must store a dummy value in those positions in the array in order to maintain the proper parent–child relationship. The choice of a dummy value depends on the information that is stored in the tree. For instance, if the elements in the tree are nonnegative integers, a negative value can be stored in the dummy nodes; or if the elements are objects we could use a `null` value.

Figure 8.25 illustrates a tree that is not complete and its corresponding array. Some of the array slots do not contain actual tree elements; they contain dummy values. The

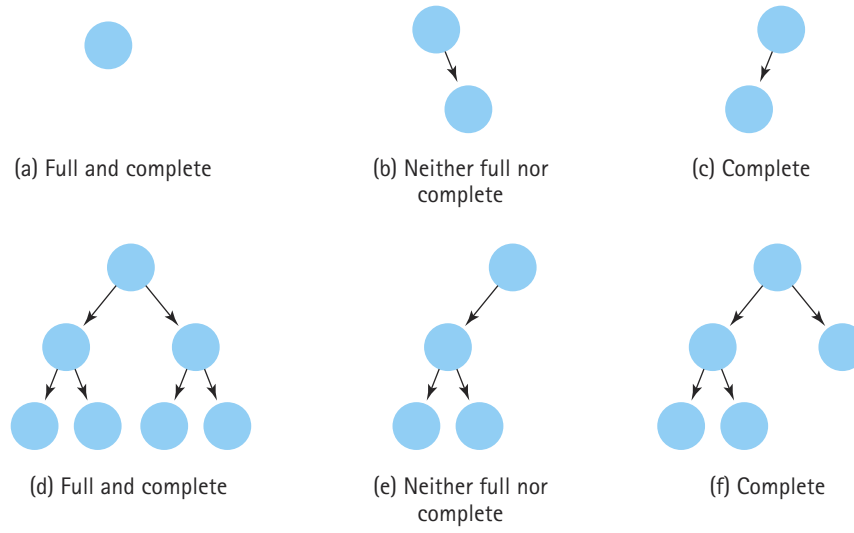


Figure 8.24 *Examples of different types of binary trees*

algorithms to manipulate the tree must reflect this situation. For example, to determine whether the node in `tree.nodes[index]` has a left child, you must check whether `index*2 + 1 < tree.numElements`, and then check to see if the value in `tree.nodes[index*2 + 1]` is the dummy value.

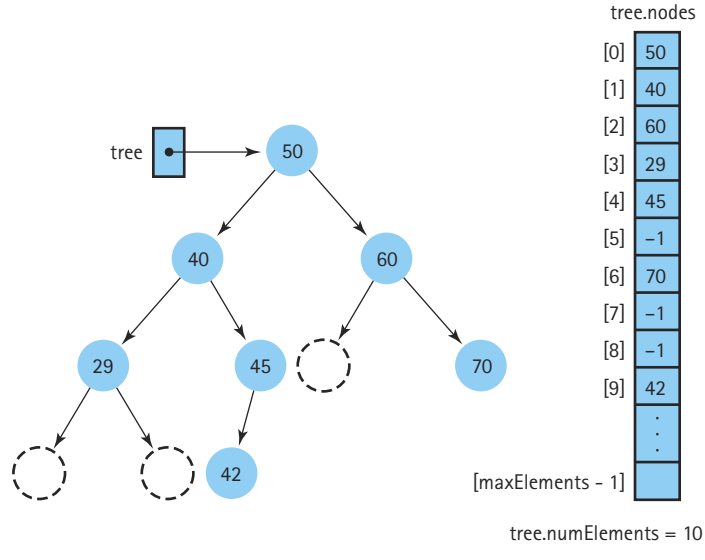


Figure 8.25 *A binary search tree stored in an array with dummy values*

Case Study

Word Frequency Generator

Problem Our firm is planning to create some text analysis software, for example, software that automatically calculates the reading level of a document. As a first step, we've been assigned the task of creating a Word Frequency Generator. This generator is to be used to perform some preliminary text analysis during the planning stage of the project; if it works well it may be incorporated into the tools developed later.

The word frequency generator is to read a text file indicated by the user, and generate an alphabetical listing of the unique words that it contains, along with a count of how many times each word occurs. The output is to go to a text file, also indicated by the user. To allow the user to control the amount of useful output from the generator, based on the particular problem they are studying, the generator must allow the user to specify a minimum word size and a minimum frequency count. The generator should skip over words smaller than the minimum word size; the generator should not include a word on its output list if the word occurs less times than the minimum frequency count. Finally, the generator should present a few summary statistics: the total number of words, the number of words whose length is at least the minimum word size, and the number of unique words of the specified size whose frequency is at least the minimum frequency count.

Discussion Clearly, the main object in this problem is a word with associated frequency. Therefore, the first thing we must do is define a "word."

We discuss this question with our manager. What is a tentative definition of a word in this context? How about "something between two blanks?" Or better yet, a "character string between two blanks." That definition works for most words. However, all words before '.' and ',' would have the '.' and ',' attached. Also, words with quotes would cause a problem. Therefore, we settle on the following definition: A word is a string of alphanumeric characters between markers where markers are whitespace and all punctuation marks. Although we "lose" some words following this definition (for example contractions such as "can't" are treated as two words "can" and "t") we decide that these small problems do not adversely affect our goals. Finally, our manager points out that all words should be transformed into lowercase characters for processing—"THE" and "the" represent the same word.

Brainstorming As usual, the first step is to list objects that might be useful in solving the problem. Scanning the problem statement above we identify the following "nouns": *word frequency generator*, *input text file*, *user*, *alphabetical listing*, *unique words*, *word count*, *output text file*, *minimum word size*, *minimum frequency count*, *output list*, *summary statistics*, and *word totals*. That's 12 candidate objects. We realize that we need to use a data structure to store words—for now we just call it the *word store* and add that to our list, giving us 13 candidate objects.

Listing the verbs in a problem statement often helps identify the actions our program needs to take. In this case, however, we are clear on the actions needed: read in the words, determine their frequencies, and output the results. A quick scan of the problem statement reminds us that the results are to be sorted alphabetically and that some pruning of the data is required based on minimum thresholds for word size and word frequency.

Filtering We have identified 13 candidate objects. A few of them can be discarded immediately: the *word frequency generator* is the entire program; *minimum word size* and *minimum frequency count* are input values. Some of the candidate objects (*input text file*, *user*, *output text file*, *output list*) are related to the user interface. We discuss the interface in the next subsection. The remaining candidates are grouped as follows:

- *summary statistics*, *word totals*—these are really the same thing; we decide the statistics can be tracked by the main processing class and presented to the user at the conclusion of processing—this does not require a separate object
- *unique words*, *word count*—these two are related since we need to have a word count for each unique word; we decide to create a class called `WordFreq` to hold a word-frequency pair; a quick analysis tells us that we have to be able to initialize objects of the class, increment the frequency, and observe both the word and the frequency values.
- *alphabetical listing*, *word store*—we realize that we can combine these objects by using a container that supports sorted traversals to hold our `WordFreq` objects; we decide to delay our choice of containers until after we perform scenario analysis, but we know it must support insertion, searching, and "in order" traversal. At this point we consider both the Sorted List and the Binary Search Tree ADTs as candidates. For now we call this class `Container`.

The User Interface We decide not to create a graphical user interface for this program. It is not worth the time because the program is only to be used for preliminary analysis of texts, as an aid in planning full fledged text analysis tools, and the program is targeted to be embedded in a larger system later, a system that already has a user interface.

Therefore, we use an I/O approach identical to that used for our test drivers and small examples throughout the text. The user enters four pieces of information to the program through command-line parameters: the name of the input file, the name of the output file, the minimum word size, and the minimum frequency count. The program reads text from the input file, breaks it into words, generates the word frequency list to the output file, and displays the summary statistics in a frame. The user closes the frame to end the program.

Error Handling For the same reasons we are not creating a graphical interface, we do not worry about checking input parameters for validity. We assume that when our program is embedded in a larger system that that system will ensure valid input. For now, we rely on the user to supply appropriate program arguments.

CRC Cards During the filtering stage, we identified two primary classes to support our program, the `WordFreq` class and a "storage" class that we call `Container` for now. Before proceeding with scenario analysis, we create preliminary CRC cards for each of these classes. These cards can be modified during the scenario analysis and design stages. Note that we intend to use one of our predefined ADTs for the `Container` class; the scenario analysis should help us determine which one is the best fit.

While creating the CRC card for `Container` we immediately realize that the `WordFreq` class must support a `compareTo` operation, since the `Container` class needs to compare `WordFreq` objects to perform many of its responsibilities. So we add `compareTo` to the list of responsibilities for `WordFreq`.

Class Name: WordFreq	Superclass: Object	Subclasses:
Main Responsibility: Hold a word-frequency pair		
Responsibilities	**Collaborations**	
Create itself(word)	None	
Increment its frequency count	None	
Compare itself to another WordFreq (otherWordFreq), return int	String	
Know its information	None	
Know word, return String		
Know freq, return int		

Class Name: Container	Superclass: Object	Subclasses:
Main Responsibility: Store WordFreq objects		
Responsibilities	**Collaborations**	
Create itself	None	
Receive a WordFreq object for storage	WordFreq	
Receive a WordFreq object and know whether it already has an object with the same key, return boolean	WordFreq	
Receive a WordFreq object and return the object it is storing with the same key, return WordFreq	WordFreq	
Provide an alphabetical iteration through the stored WordFreq objects	WordFreq	

Scenario Analysis

There really is only one scenario for this problem: read a file, break it into words, process the words, and output the results. Let's describe it in algorithmic form:

```
Get arguments (input, output, minimum size, minimum frequency) from the user
Set up input file for reading
Set up output file for writing
Set up container object
Set numWords to 0
Set numValidWords to 0
Set numValidFreqs to 0

Read a line of input
while not at the end of the input
   Break current input line into words
   Set current word to the first word from the input line
   while there are still words to process
      Increment numWords
      if current word size is OK
         Increment numValidWords
         Change word to all lower case
         wordToTry = WordFreq(current word)
         if wordToTry is already in the container
            wordInTree = container.retrieve(wordToTry)
            Increment the frequency of wordInTree
            Save wordInTree back into the container
         else
            Insert wordToTry into the container
      Set current word to the next word from the input line
   Get next line of input

Set up an in order traversal through the container
while there are more WordFreq items
   Get the next WordFreq item
   if the frequency of the WordFreq item is large enough
      Increment numValidFreqs
      Output the WordFreq information to the output file

Output the summary statistics to a Frame
```

We need to walk through this algorithm a few times to complete our helper class definitions. As we walk through it we ask ourselves, at each step, how we could accomplish that step:

- We already know how to handle the input and output.
- We can use Java's string tokenizer, introduced in Chapter 4's case study, to break the input line into words. Note that the string tokenizer class allows us to pass a string of

characters to its constructor to use as token delimiters. For this program, we pass it a string containing all the common punctuation and whitespace characters.
- Setting and incrementing the variables are trivial.
- Checking a word's size and changing it to lowercase can both be accomplished using methods of the `String` class.
- Iterating through the `WordFreq` items for output just requires using the iteration tools of whatever data structure we choose for our container. Creating the output does require the output of "`WordFreq` information" so we decide to expand the set of `WordFreq` operations to include a `toString` operation—we add this to its CRC card.

That leaves one section of the algorithm for more careful analysis, the section dealing with the container. Let's look at it again and decide exactly what container class to use.

```
if wordToTry is already in the container
   wordInTree = container.retrieve(wordToTry)
   Increment the frequency of wordInTree
   Save wordInTree back into the container
else
   Insert wordToTry into the container
```

The first thing we notice is the repeated access to the container required for each word. Potentially we have to check the container to see if the word is already there, retrieve the word from the container, and save the word back into the container. Ignoring the fact that we are not sure what "save back" means for now, we realize that we should consider efficient access to the container a high priority. Our input files could have thousands of words, so the underlying structure can be large. The need for repeated access and searching in a large structure leads us to choose the binary search tree for our container. This decision means that the `WordFreq` class must implement the `Comparable` interface, so that we can store `WordFreq` objects on our tree. We should update the CRC card for `WordFreq` accordingly.

Now let's address the "save back" question. The only way we can "save" information in our tree is to insert it; and our `insert` operation assumes that no current element matches the item being inserted. So, to "save back" the `WordFreq` object we would first have to delete the previous version of the object and then insert the new version. But, wait a minute. Do we really have to do that? Remember that our tree stores objects "by reference." When we retrieve a `WordFreq` object from the tree, we are actually retrieving a reference to the object. If we use that reference to access the object to increment its frequency count, the frequency count of the object in the tree is incremented. We do not have to "save back" the object at all!

In our discussion of the perils of "store by reference" in the feature section in Chapter 4, we stated that it is dangerous for the client to use a reference to reach into a data structure hidden by an ADT and change a data element. However, we also said this is dangerous only if the change affects the parts of the element used to determine the underlying physical relationship of the structure. In this case, the structure is based on the word information of a `WordFreq` object; we are changing the frequency information. We can reach into the tree and

increment the frequency count of one of its elements without affecting the tree's structure. So, we can reduce this part of our algorithm to

```
if wordToTry is already in the container
   wordInTree = container.retrieve(wordToTry)
   Increment the frequency of wordInTree
else
   Insert wordToTry into the container
```

Can we do anything else to increase the efficiency of the algorithm? To check "if the current word is already in the container" we need to use the Binary Search Tree's `isThere` operation. If it returns `true`, we immediately have to use the `retrieve` operation to obtain the reference to the object. Hmmm. We know that the first thing the `retrieve` operation does is search the tree to find the item to return … but we just searched the tree using the `isThere` operation. This seems wasteful, especially since we expect our text files have many repeated words, and therefore this sequence of operations has to be repeated many times.

This brings up a very important point: There are times when it is not appropriate to use an off-the-shelf container class. Using library classes—whether provided by Java or your own—allows you to write more reliable software in a shorter amount of time. These classes are already tested and debugged. If they fit the needs of your problem, use them. If they do not, then write a special-purpose method to do the job. In this case, we can extend the Binary Search Tree class with a new method that suits the purposes of our current problem. Our new method, `find`, combines the functionality of the `isThere` and `retrieve` methods. It searches the tree for an element matching its parameter item. If it finds a matching element, it returns a reference to it (just like `retrieve` would do). If it doesn't find a matching element, it returns the `null` reference. This way the client can determine whether or not the item "is there" by checking whether or not the return value is `null`. If the item is there, the client can use the returned reference to access the item. We call our extension of the `BinarySearchTree` class `BinarySearchTree2`.

With these insights, and changes, the subsection of our algorithm becomes:

```
wordInTree = container.find(wordToTry)
if (wordInTree == null)
   Insert wordToTry into the container
else
   Increment the frequency of wordInTree
```

We have reduced the number of times the tree is "searched" in order to handle a word that is already in the tree to 1.

The WordFreq Class The code for the `WordFreq` class is very straightforward. It is placed in the package `ch08.wordFreqs`. A few observations are appropriate. The code corresponding to these points is emphasized in the listing below.

- The constructor initializes the `freq` variable to 0. This means that the main program must increment a `WordFreq` object before placing it on the tree for the first time. We

could have coded the constructor to set the original frequency to 1, but we think it is more natural to begin a frequency count at 0. There may be other applications that can use `WordFreq` where this would be important.

- Java's `DecimalFormat` class allows us to force the string, used for the frequency count by the `toString` method, to be at least five characters wide. This helps line up output information for applications such as our Word Frequency Generator.

```
//--------------------------------------------------------------------------
// WordFreq.java              by Dale/Joyce/Weems                  Chapter 8
//
// Defines word-frequency pairs
//--------------------------------------------------------------------------

package ch08.wordFreqs;

import java.text.DecimalFormat;

public class WordFreq implements Comparable
{
  private String word;
  private int freq;

  DecimalFormat fmt = new DecimalFormat("00000");

  public WordFreq(String newWord)
  {
    word = newWord;
    freq = 0;
   }

  public void inc()
  {
    freq = freq + 1;
  }

  public int compareTo(Object otherWordFreq)
  {
   WordFreq other = (WordFreq)otherWordFreq;
   return this.word.compareTo(other.word);
  }

  public String toString()
  {
    return(fmt.format(freq) + " " + word);
  }
```

```
  public String wordIs()
  {
    return word;
  }

  public int freqIs()
  {
    return freq;
  }
}
```

The BinarySearchTree2 Class This class extends our current binary search tree class, adding the `find` method identified in the scenario analysis section above. Our program can instantiate objects of the class `BinarySearchTree2` and have access to all of the capabilities of our Binary Search Tree ADT plus the new Find operation. The `find` method calls the recursive `recFind` method that does most of the work.

```
//-----------------------------------------------------------------------
// BinarySearchTree2.java          by Dale/Joyce/Weems              Chapter 8
//
// Adds a method 'find' to the BinarySearchTree ADT
//-----------------------------------------------------------------------

package ch08.trees;

public class BinarySearchTree2 extends BinarySearchTree
{
  private Comparable recFind(Comparable item, BSTNode tree)
  // Returns reference to tree element that matches item
  // If no match exists, returns null
  {
    if (tree == null)
      return null;                              // Item is not found
    else if (item.compareTo(tree.info) < 0)
      return recFind(item, tree.left);          // Search left subtree
    else if (item.compareTo(tree.info) > 0)
      return recFind(item, tree.right);         // Search right subtree
    else
      return tree.info;                         // Item is found
  }

  public Comparable find (Comparable item)
  // Returns reference to this tree's element that matches item
  // If no match exists, returns null
```

```
  {
    return recFind(item, root);
  }
}
```

The Word Frequency Generator Program The main program is provided by the `FrequencyList` class. It follows the same input/output pattern as our test drivers. The main processing implements the algorithm developed in the Scenario Analysis section above.

```
//--------------------------------------------------------------------------
// FrequencyList.java            by Dale/Joyce/Weems                Chapter 8
//
// Creates a word frequency list of the words listed in the input file
// Writes the list to the output file
// Does not process words less than minSize in length
// Does not output words unless their frequency is at least minFreq
// Command-line parameters are assumed valid, as follows:
//   first: input file name
//   second: output file name
//   third: value of minSize
//   four: value of minFreq
//--------------------------------------------------------------------------

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.util.StringTokenizer;
import ch08.trees.*;
import ch08.wordFreqs.*;

public class FrequencyList
{
  private static Frame outputDisplay;

  public static void main(String[] args) throws IOException
  {
    String inLine = null;
    String word;
    WordFreq wordToTry;
    WordFreq wordInTree;
    WordFreq wordFromTree;

    BinarySearchTree2 tree = new BinarySearchTree2();
    StringTokenizer tokenizer;
```

```
int numWords = 0;
int numValidWords = 0;
int numValidFreqs = 0;
int treeSize;

// Get file name arguments from command line as entered by user
String dataFileName = args[0];
String outFileName  = args[1];

BufferedReader dataFile = new BufferedReader(new FileReader(dataFileName));
PrintWriter outFile = new PrintWriter(new FileWriter(outFileName));

// Get word and freq limits from command line as entered by user
int minSize = Integer.parseInt(args[2]);
int minFreq = Integer.parseInt(args[3]);

inLine = dataFile.readLine();
while (inLine != null)
{
  tokenizer = new StringTokenizer(
                        inLine, " \t\n\r\\\"![at]#$&*()_-+={}[]:;'<,>.?/");
  while (tokenizer.hasMoreTokens())
  {
    word = tokenizer.nextToken();
    numWords = numWords + 1;
    if (word.length() >= minSize)
    {
      numValidWords = numValidWords + 1;
      word = word.toLowerCase();
      wordToTry = new WordFreq(word);

      wordInTree = (WordFreq)tree.find(wordToTry);
      if (wordInTree == null)
      {
        // Insert new word into tree
        wordToTry.inc();               // Set frequency to 1
        tree.insert(wordToTry);
      }
      else
      {
        // Word already in tree; just increment frequency
        wordInTree.inc();
      }
    }
  }
  inLine = dataFile.readLine();
}
```

```
treeSize = tree.reset(BinarySearchTree.INORDER);
outFile.println("The words of length " + minSize + " and above,");
outFile.println("with frequency counts of " + minFreq + " and above:");
outFile.println();
outFile.println("Freq  Word");
outFile.println("---- -------");
for (int count = 1; count <= treeSize; count++)
{
  wordFromTree = (WordFreq) tree.getNextItem(BinarySearchTree.INORDER);
  if (wordFromTree.freqIs() >= minFreq)
  {
    numValidFreqs = numValidFreqs + 1;
    outFile.println(wordFromTree);
  }
}

// Close files
dataFile.close();
outFile.close();

// Set up output frame
JFrame outputFrame = new JFrame();
outputFrame.setTitle("Frequency List Generator");
outputFrame.setSize(400,100);
outputFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// Instantiate content pane and information panel
Container contentPane = outputFrame.getContentPane();
JPanel infoPanel      = new JPanel();

// Set layout
infoPanel.setLayout(new GridLayout(4,1));

// Create labels
JLabel numWordsInfo = new JLabel(numWords + " words in the input file.  ");
JLabel numValidWordsInfo =
  new JLabel(numValidWords + " of them are at least " + minSize +
             " characters.");
JLabel numValidFreqsInfo =
  new JLabel(numValidFreqs + " of these occur at least " + minFreq +
             " times.");
JLabel finishedInfo =
  new JLabel("Program completed. Close window to exit program.");

// Add information
infoPanel.add(numWordsInfo);
```

```
        infoPanel.add(numValidWordsInfo);
        infoPanel.add(numValidFreqsInfo);
        infoPanel.add(finishedInfo);
        contentPane.add(infoPanel);

        // Show information
        outputFrame.show();
    }
}
```

Testing This program should first be tested using small files, where it is easy for us to determine the expected output. The fact that the parameters for the program are passed through command-line arguments makes it easy for us to test the program on a series of input files, with varying minimum word sizes and frequency counts. Figure 8.24 shows the results of

```
8
Binary Search Trees
GOALS
Measurable goals for this chapter include that
you should be able to

* define and use the following terminology:
o binary tree
o binary search tree
o root
o parent
o child
o ancestor
o descendant
o level

etc.
```

File: chapter8.txt

```
The words of length 12 and above,
with frequency counts of 7 and above:

Freq  Word
----- ----------------
00020 binarysearchtree
00010 bstinterface
00009 hippopotamus
00018 implementation
00024 numberofnodes
00008 postcondition
00033 recnumberofnodes
00008 serializable
00007 specification
```

File: ch8out.txt

Screen

Command: java FrequencyList chapter8.txt ch8out.txt 12 7

Figure 8.26 *Example of a run of the Word Frequency Generator program*

running the program on a text file version of this chapter of the textbook, at least the current draft of this chapter. The minimum word size was set to 12 and the minimum frequency count was set to 7. Note that the output file contains the word "hippopotamus"! "Hippopotamus" is a strange word to occur so often in a computing chapter.

Summary

In this chapter we have seen how the binary tree may be used to structure sorted information to reduce the search time for any particular element. For applications in which direct access to the elements in a sorted structure is needed, the binary search tree is a very useful data type. If the tree is balanced, we can access any node in the tree with an $O(\log_2 N)$ operation. The binary search tree combines the advantages of quick random-access (like a binary search on a linear list) with the flexibility of a linked structure.

We also saw that the tree operations could be implemented very elegantly and concisely using recursion. This makes sense, because a binary tree is itself a "recursive" structure: Any node in the tree is the root of another binary tree. Each time we moved down a level in the tree, taking either the right or left path from a node, we cut the size of the (current) tree in half, a clear case of the smaller-caller.

We also discussed a tree balancing approach and a structuring approach that uses arrays. Finally, we presented a case study that used an extension of our Binary Search Tree ADT.

Summary of Classes and Support Files

The classes and files are listed in the order in which they appear in the text. The package a class belongs to, if any, is listed in parenthesis under Notes. The class and support files are available on our web site. They can be found in the `ch08` subdirectory of the `bookFiles` directory.

Classes, Interfaces, and Support Files Defined in Chapter 8

File	1st Ref.	Notes
`BSTInterface.java`	page 541	(`ch08.trees`) Specifies our Binary Search Tree ADT
`BinarySearchTree.java`	page 544	(`ch08.trees`) Reference-based implementation of our Binary Search Tree
`TDBinarySearchTree.java`	page 572	Test driver for `BinarySearchTree`
`WordFreq.java`	page 591	(`ch08.wordFreqs`) Used to hold word-frequency pairs for the case study
`BinarySearchTree2.java`	page 592	(`ch08.trees`) Extends `BinarySearchTree` with a `find` method, for use in the case study
`FrequencyList.java`	page 593	The Word-Frequency Generator program from the case study

Below is a list of the Java Library Interface that was used in this chapter for the first time in the textbook. For more information about the library classes and methods the reader can study Sun's Java documentation.

Library Classes Used in Chapter 8 for the First Time

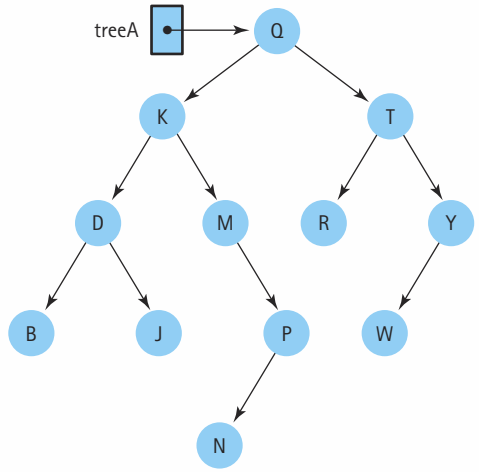
Class/Interface Name	Package	Overview	Methods Used	Where Used
`Comparable.java`	`lang`	Objects of classes that implement this interface can be compared to each other	`compareTo`	Tree classes

Exercises

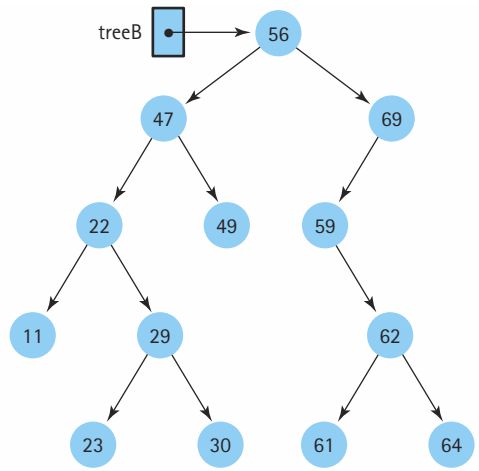
8.1 Trees

1. Binary Tree Levels:
   a. What does the level of a binary search tree mean in relation to the searching efficiency?
   b. What is the maximum number of levels that a binary search tree with 100 nodes can have?
   c. What is the minimum number of levels that a binary search tree with 100 nodes can have?
2. Which of these formulas gives the maximum total number of nodes in a tree that has N levels? (Remember that the root is Level 0.)
   a. $N^2 - 1$
   b. 2^N
   c. $2^{N+1} - 1$
   d. 2^{N+1}
3. Which of these formulas gives the maximum number of nodes in the Nth level of a binary tree?
   a. N^2
   b. 2^N
   c. 2^{N+1}
   d. $2^N - 1$
4. How many ancestors does a node in the Nth level of a binary search tree have?
5. How many different *binary trees* can be made from three nodes that contain the key values 1, 2, and 3?

6. How many different *binary search trees* can be made from three nodes that contain the key values 1, 2, and 3?
7. Draw all the possible binary trees that have four leaves and all the nonleaf nodes that have two children.



8. Answer the following questions about treeA.
   a. What are the ancestors of node P?
   b. What are the descendants of node K?
   c. What is the maximum possible number of nodes at the level of node W?
   d. What is the maximum possible number of nodes at the level of node N?
   e. What is the order in which the nodes are visited by an inorder traversal?
   f. What is the order in which the nodes are visited by a preorder traversal?
   g. What is the order in which the nodes are visited by a postorder traversal?



9. Answer the following questions about treeB.
   a. What is the height of the tree?
   b. What nodes are on level 3?
   c. Which levels have the maximum number of nodes that they could contain?
   d. What is the maximum height of a binary search tree containing these nodes? Draw such a tree.
   e. What is the minimum height of a binary search tree containing these nodes? Draw such a tree.
   f. What is the order in which the nodes are visited by an inorder traversal?
   g. What is the order in which the nodes are visited by a preorder traversal?
   h. What is the order in which the nodes are visited by a postorder traversal?
10. True or False?
   a. A preorder traversal of a binary search tree processes the nodes in the tree in the exact reverse order that a postorder traversal processes them.
   b. An inorder traversal of a binary search tree always processes the elements of the tree in the same order, regardless of the order in which the elements were inserted.
   c. A preorder traversal of a binary search tree always processes the elements of the tree in the same order, regardless of the order in which the elements were inserted.

8.2 The Logical Level

11. Describe the differences between our specifications of the Sorted List ADT and the Binary Search Tree ADT.
12. Suppose you decide to change our Binary Search Tree to allow duplicate elements. How would you have to change the Binary Search Tree specifications?
13. Our binary search trees hold elements of type `Comparable`. What would be the consequences of changing this to the type `Listable`?
14. List six Java Library classes that implement the `Comparable` interface. (The answer is not in this textbook—it requires research!)
15. Lots of preconditions are stated for the Binary Search Tree operations defined in the `BSTInterface` interface. Describe an alternative approach to using all of these preconditions.

8.3 The Application Level

16. Write a client method that returns a count of the number of nodes of a binary search tree that contain a value less than or equal to the parameter value. The signature of the method is:

```
int countLess(BinarySearchTree tree, Comparable maxValue)
```

17. Write a client method that returns a reference to the information in the node with the "smallest" value in a binary search tree. The signature of the method is:

```
Comparable min(BinarySearchTree tree)
```

18. Write a client method that returns a reference to the information in the node with the "largest" value in a binary search tree. The signature of the method is:

```
Comparable max(BinarySearchTree tree)
```

8.4 The Implementation Level—Declarations and Simple Operations

19. Extend the Binary Search Tree ADT to include a public method `leafCount` that returns the number of leaf nodes in the tree.
20. Extend the Binary Search Tree ADT to include a public method `singleParentCount` that returns the number of nodes in the tree that have only one child.
21. The Binary Search Tree ADT is extended to include a `boolean` method `similarTrees` that receives references to two binary trees and determines if the shapes of the trees are the same. (The nodes do not have to contain the same values, but each node must have the same number of children.)
    a. Write the declaration of method `similarTrees`. Include adequate comments.
    b. Write the body of method `similarTrees`.

8.5 Iterative Versus Recursive Method Implementations

22. Use the Three-Question Method to verify the recursive version of the `numberOfNodes` method.
23. We need a public method for our Binary Search Tree ADT that returns a reference to the information in the node with the "smallest" value in the tree. The signature of the method is:

```
public Comparable min()
```

    a. Design an iterative version of the method.
    b. Design a recursive version of the method.
    c. Which approach is better? Explain.

24. We need a public method for our Binary Search Tree ADT that returns a reference to the information in the node with the "largest" value in the tree. The signature of the method is:

```
public Comparable max()
```

    a. Design an iterative version of the method.
    b. Design a recursive version of the method.
    c. Which approach is better? Explain.

25. We need a public method for our Binary Search Tree ADT that returns a count of the number of nodes of the tree that contain a value less than or equal to the parameter value. The signature of the method is:

```
public int countLess(Comparable maxValue)
```

    a. Design an iterative version of the method.
    b. Design a recursive version of the method.
    c. Which approach is better? Explain.

8.6 The Implementation Level—More Operations

26. The `BinarySearchTree` class used a queue as an auxiliary storage structure for iterating through the elements in the tree. Discuss the relative merits of using a dynamically allocated array-based queue versus a dynamically allocated linked queue.
27. Show what treeA (page 599) would look like after each of the following changes. (Use the original tree to answer each part.)
    a. Add node C.
    b. Add node Z.
    c. Add node X.
    d. Delete node M.
    e. Delete node Q.
    f. Delete node R.
28. Draw the binary search tree whose elements are inserted in the following order:

    50 72 96 94 107 26 12 11 9 2 10 25 51 16 17 95

Exercises 29–31 use treeB (page 599).

29. Trace the path that would be followed in searching for
    a. a node containing 61.
    b. a node containing 28.
30. Show how treeB would look after the deletion of 29, 59, and 47.
31. Show how the (original) treeB would look after the insertion of nodes containing 63, 77, 76, 48, 9, and 10 (in that order).
32. The key of each node in a binary search tree is a short character string.
    a. Show how such a tree would look after the following words were inserted (in the order indicated):

    "hippopotamus" "canary" "donkey" "deer" "zebra" "yak" "walrus" "vulture" "penguin" "quail"

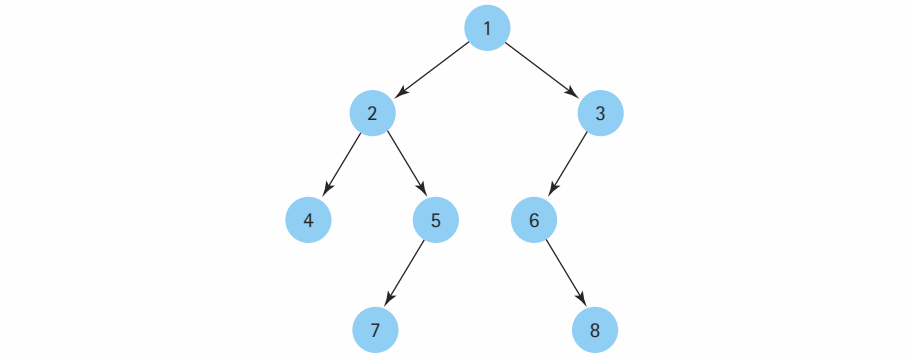
b. Show how the tree would look if the same words were inserted in this order:

"quail" "walrus" "donkey" "deer" "hippopotamus" "vulture" "yak" "penguin" "zebra" "canary"

c. Show how the tree would look if the same words were inserted in this order:

"zebra" "yak" "walrus" "vulture" "quail" "penguin" "hippopotamus" "donkey" "deer" "canary"

Examine the following binary search tree and answer the questions in Exercises 33–36. The numbers on the nodes are labels so that we can talk about the nodes; they are not key values within the nodes.



33. If an item is to be inserted whose key value is less than the key value in node 1 but greater than the key value in node 5, where would it be inserted?
34. If node 1 is to be deleted, the value in which node could be used to replace it?
35. 4 2 7 5 1 6 8 3 is a traversal of the tree in which order?
36. 1 2 4 5 7 3 6 8 is a traversal of the tree in which order?

8.7 Comparing Binary Search Trees to Linear Lists

37. One hundred integer elements are chosen at random and inserted into a sorted linked list and a binary search tree. Describe the efficiency of searching for an element in each structure, in terms of Big-O.
38. One hundred integer elements are inserted in order, from smallest to largest, into a sorted linked list and a binary search tree. Describe the efficiency of searching for an element in each structure, in terms of Big-O.
39. Write a client `boolean` method `matchingItems` that determines if a binary search tree and a sorted list contain the same values. Assume that the tree and the list both store `Listable` elements. The signature of the method is:

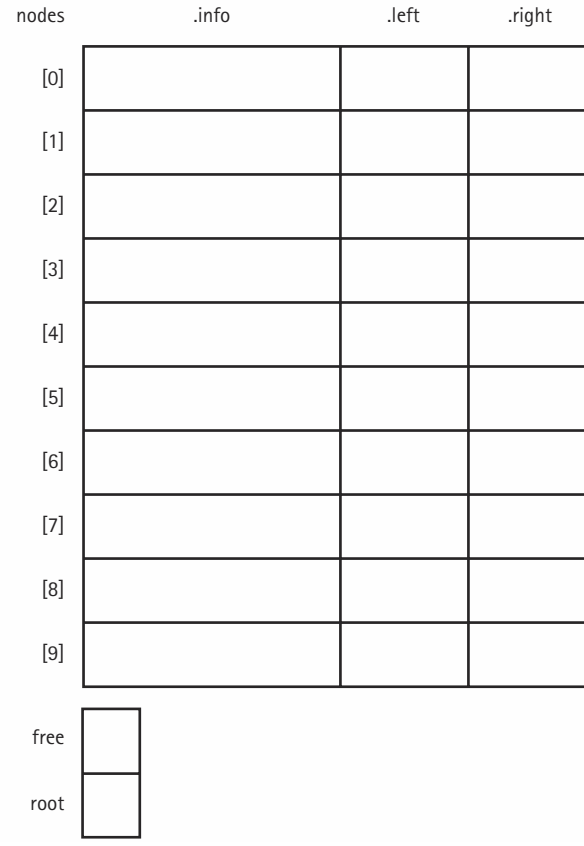
```
boolean matchingItems(BinarySearchTree tree, SortedList list)
```

40. In Chapter 6 we discussed how a linked list could be stored in an array of nodes using index values as "references" and managing our list of free nodes. We can use these same techniques to store the nodes of a binary search tree in an array, rather than using dynamic storage allocation. Free space is linked through the left member.

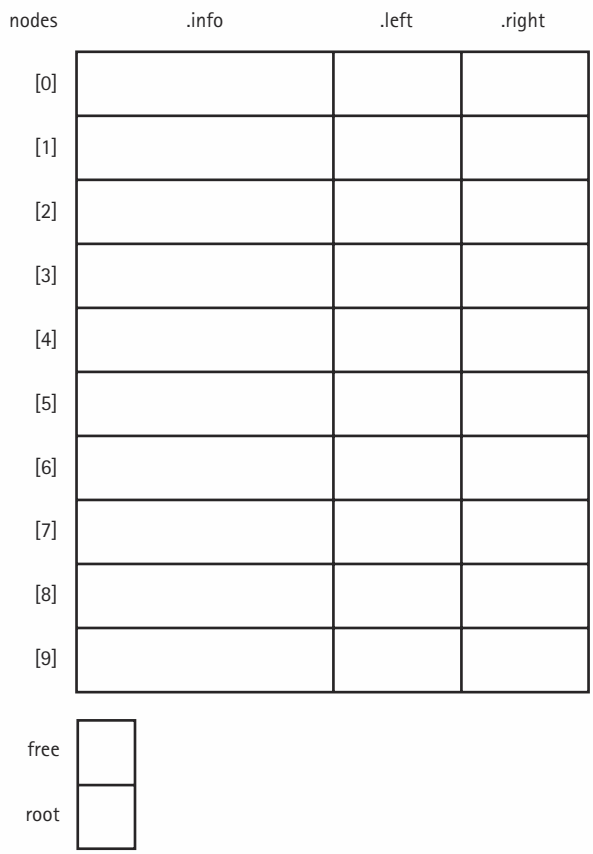
    a. Show how the array would look after these elements had been inserted in this order:

    Q L W F M R N S

    Be sure to fill in all the spaces. If you do not know the contents of a space, use '?'.



    b. Show the contents of the array after 'B' has been inserted and 'R' has been deleted.



8.8 Balancing a Binary Search Tree

41. Show the tree that would result from storing the nodes of the tree in Figure 8.20(a) in postorder order into an array, and then traversing the array in index order while inserting the nodes into a new tree.
42. Using the Balance algorithm, show the tree that would be created if the following values represented the inorder traversal of the original tree
    a. 3 6 9 15 17 19 29
    b. 3 6 9 15 17 19 29 37
43. Revise our `BSTInterface` interface and `BinarySearchTree` class to include the `balance` method. How can you test your revision?

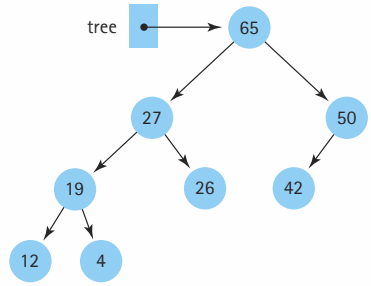
8.9 A Nonlinked Representation of Binary Trees

44. Consider the following trees.
    a. Which fulfill the binary search tree property?

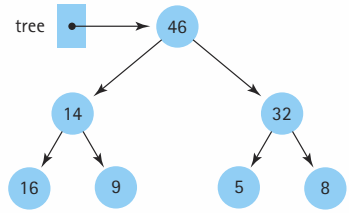
b. Which are complete?

c. Which are full?

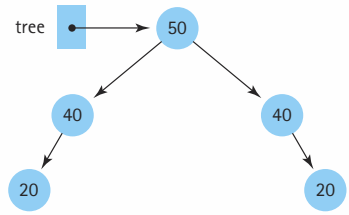
(a)



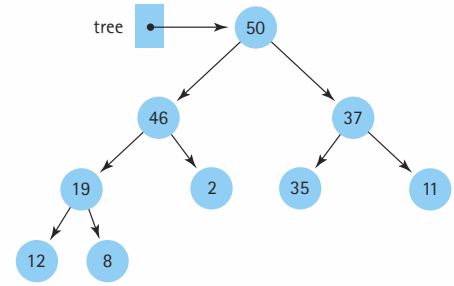
(b)



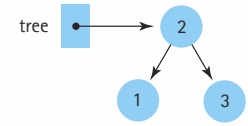
(c)



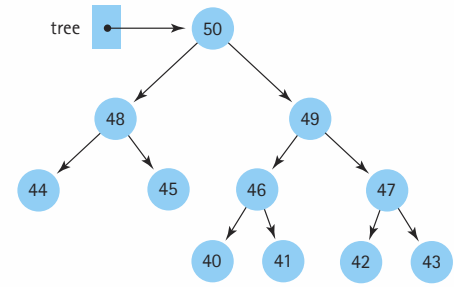
(d)



(e)



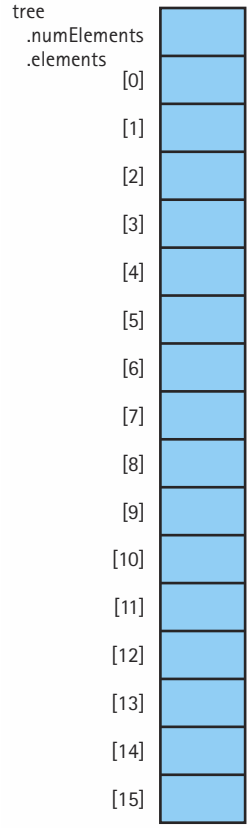
(f)

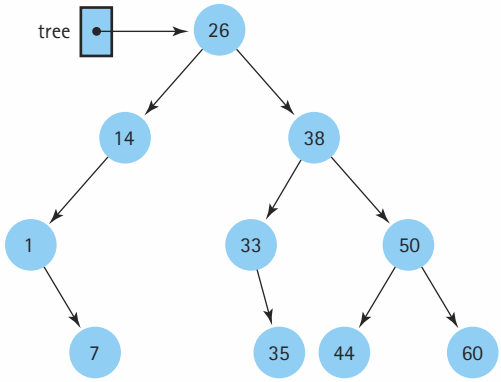


45. The elements in a binary tree are to be stored in an array, as described in the section. Each element is a nonnegative `int` value.

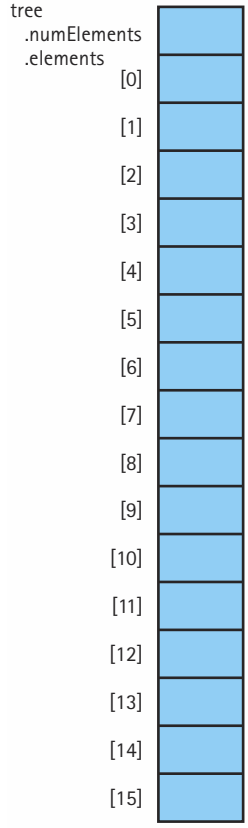
    a. What value can you use as the dummy value, if the binary tree is not complete?

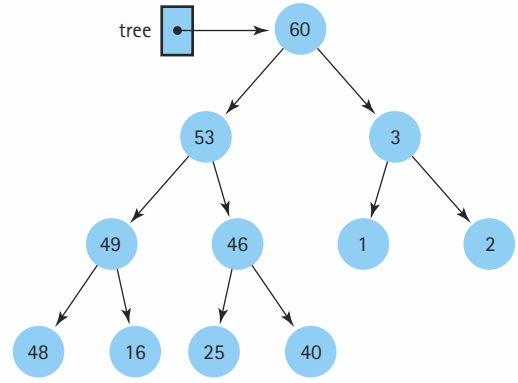
    b. Show the contents of the array, given the tree illustrated below.



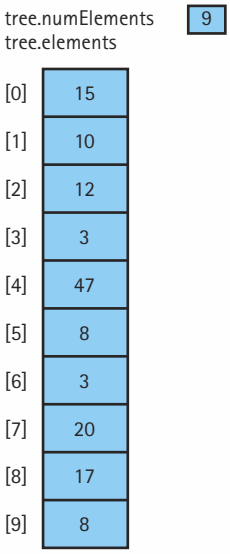


46. The elements in a complete binary tree are to be stored in an array, as described in the section. Each element is a nonnegative `int` value. Show the contents of the array, given the tree illustrated below.





47. Given the array pictured below, draw the binary tree that can be created from its elements. (The elements are arranged in the array as discussed in the section.)



48. A complete binary tree is stored in an array called `treeNodes`, which is indexed from 0 to 99, as described in the section. The tree contains 85 elements. Mark each of the following statements as True or False, and explain your answers.

- **a.** `treeNodes[42]` is a leaf node.
- **b.** `treeNodes[41]` has only one child.
- **c.** The right child of `treeNodes[12]` is `treeNodes[25]`.
- **d.** The subtree rooted at `treeNodes[7]` is a full binary tree with four levels.
- **e.** The tree has seven levels that are full, and one additional level that contains some elements.

8.10 Word Frequency Generator—A Case Study

49. You wouldn't expect to find the word "hippopotamus" very often in a computer book. After all, a hippopotamus is an animal, not a data structure. Yet the word "hippopotamus" does appear many times in this chapter alone. How many times does the word "hippopotamus" appear in this chapter?

50. We want the Word Frequency Generator program to output one additional piece of information, the number of unique words in the input file.
    a. Describe two separate ways you could solve this problem, that is, two ways to handle the additional words the program now must track.
    b. Which approach do you believe is better? Why?
    c. Implement the change.
51. Design and create a graphical user interface for the Word Frequency Generator using
    a. The graphical components used in previous case studies
    b. Use `JSlider` objects to obtain the word size and frequency minimums from the user.
    c. Use `JFileChooser` objects to obtain the input and output file names from the user.

Chapter 9

Priority Queues, Heaps, and Graphs

Goals

Measurable goals for this chapter include that you should be able to

- describe a priority queue at the logical level and discuss alternate implementation approaches
- define a heap and the operations reheap up and reheap down
- implement a priority queue as a heap
- describe the shape and order properties of a heap, and implement a heap using a nonlinked tree representation of an array
- compare the implementations of a priority queue using a heap, linked list, and binary search tree
- define the following terms related to graphs:

  directed graph
  undirected graph
  vertex
  edge
  path
  complete graph
  weighted graph
  adjacency matrix
  adjacency list
- implement a graph using an adjacency matrix to represent the edges
- explain the difference between a depth-first and a breadth-first search and implement these searching strategies using stacks and queues for auxiliary storage
- implement a shortest-paths operation, using a priority queue to access the edge with the minimum weight
- save an object or structure to a file from one program and retrieve it for use in another program

So far, we have examined several basic data structures in depth, discussing their uses and operations, as well as one or more implementations of each. As we have constructed these programmer-defined data structures out of the built-in types provided by our high-level language, we have noted variations that adapt them to the needs of different applications. In Chapter 8 we looked at how a tree structure, the binary search tree, facilitates searching data stored in a linked structure. In this chapter we see how other branching structures are defined and implemented to support a variety of applications.

9.1 Priority Queues

A priority queue is an abstract data type with an interesting accessing protocol. Only the *highest-priority* element can be accessed. "Highest priority" can mean different things, depending on the application. Consider, for example, a small company with one secretary. When employees leave work on the secretary's desk, which jobs get done first? The jobs are processed in order of the employee's importance in the company; the secretary completes the president's work before starting the vice-president's, and does the marketing director's work before the work of the staff programmers. The *priority* of each job relates to the level of the employee who initiated it.

In a telephone answering system, calls are answered in the order that they are received; that is, the highest-priority call is the one that has been waiting the longest. Thus, a FIFO queue can be considered a priority queue whose highest-priority element is the one that has been queued the longest time.

Sometimes a printer shared by a number of computers is configured to always print the smallest job in its queue first. This way, someone who is only printing a few pages does not have to wait for large jobs to finish. For such printers, the priority of the jobs relates to the size of the job; shortest job first.

Priority queues are useful for any application that involves processing items by priority.

Logical Level

The operations defined for the Priority Queue ADT include enqueing items and dequeing items, as well as testing for an empty or full priority queue. These operations are very similar to those specified for the FIFO queue discussed in Chapter 4. The `enqueue` operation adds a given element to the priority queue. The `dequeue` operation removes the highest-priority element from the priority queue and returns it to the user. The difference is that the Priority Queue does not follow the "first in, first out" approach; the Priority Queue always returns the highest priority item from the current set of enqueued items, no matter when it was enqueued. Here is the specification, as a Java interface named `PriQueueInterface` (note that it is in a package called `ch09.priorityQueues`.)

```
//--------------------------------------------------------------------------
// PriQueueInterface.java          by Dale/Joyce/Weems                Chapter 9
//
// Interface for a class that implements a priority queue of Comparable Objects
//--------------------------------------------------------------------------
```

```
package ch09.priorityQueues;

public interface PriQueueInterface
// Interface for a class that implements a priority queue of Comparable Objects
{
  public boolean isEmpty();
  // Effect:         Determines whether this priority queue is empty
  // Postcondition:  Return value = (this priority queue is empty)

  public boolean isFull();
  // Effect:         Determines whether this priority queue is full
  // Postcondition:  Return value = (priority queue is full)

  public void enqueue(Comparable item);
  // Effect:         Adds item to this priority queue
  // Postconditions: If (this priority queue is full)
  //                   an unchecked exception that communicates 'enqueue
  //                   on priority queue full' is thrown
  //                 Else
  //                   item is in this priority queue

  public Comparable dequeue();
  // Effect:         Removes element with highest priority from this
  //                   priority queue and returns a reference to it
  // Postconditions: If (this priority queue is empty)
  //                   an unchecked exception that communicates 'dequeue
  //                   on empty priority queue' is thrown
  //                 Else
  //                   Highest priority element has been removed.
  //                   Return value = (the removed element)
}
```

A few notes based on the specification:

- Our Priority Queues hold objects of type `Comparable`, just as our Binary Search Trees do. This allows us to rank the items by priority.
- Our Priority Queues can hold duplicate items, that is, items with the same key value.
- We implement Priority Queues "by reference." For example, note that the `enqueue` operation's effect is "adds item to this priority queue" and not "adds copy of item to this priority queue."
- Attempting to enqueue an item into a full priority queue, or dequeue an item from an empty priority queue, causes an unchecked exception to be thrown.

We define the exceptions using the standard approach established in Chapter 4. Here are the definitions of the two exception classes used by our priority queue class:

```
package ch09.priorityQueues;

class PriQUnderflowException extends RuntimeException
{
  public PriQUnderflowException()
  {
  }

  public PriQUnderflowException(String message)
  {
    super(message);
  }
}

package ch09.priorityQueues;

class PriQOverflowException extends RuntimeException
{
  public PriQOverflowException()
  {
  }

  public PriQOverflowException(String message)
  {
    super(message);
  }
}
```

Application Level

In discussing FIFO queue applications in Chapter 4, we said that the operating system of a multi-user computer system may use job queues to save user requests in the order in which they are made. Another way such requests may be handled is according to how important the job request is. That is, the head of the company might get higher priority than the junior programmer. Or an interactive program might get higher priority than a job to print out a report that isn't needed until the next day. To handle these requests efficiently, the operating system may use a priority queue.

Priority queues are also useful in sorting. Given a set of elements to sort, we can enqueue the elements into a priority queue, and then dequeue them in sorted order (from largest to smallest). We look more at how priority queues can be used in sorting in Chapter 10.

Implementation Level

There are many ways to implement a priority queue. In any implementation, we want to be able to access the element with the highest priority quickly and easily. Let's briefly consider some possible approaches:

An Unsorted List

Enqueing an item would be very easy. Simply insert it at the end of the list. However, dequeing would require searching through the entire list to find the largest element.

An Array-Based Sorted List

Dequeing is very easy with this approach. Simply return the last list element and reduce the size of the list; `dequeue` is a O(1) operation. Enqueing however would be more expensive; we have to find the place to enqueue the item ($O(\log_2 N)$ if we use binary search) and rearrange the elements of the list after removing the item to return ($O(N)$).

A Reference-Based Sorted List

Let's assume the linked list is kept sorted from largest to smallest. Dequeing simply requires removing and returning the first list element, an operation that only requires a few steps. But enqueing again is $O(N)$ since we must search the list one element at a time to find the insertion location.

A Binary Search Tree

For this approach, the `enqueue` operation is implemented as a standard binary search tree `insert` operation. We know that requires $O(\log_2 N)$ steps on average. Assuming we have access to the underlying implementation structure of the tree, we can implement the `dequeue` operation by returning the rightmost tree element. We follow the right subtree references down, maintaining a trailing reference as we go, until we reach a node with an empty right subtree. The trailing reference allows us to "unlink" the node from the tree. We then return the node. This is also a $O(\log_2 N)$ operation on average.

The binary tree approach is the best—it only requires, on average, $O(\log_2 N)$ steps for both `enqueue` and `dequeue`. However, if the tree is skewed the performance degenerates to $O(N)$ steps for each operation. In the next section we present an approach, called the heap, that guarantees $O(\log_2 N)$ steps, even in the worst case.

9.2 Heaps

A heap[1] is an implementation of a Priority Queue that uses a binary tree that satisfies two properties, one concerning its shape and the other concerning the order of its elements. The *shape property* is simply stated: the tree must be a complete binary tree (see Section 8.9). The *order property* says that, for every node in the tree, the value stored in that node is greater than or equal to the value in each of its children.

Heap An implementation of a Priority Queue based on a complete binary tree, each of whose elements contains a value that is greater than or equal to the value of each of its children

It might be more accurate to call this structure a "maximum heap," since the root node contains the maximum value in the structure. It is also possible to create a "minimum

[1] *Heap* is also a synonym for the free store of a computer—the area of memory available for dynamically allocated data. The heap as a data structure is not to be confused with this unrelated computer system concept of the same name.

heap," each of whose elements contains a value that is *less* than or equal to the value of each of its children. The term *heap* is used for both the abstract data type—the Priority Queue implementation—and for the underlying structure, the tree that fulfills the shape and order properties.

Figure 9.1 shows two trees containing the letters 'A' through 'J' that fulfill both the shape and order properties. Notice that the placement of the values differs in the two trees, but the shape is the same: a complete binary tree of ten elements. Note also that the two trees have the same root node. A group of values can be stored in a binary tree in many ways and still satisfy the order property of heaps. Because of the shape property, we know that the shape of all heap trees with a given number of elements is the same. We also know, because of the order property, that the root node always contains the

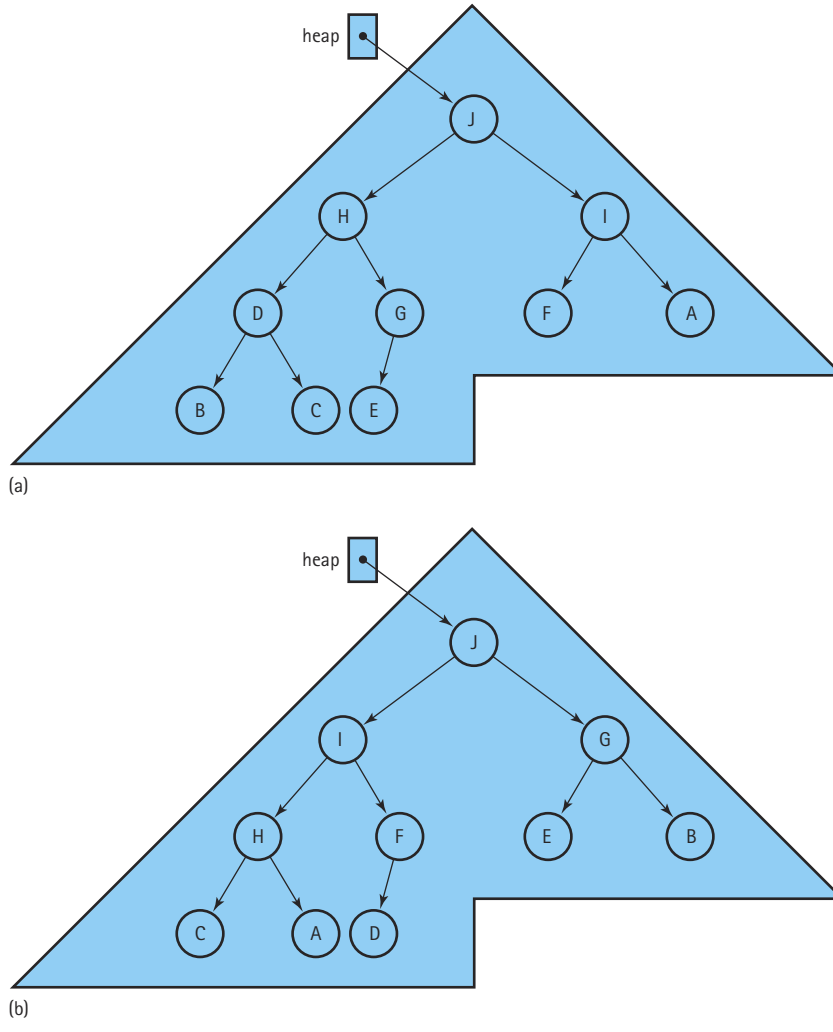


Figure 9.1 *Two heaps containing the letters 'A' through 'J'*

largest value in the tree. This helps us implement an efficient `dequeue` operation. Finally, note that every subtree of a heap is also a heap.

Let's say that we want to dequeue an element from the heap, in other words, we want to remove and return the element with the largest value from the tree. The largest element is in the root node, so we can easily remove it, as illustrated in Figure 9.2(a).

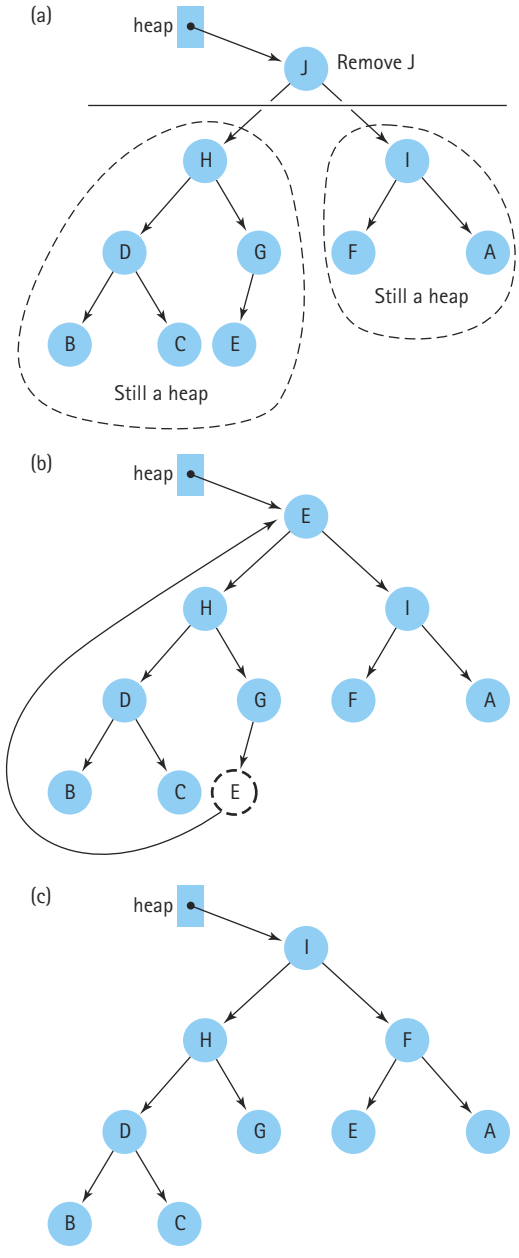


Figure 9.2 *The `reheapDown` operation*

But this leaves a hole in the root position. Because the heap's tree must be complete, we decide to fill the hole with the bottom rightmost element from the tree; now the structure satisfies the shape property (Figure 9.2b). However, the replacement value came from the bottom of the tree, where the smaller values are; the tree no longer satisfies the order property of heaps.

This situation suggests one of the standard heap-support operations. Given a binary tree that satisfies the heap properties, *except that the root position is empty*, insert an item into the structure so that it is again a heap. This operation, called `reheapDown`, involves starting at the root position and moving the "hole" (the empty position) down, while moving tree elements up, until finding a position for the hole where the item can be inserted (see Figure 9.2c). We say that we `swap` the hole with one of its children. The `reheapDown` operation has the following specification.

reheapDown (item)

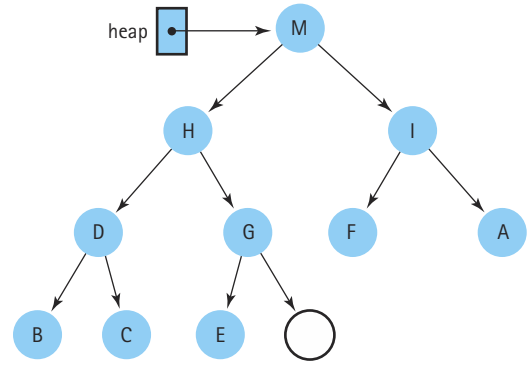
Effect:	Adds item to the heap.
Precondition:	The root of the tree is empty.
Postcondition:	item is in the heap.

To dequeue an element from the heap, we remove and return the root element, remove the bottom rightmost element, and then pass the bottom rightmost element to `reheapDown`, to restore the heap.

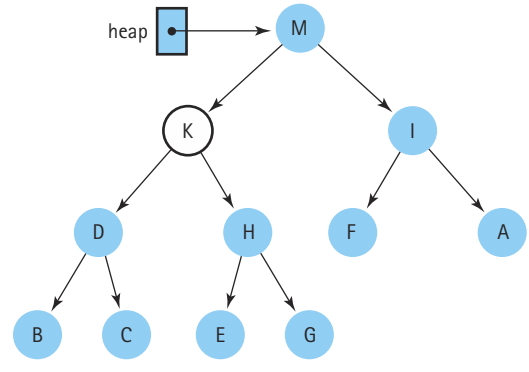
Now let's say that we want to enqueue an element to the heap—where do we put it? The shape property tells us that the tree must be complete, so we put the new element in the next bottom rightmost place in the tree, as illustrated in Figure 9.3(a). Now the shape property is satisfied, but the order property may be violated. This situation illustrates the need for another heap-support operation. Given a binary tree that satisfies the heap properties, except that the last position is empty, insert a given item into the structure so that it is again a heap. Instead of inserting the item in the next bottom rightmost position in the tree, we imagine we have another hole there. We then float the hole position up the tree, while moving tree elements down, until the hole is in a position (see Figure 9.3b) that allows us to legally insert the item. This operation is called `reheapUp`. Here is the specification.

reheapUp (item)

Effect:	Adds item to the heap.
Precondition:	The last index position of the tree is empty.
Postcondition:	item is on the heap.



(a) Add K



(b) reheapUp

Figure 9.3 *The* `reheapUp` *operation*

Heap Implementation

Although we have graphically depicted heaps as binary trees with nodes and links, it would be very impractical to implement the heap operations using the usual linked-tree representation. The shape property of heaps tells us that the binary tree is complete, so we know that it is never unbalanced. Thus, we can easily store the tree in an array with implicit links, as discussed in Section 8.9, without wasting any space. Figure 9.4 shows how the values in a heap would be stored in this array representation. If a heap with `numElements` elements is implemented this way, the shape property says that the heap elements are stored in `numElements` consecutive slots in the array, with the root element in the first slot (index 0) and the last leaf node in the slot with index `numElements - 1`.

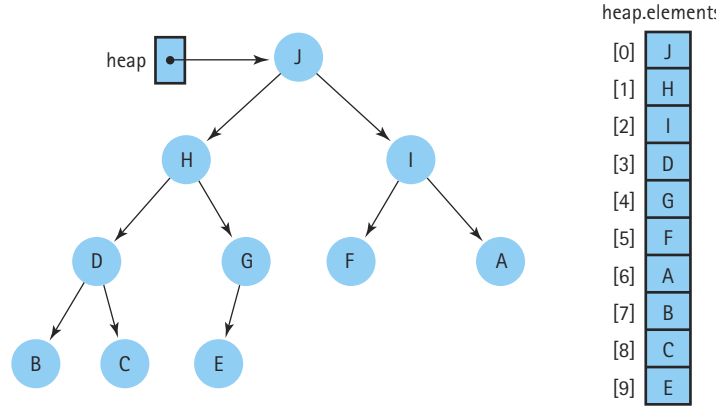


Figure 9.4 *Heap values in an array representation*

Recall that when we use this representation of a binary tree, the following relationships hold for an element at position `index`:

- If the element is not the root, its parent is at position `(index - 1) / 2`.
- If the element has a left child, the child is at position `(index * 2) + 1`.
- If the element has a right child, the child it is at position `(index * 2) + 2`.

These relationships allow us to efficiently calculate the parent, left child, or right child of any node! And since the tree is complete we do not waste space using the array representation. Time efficiency *and* space efficiency! We make use of these features in our heap implementation.

Here is the beginning of our `Heap` class. As you can see, it implements `PriQueueInterface`. Since it implements a priority queue, we placed it in the `ch09.priorityQueues` package. Also note that the only constructor requires an integer argument, used to set the size of the underlying array. The `isEmpty` and `isFull` operations are trivial.

```
//--------------------------------------------------------------------------
// Heap.java                by Dale/Joyce/Weems                    Chapter 9
//
// Defines all constructs for a heap of Comparable objects
// The dequeue method returns the largest value in the heap
//--------------------------------------------------------------------------

package ch09.priorityQueues;

public class Heap implements PriQueueInterface
{
  private Comparable[] elements;  // Array that holds priority queue elements
```

```
  private int lastIndex;          // Index of last element in priority queue
  private int maxIndex;           // Index of last position in array

  // Constructor
  public Heap(int maxSize)
  {
    elements = new Comparable[maxSize];
    lastIndex = -1;
    maxIndex = maxSize - 1;
  }

  public boolean isEmpty()
  // Determines whether this priority queue is empty
  {
    return (lastIndex == -1);
  }

  public boolean isFull()
  // Determines whether this priority queue is full
  {
    return (lastIndex == maxIndex);
  }
...
}
```

The enqueue Method

We next look at the `enqueue` method. It is the simpler of the two transformer methods. If we assume the existence of a `reheapUp` helper method, as specified previously, the `enqueue` method is:

```
public void enqueue(Comparable item) throws PriQOverflowException
// Adds item to this priority queue
// Throws PriQOverflowException if priority queue already full
{
  if (lastIndex == maxIndex)
    throw new PriQOverflowException("Priority queue is full");
  else
  {
    lastIndex = lastIndex + 1;
    reheapUp(item);
  }
}
```

If the array is already full, we throw the appropriate exception. Otherwise, we increase the `lastIndex` value and call the `reheapUp` method. Of course, the `reheapUp` method is doing all of the interesting work. Let's look at it more closely.

The `reheapUp` algorithm starts with a tree whose last node is empty; we continue to call this empty node the hole. We swap the hole up the tree until it reaches a spot where the `item` argument can be placed into the hole without violating the order property of the heap. While the hole moves up the tree, the elements it is replacing move down the tree, filling in the previous location of the hole. This is illustrated in Figure 9.5.

Note that the sequence of nodes between a leaf and the root of a heap can be viewed as a sorted linked list. This is guaranteed by the heap's order property. The `reheapUp` algorithm is essentially inserting an item into this sorted linked list. As we

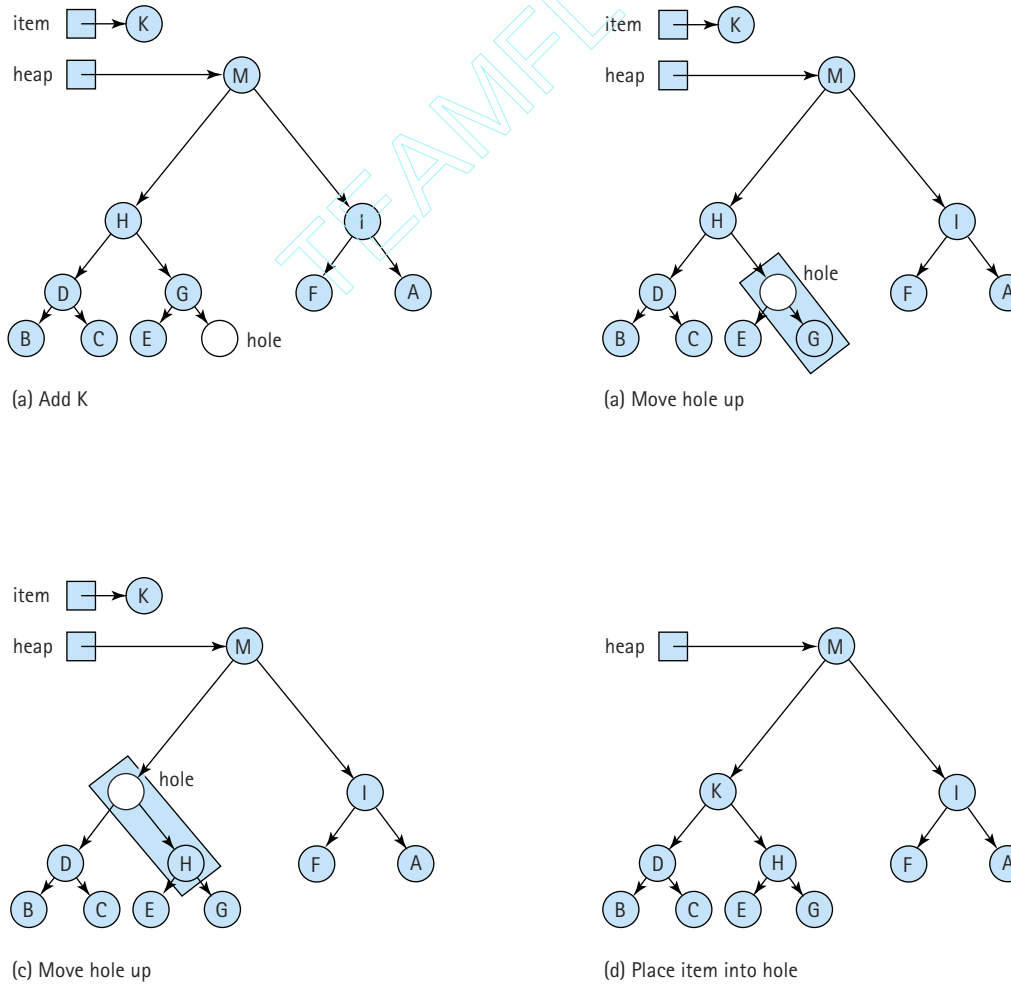


Figure 9.5 *The `reheapUp` operation in action*

progress from the leaf to the root along this path, we compare the value of `item` with the value in the hole's parent node. If the parent's value is smaller, we cannot place `item` into the current hole, since the order property would be violated, so we move the hole up. Moving the hole up really means copying the value of the hole's parent into the hole's location. Now the parent's location is available and it becomes the new hole. We repeat this process until (1) the hole is the root of the heap, or (2) `item`'s value is less than or equal to the value in the hole's parent node. In either case, we can now safely copy `item` into the hole's position.

Here's the algorithm:

reheapUp(item)

```
Set hole to lastIndex.
while (the hole is not the root) And (item > the hole's parent.info( ))
    Swap hole with hole's parent
Set hole.info to item
```

This algorithm requires us to be able to quickly find a given node's parent. This appears difficult, based on our experiences with references that can only be traversed in one direction. But, as we saw earlier, it is very simple with our implicit link implementation:

- If the element is not the root its parent is at position `(index - 1) / 2`.

Here is the code for the `reheapUp` method:

```
private void reheapUp(Comparable item)
// Current lastIndex position is empty
// Inserts item into the tree and maintains shape and order properties
{
  int hole = lastIndex;
  while ((hole > 0)                                          // Hole is not root
         &&                                                  // Short circuit
         (item.compareTo(elements[(hole - 1) / 2]) > 0))     // item > hole's
                                                             //  parent
  {
    elements[hole] = elements[(hole - 1) / 2];  // Move hole's parent down
    hole = (hole - 1) / 2;                      // Move hole up
  }
  elements[hole] = item;                        // Place item into final hole
}
```

This method takes advantage of the short circuit nature of Java's `&&` operator. If the current `hole` is the root of the heap then the first half of the while loop control expression

```
(hole > 0)
```

is `false` and the second half

```
(item.compareTo(elements[(hole - 1) / 2]) > 0))
```

is not evaluated. If it was evaluated in that case, it would cause a run-time "array access out of bounds" error.

The dequeue Method

Finally, we look at the `dequeue` method. As for `enqueue`, if we assume the existence of the helper method, in this case the `reheapDown` method, the `dequeue` method is very simple:

```
public Comparable dequeue() throws PriQUnderflowException
// Removes element with highest priority from this priority queue
//  and returns a reference to it
// Throws PriQUnderflowException if priority queue is empty
{
  Comparable hold;      // Item to be dequeued and returned
  Comparable toMove;    // Item to move down heap

  if (lastIndex == -1)
    throw new PriQUnderflowException("Priority queue is empty");
  else
  {
    hold = elements[0];             // Remember item to be returned
    toMove = elements[lastIndex];   // Item to reheap down
    lastIndex = lastIndex - 1;      // Decrease priority queue size
    reheapDown(toMove);             // Restore heap properties
    return hold;                    // Return largest element
  }
```

If the array is empty, we throw the appropriate exception. Otherwise, we first make a copy of the root element (the maximum element in the tree), so that we can return it to the client program when we are finished. We also make a copy of the element in the "last" array position. Recall that this is the element we use to move into the hole vacated by the root element, so we call it the `toMove` element. We decrement the `lastIndex` variable to reflect the new bounds of the array and pass the `toMove` element to the `reheapDown` method. If that method does its job, the only thing remaining to do is to return the saved value of the previous root element (`hold`) to the client.

Let's look at the `reheapDown` algorithm more closely. In many ways, it is similar to the `reheapUp` algorithm. In both cases, we have a "hole" in the tree and an `item` to be placed into the tree so that the tree remains a heap. In both cases, we move the hole through the tree (actually moving tree elements into the hole) until it reaches a location where it can legally hold the `item`. However, `reheapDown` is a more complex operation since it is moving the hole down the tree instead of up the tree. When we are moving down, there are more decisions for us to make.

When `reheapDown` is first called, the root of the tree can be considered a hole; that position in the tree is available, since the `dequeue` method has already saved the contents in its `hold` variable. The job of `reheapDown` is to "move" the hole down the tree until it reaches a spot where `item` can replace it. See Figure 9.6.

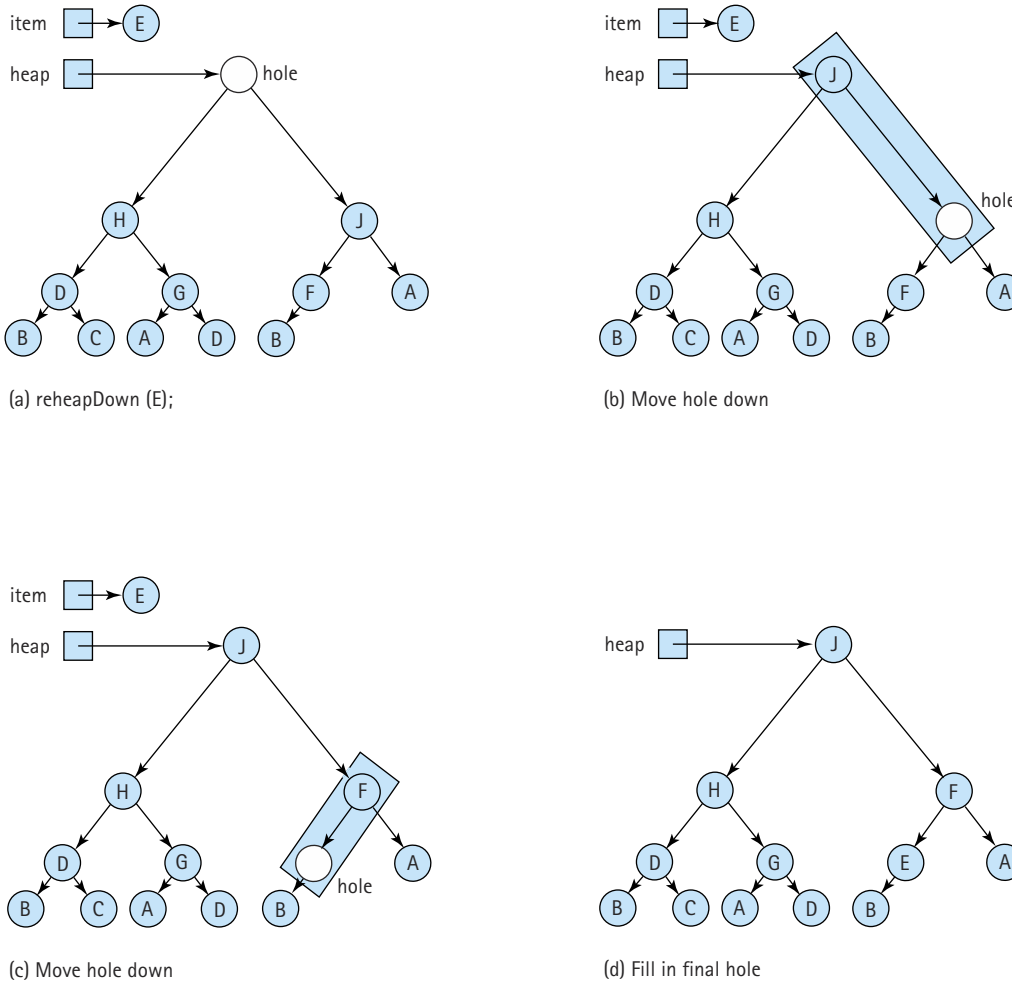


Figure 9.6 *The `reheapDown` operation in action*

Before we can move the hole we need to know where to move it. It should either move to its left child or its right child, or it should stay where it is. Let's assume the existence of another helper method, called `newHole`, that provides us this information. The specification for `newHole` is:

```
private int newHole(int hole, Comparable item)
// If either child of hole is larger than item, this returns the index
//  of the larger child; otherwise, it returns the index of hole
```

Given the index of the hole, `newHole` returns the index of the next location for the hole. If `newHole` returns the same index that is passed to it, we know the hole is at its final location. The `reheapDown` algorithm repeatedly calls `newHole` to find the next index for the hole, and then moves the hole down to that location. It does this until `newHole` returns the same index that is passed to it. The existence of `newHole` simplifies `reheapDown` so that we can now create its code:

```
private void reheapDown(Comparable item)
// Current root position is "empty";
// Inserts item into the tree and ensures shape and order properties
{
  int hole = 0;       // Current index of hole
  int newhole;        // Index where hole should move to

  newhole = newHole(hole, item);   // Find next hole
  while (newhole != hole)
  {
    elements[hole] = elements[newhole];  // Move element up
    hole = newhole;                      // Move hole down
    newhole = newHole(hole, item);       // Find next hole
  }
  elements[hole] = item;           // Fill in the final hole
}
```

Now the only thing left to do is create the `newHole` method. This method does quite a lot of work for us. Consider Figure 9.6 again. Given the initial configuration, `newHole` should return the index of the node containing J, the right child of the hole node; J is larger than either the item (E) or the left child of the hole node (H). So, `newHole` must compare three values (the values in `item`, the left child of the hole node, and the right child of the hole node) and return the index of the `Greatest`. Think about that. It doesn't seem very hard but it does become a little messy when described in algorithmic form:

Greatest(left, right, item) returns index

```
if (left.value( ) < right.value( ))
      if (right.value( ) <= item.value( ))
            return item
      else
            return right
else
if (left.value( ) <= item.value( ))
      return item;
else
      return left;
```

Of course, there are other approaches to the `Greatest` algorithm, but they all require about the same number of comparisons. One benefit of the above algorithm is that if `item` is tied for being the largest of the three arguments its index is returned. This helps our program be efficient since in this situation we want the hole to stop moving (`reheapDown` breaks out of its loop when the value of `hole` is returned). Trace the algorithm with various combinations of arguments to convince yourself that it works.

The `Greatest` algorithm only applies to the case when the hole node has two children. The `newHole` method must also handle the cases where the hole node is a leaf and where the hole node has only one child. How can we tell if a node is a leaf or if it only has one child? Easily, based on the fact that our tree is complete. First, we calculate the expected position of the left child; if this position is greater than `lastIndex`, then the tree has no node at this position and the hole node is a leaf. (Remember, if it doesn't have a left child, it cannot have a right child since the tree is complete.) In this case `newHole` just returns the index of its hole parameter, since the hole cannot move anymore. If the expected position of the left child is equal to `lastIndex` then the node has only one child, and `newHole` returns the index of that child if the child's value is larger than the value of `item`.

Here is the code for `newHole`. As you can see, it is a sequence of *if-else* statements that capture the approaches described in the preceding two paragraphs.

```
private int newHole(int hole, Comparable item)
// If either child of hole is larger than item this returns the index
//  of the larger child; otherwise, it returns the index of hole
{
  int left = (hole * 2) + 1;
  int right = (hole * 2) + 2;
```

```
    if (left > lastIndex)
      // Hole has no children
      return hole;
    else
    if (left == lastIndex)
      // Hole has left child only
      if (item.compareTo(elements[left]) < 0)
        // item < left child
        return left;
      else
        // item >= left child
        return hole;
    else
    // Hole has two children
    if (elements[left].compareTo(elements[right]) < 0)
      // left child < right child
      if (elements[right].compareTo(item) <= 0)
        // right child <= item
        return hole;
      else
        // item < right child
        return right;
    else
    // left child >= right child
    if (elements[left].compareTo(item) <= 0)
      // left child <= item
      return hole;
    else
      // item < left child
      return left;
  }
```

Heaps Versus Other Representations of Priority Queues

How efficient is the heap implementation of a priority queue? The constructor, `isEmpty`, and `isFull` methods are trivial, so we examine only the operations to add and remove elements. The `enqueue` and `dequeue` methods both consist of a few basic operations plus a call to a helper method. The `reheapUp` method creates a slot for a new element by moving a hole up the tree, level by level; because a complete tree is of minimum height, there are at most $\log_2 N$ levels above the leaf level (N = number of elements). So `enqueue` is an $O(\log_2 N)$ operation. The `reheapDown` method is invoked to fill the hole in the root created by the `dequeue` method. This operation moves the hole down in the tree, level by level. Again, there are at most $\log_2 N$ levels below the root; therefore, `dequeue` is also an $O(\log_2 N)$ operation.

How does this implementation compare to the others we mentioned in the previous section? If we implement the priority queue with a linked list, sorted from largest to

Table 9.1 *Comparison of Priority Queue Implementations*

	enqueue	dequeue
Heap	$O(\log_2 N)$	$O(\log_2 N)$
Linked List	$O(N)$	$O(1)$
Binary Search Tree		
Balanced	$O(\log_2 N)$	$O(\log_2 N)$
Skewed	$O(N)$	$O(N)$

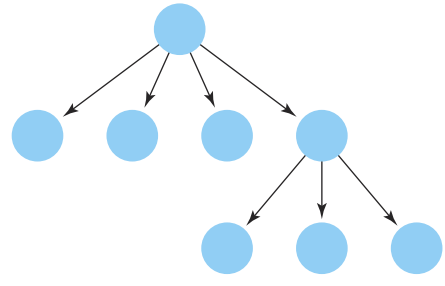
smallest priority, `dequeue` merely removes the first node from the list—an O(1) operation. `enqueue`, however, must search up to all the elements in the list to find the appropriate insertion place; thus it is an $O(N)$ operation.

If the priority queue is implemented using a binary search tree, the efficiency of the operations depends on the shape of the tree. When the tree is bushy, both `dequeue` and `enqueue` are $O(\log_2 N)$ operations. In the worst case, if the tree degenerates to a linked list sorted from smallest to largest priority, both `enqueue` and `dequeue` have $O(N)$ efficiency. Table 9.1 summarizes the efficiency of the different implementations.

Overall, the binary search tree looks good, if it is balanced. It can, however, become skewed, which reduces the efficiency of the operations. The heap, on the other hand, is always a tree of minimum height. The heap is not a good structure for accessing a randomly selected element, but that is not one of the operations defined for priority queues. The accessing protocol of a priority queue specifies that only the largest (or highest-priority) element can be accessed. The linked list is excellent for this operation (assuming the list is sorted from largest to smallest), but we may have to search the whole list to find the place to add a new element. For the operations specified for priority queues, therefore, the heap is an excellent choice.

9.3 Introduction to Graphs

Binary trees provide a very useful way of representing relationships in which a hierarchy exists. That is, a node is pointed to by at most one other node (its parent), and each node points to at most two other nodes (its children). If we remove the restriction that each node can have at most two children, we have a general tree, as pictured here.



Graph A data structure that consists of a set of nodes and a set of edges that relate the nodes to each other

Vertex A node in a graph

Edge (arc) A pair of vertices representing a connection between two nodes in a graph

Undirected graph A graph in which the edges have no direction

Directed graph (digraph) A graph in which each edge is directed from one vertex to another (or the same) vertex

If we also remove the restriction that each node may have only one parent node, we have a data structure called a graph. A graph is made up of a set of nodes called vertices and a set of lines called edges (or arcs) that connect the nodes.

The set of edges describes relationships among the vertices. For instance, if the vertices are the names of cities, the edges that link the vertices could represent roads between pairs of cities. Because the road that runs between Houston and Austin also runs between Austin and Houston, the edges in this graph have no direction. This is called an undirected graph. However, if the edges that link the vertices represent flights from one city to another, the direction of each edge is important. The existence of a flight (edge) from Houston to Austin does not assure the existence of a flight from Austin to Houston. A graph whose edges are directed from one vertex to another is called a directed graph, or digraph.

From a programmer's perspective, vertices represent whatever is the subject of our study: people, houses, cities, courses, and so on. However, mathematically, vertices are the abstract concept upon which graph theory rests. In fact, there is a great deal of formal mathematics associated with graphs. In other computing courses, you may analyze graphs and prove theorems about them. This textbook introduces the graph as an abstract data type, teaches some basic terminology, discusses how a graph might be implemented, and describes how algorithms that manipulate graphs make use of stacks, queues, and priority queues.

Formally, a graph G is defined as follows:

$$G = (V, E)$$

where

V(G) is a finite, nonempty set of vertices

E(G) is a set of edges (written as pairs of vertices)

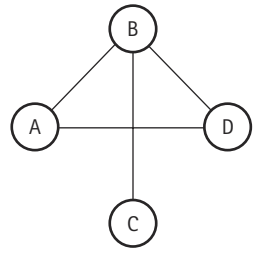
The set of vertices is specified by listing them in set notation, within { } braces. The following set defines the four vertices of the graph pictured in Figure 9.7(a):

$$V(\text{Graph1}) = \{A, B, C, D\}$$

The set of edges is specified by listing a sequence of edges. Each edge is denoted by writing the names of the two vertices it connects in parentheses, with a comma between them. For instance, the vertices in Graph1 in Figure 9.7(a) are connected by the four edges described below:

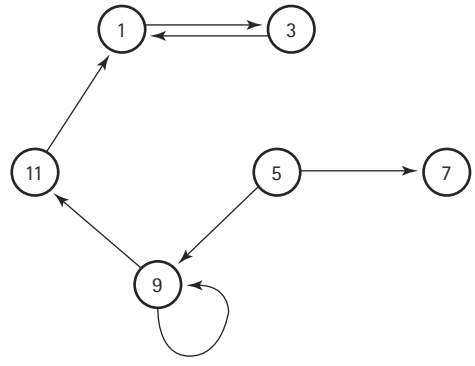
$$E(\text{Graph1}) = \{(A, B), (A, D), (B, C), (B, D)\}$$

(a) Graph1 is an undirected graph.



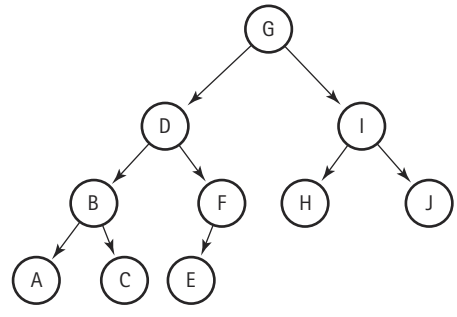
V(Graph1) = {A, B, C, D}
E(Graph1) = {(A, B), (A, D), (B, C), (B, D)}

(b) Graph2 is a directed graph.



V(Graph2) = {1, 3, 5, 7, 9, 11}
E(Graph2) = {(1, 3), (3, 1), (5, 7), (5, 9), (9, 11), (9, 9), (11, 1)}

(c) Graph3 is a directed graph.



V(Graph3) = {A, B, C, D, E, F, G, H, I, J}
E(Graph3) = {(G, D), (G, J), (D, B), (D, F), (I, H), (I, J), (B, A), (B, C), (F, E)}

Figure 9.7 *Some examples of graphs*

Because Graph1 is an undirected graph, the order of the vertices in each edge is unimportant. The set of edges in Graph1 can also be described as follows:

E(Graph1) = {(B, A), (D, A), (C, B), (D, B)}

If the graph is a digraph, the direction of the edge is indicated by which vertex is listed first. For instance, in Figure 9.7(b), the edge (5,7) represents a link from vertex 5 to vertex 7. However, there is no corresponding edge (7,5) in Graph2. Note that in pictures of digraphs, the arrows indicate the direction of the relationship.

If two vertices in a graph are connected by an edge, they are said to be adjacent. In Graph1 (Figure 9.7a), vertices A and B are adjacent, but vertices A and C are not. If the vertices are connected by a directed edge, then the first vertex is said to be *adjacent to* the second, and the second vertex is said to be *adjacent from* the first. For example, in Graph2 (in Figure 9.7b), vertex 5 is adjacent to vertices 7 and 9, while vertex 1 is adjacent from vertices 3 and 11.

Adjacent vertices Two vertices in a graph that are connected by an edge

Path A sequence of vertices that connects two nodes in a graph

Complete graph A graph in which every vertex is directly connected to every other vertex

Weighted graph A graph in which each edge carries a value

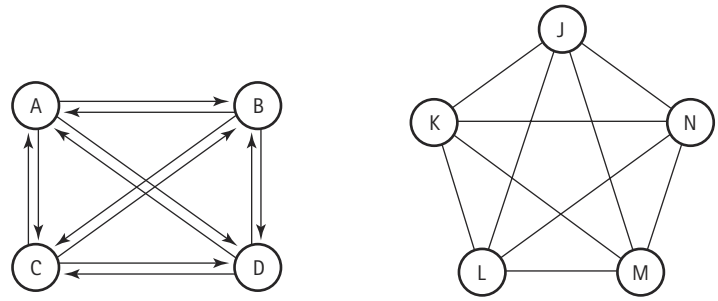
The picture of Graph3 in Figure 9.7(c) may look familiar; it is the tree we looked at earlier in connection with the nonlinked representation of a binary tree. A tree is a special case of a directed graph in which each vertex may only be adjacent from one other vertex (its parent node) and one vertex (the root) is not adjacent from any other vertex.

A path from one vertex to another consists of a sequence of vertices that connect them. For a path to exist, there must be an uninterrupted sequence of edges from the first vertex, through any number of vertices, to the second vertex. For example, in Graph2, there is a path from vertex 5 to vertex 3, but not from vertex 3 to vertex 5. Note that in a tree, such as Graph3 (Figure 9.7c), there is a unique path from the root to every other node in the tree.

A complete graph is one in which every vertex is adjacent to every other vertex. Figure 9.8 shows two complete graphs. If there are N vertices, there are $N * (N - 1)$ edges in a complete directed graph and $N * (N - 1) / 2$ edges in a complete undirected graph.

A weighted graph is a graph in which each edge carries a value. Weighted graphs can be used to represent applications in which the *value* of the connection between the vertices is important, not just the *existence* of a connection. For instance, in the weighted graph pictured in Figure 9.9, the vertices represent cities and the edges indicate the Air Busters Airlines flights that connect the cities. The weights attached to the edges represent the air distances between pairs of cities.

To see whether we can get from Denver to Washington, we look for a path between them. If the total travel distance is determined by the sum of the distances between each pair of cities along the way, we can calculate the travel distance by adding the weights attached to the edges that constitute the path between them. Note that there may be multiple paths between two vertices. Later in this chapter, we talk about a way to find the shortest path between two vertices.



(a) Complete directed graph. (b) Complete undirected graph.

Figure 9.8 *Two complete graphs*

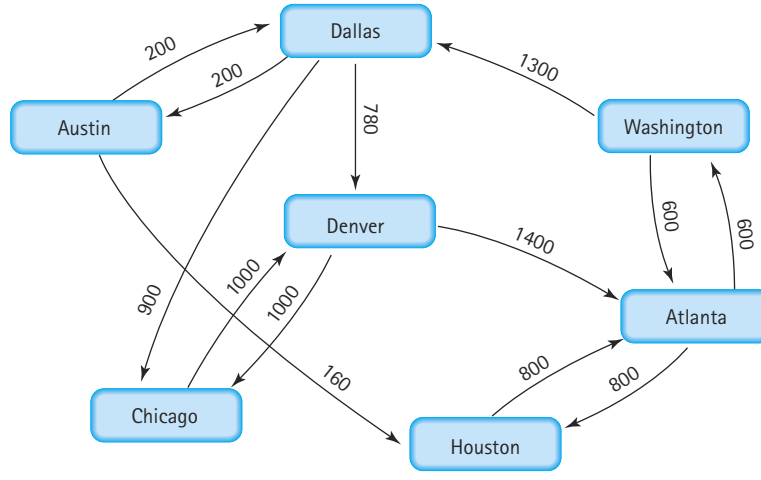


Figure 9.9 *A weighted graph*

Logical Level

We have described a graph at the abstract level as a set of vertices and a set of edges that connect some or all of the vertices to one another. What kind of operations are defined on a graph? In this chapter we specify and implement a small set of useful graph operations. Many other operations on graphs can be defined; we have chosen operations that are useful in the graph applications described later in the chapter.

The specification for the ADT Graph, listed below, includes methods for checking whether the graph is empty or full, and methods to add vertices and edges. The method `weightIs` returns the weight of the edge between two given vertices; if there is no such edge it returns a special value indicating that fact. The special value could

vary from one application to another. For example, the value −1 could be used for a graph whose edges represent distances, since there's no such thing as a negative distance. The special value for a specific application could be passed to the Graph constructor.

The last method specified, `getToVertices`, returns a queue of vertex objects. This works since both our graph and queue ADTs use the "by reference" storage approach. Note that the return value of the method is of "type" `QueueInterface`. The implementation programmer can choose to use any queue class that implements this interface.

```
//--------------------------------------------------------------------------
// WeightedGraphInterface.java       by Dale/Joyce/Weems             Chapter 9
//
// Interface for a class that implements a directed graph with weighted edges
// Vertices are Objects
// Edge weights are integers
//--------------------------------------------------------------------------

package ch09.graphs;

import ch04.queues.*;

public interface WeightedGraphInterface
{
  public boolean isEmpty();
  // Effect:         Determines whether this graph is empty
  // Postcondition:  Return value = (this graph is empty)

  public boolean isFull();
  // Effect:         Determines whether this graph is full
  // Postcondition:  Return value = (this graph is full)

  public void addVertex(Object vertex);
  // Effect:         Adds vertex to the graph
  // Precondition:   Graph is not full
  // Postcondition:  vertex is in V(graph)

  public void addEdge(Object fromVertex, Object toVertex, int weight);
  // Effect:         Adds an edge with the specified weight from fromVertex
  //                   to toVertex
  // Precondition:   fromVertex and toVertex are in V(graph)
  // Postcondition:  (fromVertex, toVertex) is in E(graph) with the specified
  //                   weight
```

```
  public int weightIs(Object fromVertex, Object toVertex);
  // Effect:          Determines the weight of the edge from fromVertex to
  //                  toVertex
  // Precondition:    fromVertex and toVertex are in V(graph)
  // Postcondition:   if edge exists, Return value = (weight of edge from
  //                  fromVertex to toVertex);
  //                  otherwise, return value = (special "null-edge" value)

  public QueueInterface getToVertices(Object vertex)
  // Effect:          Returns a queue of the vertices that are adjacent from
  //                  vertex
  // Precondition:    vertex is in V(graph)
  // Postcondition:   returns a queue containing all the vertices that are
  //                  adjacent from vertex
}
```

Application Level

The graph specification given in the last section included only the most basic operations. It did not include any traversal operations. As you might imagine, there are many different orders in which we can traverse a graph. As a result, we consider traversal a graph application rather than an innate operation. The basic operations given in our specification allow us to implement different traversals *independent* of how the graph itself is actually implemented.

In Chapter 8, we discussed the postorder tree traversal, which goes to the deepest level of the tree and works up. This strategy of going down a branch to its deepest point and moving up is called a *depth-first* strategy. Another systematic way to visit each vertex in a tree is to visit each vertex on level 0 (the root), then each vertex on level 1, then each vertex on level 2, and so on. Visiting each vertex by level in this way is called a *breadth-first* strategy. With graphs, both depth-first and breadth-first strategies are useful. We outline both algorithms within the context of the airline example.

Depth-First Searching

One question we can answer with the graph in Figure 9.9 is "Can I get from city X to city Y on my favorite airline?" This is equivalent to asking "Does a path exist in the graph from vertex X to vertex Y?" Using a depth-first strategy, let's develop an algorithm that determines if a path exists from `startVertex` to `endVertex`.

We need a systematic way to keep track of the cities as we investigate them. With a depth-first search, we examine the first vertex that is adjacent from `startVertex`; if this is `endVertex`, the search is over. Otherwise, we examine all the vertices that can be reached in one step (are adjacent from) this vertex. Meanwhile, we need to store the other vertices that are adjacent from `startVertex`. If a path does not exist from the first vertex, we come back and try the second, third, and so on. Because we want to

travel as far as we can down one path, backtracking if the `endVertex` is not found, a stack is a good structure for storing the vertices. Here is the algorithm we use:

DepthFirstSearch (startVertex, endVertex): returns boolean

```
Set found to false
stack.push(startVertex)
do
      vertex = stack.top( )
      stack.pop( )
      if vertex = endVertex
            Set found to true
      else
            Push all adjacent vertices onto stack
while !stack.isEmpty( ) AND !found
return found
```

Let's apply this algorithm to the sample airline-route graph in Figure 9.9. We want to fly from Austin to Washington. We initialize our search by pushing our starting city onto the stack (Figure 9.10a). At the beginning of the loop we retrieve the current city, Austin, from the stack (`top`) and then remove it from the stack (`pop`). The places we can reach directly from Austin are Dallas and Houston; we push both these vertices onto the stack (Figure 9.10b). At the beginning of the second iteration we retrieve and remove the top vertex from the stack—Houston. Houston is not our destination, so we resume our search from there. There is only one flight out of Houston, to Atlanta; we push Atlanta onto the stack (Figure 9.10c). Again we retrieve and remove the top vertex from the stack. Atlanta is not our destination, so we continue searching from there. Atlanta has flights to two cities: Houston and Washington.

But we just came from Houston! We don't want to fly back to cities that we have already visited; this could cause an infinite loop. We have to prevent cycling in this

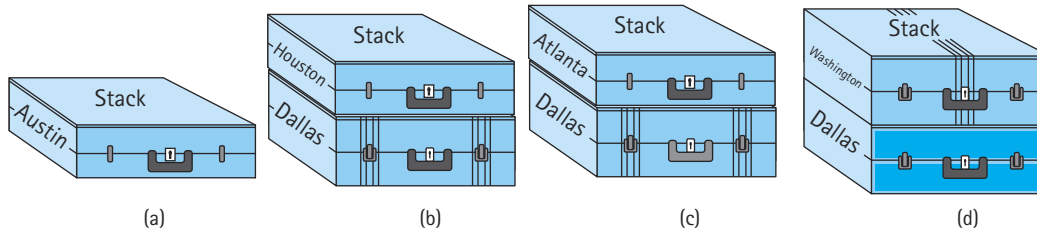


Figure 9.10 *Using a stack to store the routes*

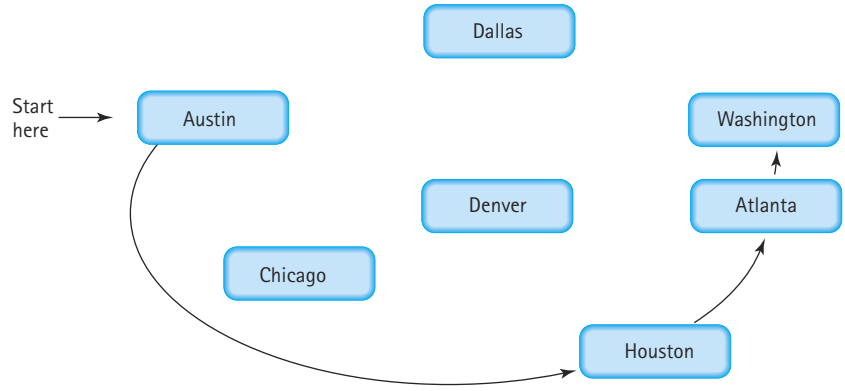


Figure 9.11 *The depth-first search*

algorithm. We must mark a city as having been visited so that it is not investigated a second time. Let's assume that we have marked the cities that have already been tried, and continue our example. Houston has already been visited, so we ignore it. The second adjacent vertex, Washington, has not been visited so we push it onto the stack (Figure 9.10d). Again we retrieve and remove the top vertex from the stack. Washington is our destination, so the search is complete. The path from Austin to Washington, using a depth-first search, is illustrated in Figure 9.11.

This search is called a depth-first search because we go to the deepest branch, examining all the paths beginning at Houston, before we come back to search from Dallas. When you have to backtrack, you take the branch closest to where you dead-ended. That is, you go as far as you can down one path before you take alternative choices at earlier branches.

Before we look at the source code of the depth-first search algorithm, let's talk a little more about "marking" vertices on the graph. Before we begin the search, any marks in the vertices must be cleared to indicate they are not yet visited. Let's call this method `clearMarks`. As we visit each vertex during the search, we mark it. Let's call this method `markVertex`. Before we process each vertex we can ask, "Have we visited this vertex before?" The answer to this question is returned by the method `isMarked`. If we have already visited this vertex, we ignore it and go on. We must add these three methods to the specifications of the Graph ADT.

Additions to Weighted Graph ADT

```
public void clearMarks();
// Effect:         Sets marks for all vertices to false
// Postcondition:  All marks have been set to false

public void markVertex(Object vertex);
// Effect:         Sets mark for vertex to true
// Precondition:   vertex is in V(graph)
// Postcondition:  isMarked(vertex) is true
```

```
public boolean isMarked(Object vertex);
// Effect:         Determines if vertex has been marked
// Precondition:   vertex is in V(graph)
// Postcondition:  return value = (vertex is marked true)
```

Method `depthFirstSearch` receives a graph object, a starting vertex, and a target vertex. It uses the depth-first strategy to determine if there is a path from the starting city to the ending city, displaying the names of all the cities visited in the search. Note that there is nothing in the method that depends on the implementation of the graph. The method is implemented as a graph application; it uses the Graph ADT operations (including the mark operations), without knowing how the graph is represented. In the following code, we assume that a stack and a queue implementation have been imported into the client class. (The `depthFirstSearch` method is included in the `UseGraph.java` application, available on our web site.)

```
private static boolean depthFirstSearch(WeightedGraphInterface graph,
                                        Object startVertex,
                                        Object endVertex    )

// Returns true if a path exists on graph, from startVertex to endVertex;
//  otherwise, returns false

{
  StackInterface stack = new LinkedStack();
  QueueInterface vertexQueue = new LinkedQueue();

  boolean found = false;
  Object vertex;
  Object item;

  graph.clearMarks();
  stack.push(startVertex);
  do
  {
    vertex = stack.top();
    stack.pop();
    if (vertex == endVertex)
       found = true;
    else
    {
      if (!graph.isMarked(vertex))
      {
        graph.markVertex(vertex);
        vertexQueue = graph.getToVertices(vertex);
```

```
        while (!vertexQueue.isEmpty())
        {
          item = vertexQueue.dequeue();
          if (!graph.isMarked(item))
            stack.push(item);
        }
      }
    }
  } while (!stack.isEmpty() && !found);

  return found;
}
```

Breadth-First Searching

A breadth-first search looks at all possible paths at the same depth before it goes to a deeper level. In our flight example, a breadth-first search checks all possible one-stop connections before checking any two-stop connections. For most travelers, this is the preferred approach for booking flights.

When we come to a dead end in a depth-first search, we back up as little as possible. We try another route from a recent vertex—the route on top of our stack. In a breadth-first search, we want to back up as far as possible to find a route originating from the earliest vertices. The stack is not the right structure for finding an early route. It keeps track of things in the order opposite of their occurrence—the latest route is on top. To keep track of things in the order in which they happened, we use a FIFO queue. The route at the front of the queue is a route from an earlier vertex; the route at the back of the queue is from a later vertex.

To modify the search to use a breadth-first strategy, we change all the calls to stack operations to the analogous FIFO queue operations. Searching for a path from Austin to Washington, we first enqueue all the cities that can be reached directly from Austin: Dallas and Houston (Figure 9.12a). Then we dequeue the front queue element. Dallas is not the destination we seek, so we enqueue all the adjacent cities that have not yet been visited: Chicago and Denver (Figure 9.12b). (Austin has been visited already, so it is not enqueued.) Again we dequeue the front element from the queue. This element is the other "one-stop" city, Houston. Houston is not the desired destination, so we continue the search. There is only one flight out of Houston, and it is to Atlanta. Because we haven't visited Atlanta before, it is enqueued (Figure 9.12c).

Now we know that we cannot reach Washington with one stop, so we start examining the two-stop connections. We dequeue Chicago; this is not our destination, so we put its adjacent city, Denver, into the queue (Figure 9.12d). Now this is an interesting situation: Denver is in the queue twice. Should we mark a city as having been visited when we put it in the queue or after it has been dequeued, when we are examining its outgoing flights? If we mark it only after it is dequeued, there may be multiple copies of the same vertex in the queue (so we need to check to see if a city is marked after it is dequeued.)

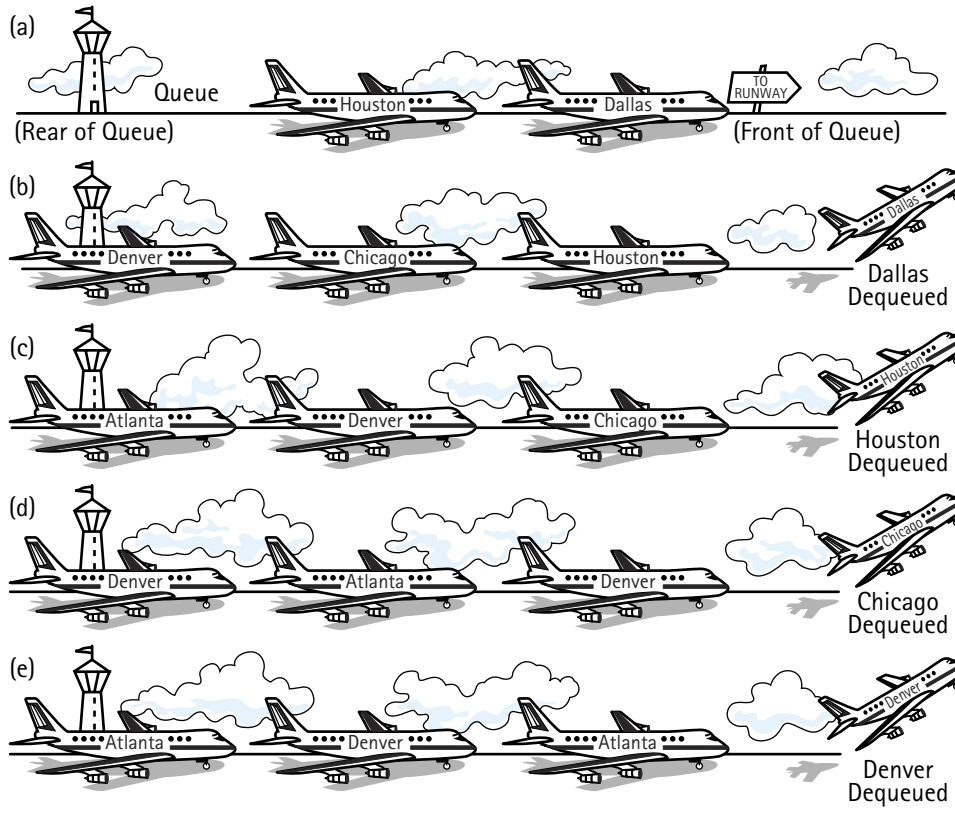


Figure 9.12 *Using a queue to store the routes*

An alternative approach is to mark the city as having been visited before it is put into the queue. Which is better? It depends on the processing goals. You may want to know whether there are alternative routes, in which case you would want to put a city into the queue more than once.

Back to our example. We have put Denver into the queue in one step and removed its previous entry at the next step. Denver is not our destination, so we put its adjacent cities that we haven't already marked (only Atlanta) into the queue (Figure 9.12e). This processing continues until Washington is put into the queue (from Atlanta), and is finally dequeued. We have found the desired city, and the search is complete. This search is illustrated in Figure 9.13.

The source code for the `breadthFirstSearch` method is identical to the depth-first search, except for the replacement of the stack with a FIFO queue. It is also included in the `UseGraph.java` application, available on our web site.

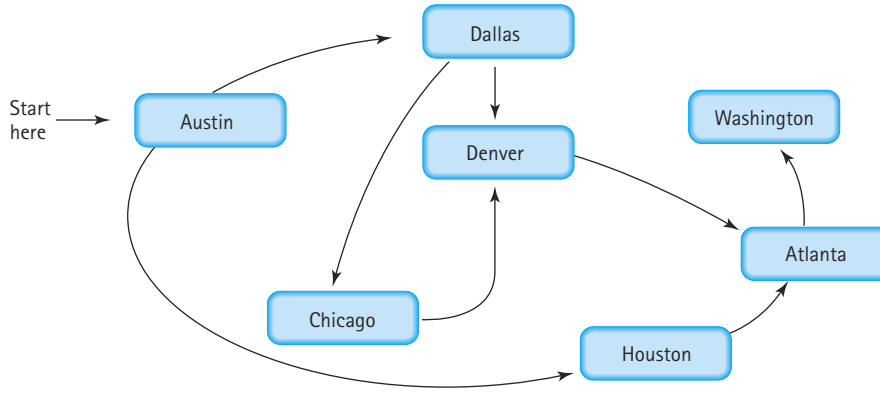


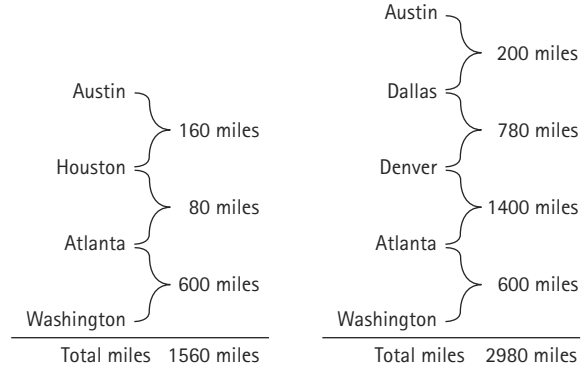
Figure 9.13 *The breadth-first search*

```
private static boolean breadthFirstSearch(WeightedGraphInterface graph,
                                          Object startVertex,
                                          Object endVertex    )

// Returns true if a path exists on graph, from startVertex to endVertex;
//  otherwise, returns false

{
  QueueInterface queue = new LinkedQueue();
  QueueInterface vertexQueue = new LinkedQueue();

  boolean found = false;
  Object vertex;
  Object item;

  graph.clearMarks();
  queue.enqueue(startVertex);
  do
  {
    vertex = queue.dequeue();
    if (vertex == endVertex)
       found = true;
```

```
      else
      {
        if (!graph.isMarked(vertex))
        {
          graph.markVertex(vertex);
          vertexQueue = graph.getToVertices(vertex);

          while (!vertexQueue.isEmpty())
          {
            item = vertexQueue.dequeue();
            if (!graph.isMarked(item))
              queue.enqueue(item);
           }
        }
      }
    } while (!queue.isEmpty() && !found);

    return found;
  }
```


The Single-Source Shortest-Paths Problem

We know from the two search operations just discussed that there may be multiple paths from one vertex to another. Suppose that we want to find the shortest path from Austin to each of the other cities that Air Busters serves. By "shortest path" we mean the path whose edge values (weights), added together, have the smallest sum. Consider the following two paths from Austin to Washington:



Clearly, the first path is preferable, unless you want to collect extra frequent-flyer miles.

Let's develop an algorithm that displays the shortest path from a designated starting city to *every other city* in the graph—this time we are not searching for a path between a starting city and an ending city. As in the two graph searches described earlier, we need an auxiliary structure for storing cities that we process later. By retrieving the city that was most recently put into the structure, the depth-first search

tries to keep going "forward." It tries a one-flight solution, then a two-flight solution, then a three-flight solution, and so on. It backtracks to a fewer-flight solution only when it reaches a dead end. By retrieving the city that had been in the structure the longest time, the breadth-first search tries all one-flight solutions, then all two-flight solutions and so on. The breadth-first search finds a path with a minimum number of flights.

But a minimum *number* of flights does not necessarily mean the minimum total distance. Unlike the depth-first and breadth-first searches, this *shortest-path traversal* must use the number of miles (edge weights) between cities. We want to retrieve the vertex that is *closest* to the current vertex—that is, the vertex connected with the minimum edge weight. If we consider minimum distance to be the highest priority, then we know of a perfect structure—the priority queue. Our algorithm can use a priority queue whose elements are flights (edges) with the distance from the starting city as the priority. That is, the items on the priority queue are objects with three attributes: `fromVertex`, `toVertex`, and `distance`. We use an inner class `Flights` to define these objects. Here is the algorithm:

shortestPaths

```
graph.ClearMarks( )
Set item.fromVertex to startVertex
Set item.toVertex to startVertex
Set item.distance to 0
pq.enqueue(item)

do
    item = pq.dequeue( )
    if item.toVertex is not marked
        Mark item.toVertex
        Write item.fromVertex, item.toVertex, item.distance
        Set item.fromVertex to item.toVertex
        Set minDistance to item.distance
        Get queue vertexQueue of vertices adjacent from item.fromVertex
        while more vertices in vertexQueue
            Get next vertex from vertexQueue
            if vertex not marked
                Set item.toVertex to vertex
                Set item.distance to minDistance + graph.weightIs(fromVertex, vertex)
                pq.enqueue(item)
while !pq.isEmpty( )
```

The algorithm for the shortest-path traversal is similar to those we used for the depth-first and breadth-first searches, but there are three major differences:

1. We use a priority queue rather than a FIFO queue or stack.
2. We stop only when there are no more cities to process; there is no destination.
3. It is incorrect if we use a reference-based priority queue improperly!

When we code this algorithm we are likely to make a subtle, but crucial, error. The error is related to the fact that our queues store information "by reference," and not "by copy." Take a minute to look over the algorithm again to see if you can spot the error, before continuing.

Recall the feature section in Chapter 4 that discussed the dangers of storing information by reference. In particular, it warned us to be careful when inserting an object into a structure and later making changes to that object. If we use the same reference to the object when we make changes to it, the changes are made to the object that is in the structure. Sometimes this is what we want (see the case study in Chapter 8). Sometimes this causes problems, as in the current example. Here is the incorrect part of the algorithm:

```
while more vertices in vertexQueue
    Get next vertex from vertexQueue
    if vertex not marked
        Set item.toVertex to vertex
        Set item.distance to minDistance + graph.weightIs(fromVertex, vertex)
        pq.enqueue(item)
```

Now can you see the problem? This part of the algorithm walks through the queue of vertices adjacent to the current vertex, and enqueues `Flights` objects onto the priority queue `pq` based on the information. The `item` variable is actually a reference to a `Flights` object. Suppose the queue of adjacent vertices has information in it related to the cities Atlanta and Houston. The first time through this loop we insert information related to Atlanta in `item` and enqueue it in `pq`. But the next time through the loop we make changes to the `Flights` object referenced by `item`. We update it to contain information about Houston. And we again enqueue it in `pq`. So now `pq` contains information about Atlanta and Houston, correct? Nope. When we change the information in `item` to the Houston information, those changes are reflected in the `item` that is already on `pq`. The item variable still references that object. In reality the `pq` structure now contains two references to the same `item`, and that `item` contains Houston information.

To solve this problem we must create a new `item` before storing on `pq`. Here is a revised algorithm:

```
while more vertices in vertexQueue
    Get next vertex from vertexQueue
    if vertex not marked
        Create newItem
        Set newItem.fromVertex to item.fromVertex
        Set newItem.toVertex to vertex
        Set newItem.distance to minDistance + graph.weightIs(fromVertex, vertex)
        pq.enqueue(newItem)
```

Here is the source code for the shortest-path algorithm (also included in the `UseGraph.java` application, available on our web site). As before, the code assumes that a priority queue and a queue implementation have been imported into the client class. For the priority queue we use our `Heap` class. Notice that we intend for a smaller distance to indicate a higher priority. But our `Heap` class implements a *maximum* heap, returning the *largest* value from the `dequeue` method. To fix this problem we could define a new heap class, a minimum heap. But there is an easier way. The current `Heap` class bases its decision about what is "larger" on the values returned by the heap item's `compareTo` method. So, we just define the `compareTo` method of the `Flights` class to indicate that the current flight is "larger" then the parameter flight if its `distance` is smaller. This means, for every item in the heap's tree, `item.distance` is `less` than or equal to the `distance` value of each of its children. We can still use our maximum heap.

```
private static void shortestPaths(WeightedGraphInterface graph,
                                  Object startVertex  )
// Writes the shortest path from startVertex to every
//  other vertex in graph
{
  class Flights implements Comparable
  {
    private Object fromVertex;
    private Object toVertex;
    private int distance;

    public int compareTo(Object otherFlights)
    {
     Flights other = (Flights)otherFlights;
     return (other.distance - this.distance); // Shorter is better
    }
  }
```

```
    Flights item;
    Flights saveItem;          // For saving on priority queue
    int minDistance;

    Heap pq = new Heap(10);    // Assume at most 10 vertices
    Object vertex;
    QueueInterface vertexQueue = new LinkedQueue();

    graph.clearMarks();
    saveItem = new Flights();
    saveItem.fromVertex = startVertex;
    saveItem.toVertex = startVertex;
    saveItem.distance = 0;
    pq.enqueue(saveItem);

    System.out.println("Last Vertex  Destination   Distance");
    System.out.println("-----------------------------------");

    do
    {
      item = (Flights)pq.dequeue();
      if (!graph.isMarked(item.toVertex))
      {
        graph.markVertex(item.toVertex);
        System.out.print(item.fromVertex);
        System.out.print("   ");
        System.out.print(item.toVertex);
        System.out.println("   " + item.distance);
        item.fromVertex = item.toVertex;
        minDistance = item.distance;
        vertexQueue = graph.getToVertices(item.fromVertex);
        while (!vertexQueue.isEmpty())
        {
          vertex = vertexQueue.dequeue();
          if (!graph.isMarked(vertex))
          {
            saveItem = new Flights();
            saveItem.fromVertex = item.fromVertex;
            saveItem.toVertex = vertex;
            saveItem.distance = minDistance +
                graph.weightIs(item.fromVertex, vertex);
            pq.enqueue(saveItem);
          }
        }
      }
    } while (!pq.isEmpty());
  }
```

The output from this method is a table of city pairs (edges) showing the total minimum distance from `startVertex` to each of the other vertices in the graph, as well as the last vertex visited before the destination. We assume that printing a vertex means printing the name of the corresponding city. If `graph` contains the information shown in Figure 9.9, the method call

```
shortestPaths(graph, startVertex);
```

where `startVertex` corresponds to Washington, would print a table like the following:

Last Vertex	Destination	Distance
Washington	Washington	0
Washington	Atlanta	600
Washington	Dallas	1300
Atlanta	Houston	1400
Dallas	Austin	1500
Dallas	Denver	2080
Dallas	Chicago	2200

The shortest-path distance from Washington to each destination is shown in the two columns to the right. For example, our flights from Washington to Chicago total 2,200 miles. The left-hand column shows which city immediately preceded the destination in the traversal. Let's figure out the shortest path from Washington to Chicago. We see from the left-hand column that the next-to-last vertex in the path is Dallas. Now we look up Dallas in the Destination (middle) column: The vertex before Dallas is Washington. The whole path is Washington-Dallas-Chicago. (We might want to consider another airline for a more direct route!)

Implementation Level

Array-Based Implementation

A simple way to represent V(graph), the vertices in the graph, is with an array where the array elements are the vertices. For example, if the vertices represent city names, the array might hold strings. A simple way to represent E(graph), the edges in a graph, is by using an adjacency matrix, a two-dimensional array of edge values (weights), where the indexes of a weight correspond to the vertices connected by the edge. Thus, a graph consists of an integer variable `numVertices`, a one-dimensional array `vertices`, and a two-dimensional array `edges`. Figure 9.14 depicts the implementation of the graph of Air Busters flights between seven cities. For simplicity, we omit the additional `boolean` data needed to mark vertices as "visited"

Adjacency matrix For a graph with *N* nodes, an *N* by *N* table that shows the existence (and weights) of all edges in the graph

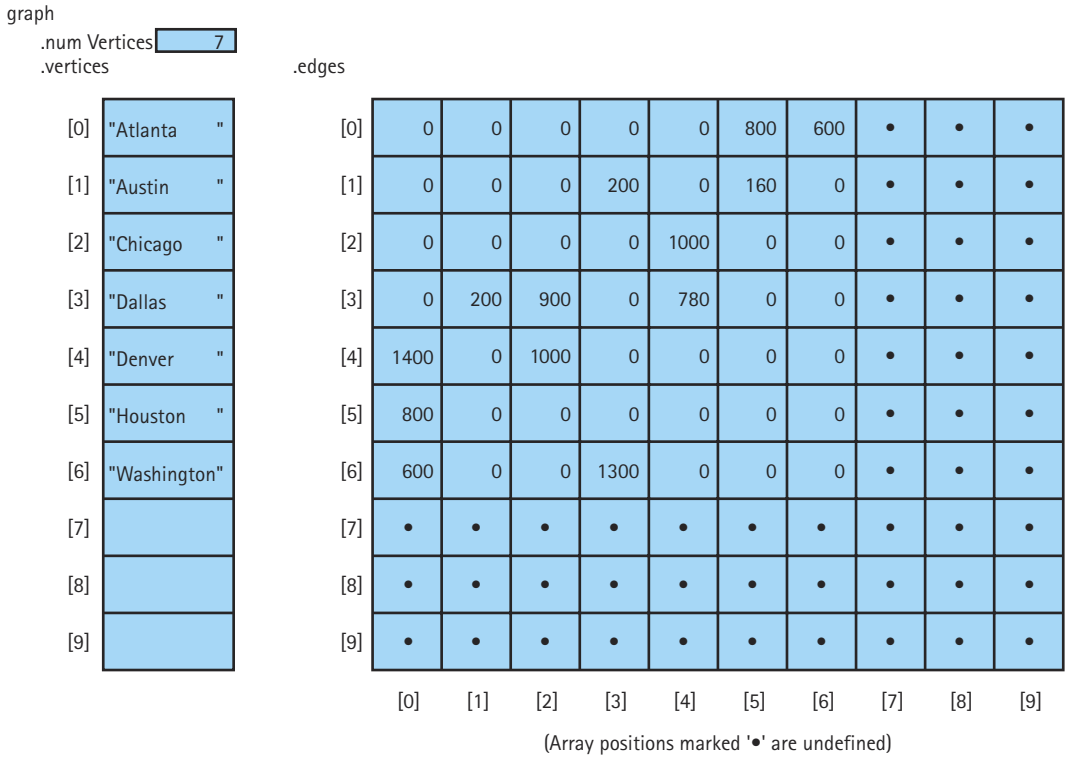


Figure 9.14 *Matrix representation of graph of flight connections between cities*

during a traversal from the figure. While the city names in Figure 9.14 are in alphabetical order, there is no requirement that the elements in this array be sorted.

At any time, within this representation of a graph,

- `numVertices` is the number of vertices in the graph.
- V(graph) is contained in `vertices[0]..vertices[numVertices - 1]`.
- E(graph) is contained in the square array `edges[0][0]..edges[numVertices - 1][numVertices - 1]`.

The names of the cities are contained in `graph.vertices`. The weight of each edge in `graph.edges` represents the air distance between two cities that are connected by a flight. For example, the value in `graph.edges[1][3]` tells us that there is a direct flight between Austin and Dallas, and that the air distance is 200 miles. A `NULL_EDGE` value (0) in `graph.edges[1][6]` tells us that the airline has no direct flights between Austin and Washington. Because this is a weighted graph with weights being air distances, we use `int` for the edge value type. If this were not a weighted graph, the edge

value type would be `boolean`, and each position in the adjacency matrix would be `true` if an edge exists between the pair of vertices, and `false` if no edge exists.

Here is the beginning of the definition of class `WeightedGraph`. For simplicity we assume that the edge value type is `int` and that a null edge is indicated by a 0 value.

```
//---------------------------------------------------------------------------
// WeightedGraph.java            by Dale/Joyce/Weems                Chapter 9
//
// Implements (partially) a directed graph with weighted edges
// Vertices are Objects
// Edge weights are integers
//---------------------------------------------------------------------------

package ch09.graphs;

import ch04.queues.*;
import ch05.queues.*;

public class WeightedGraph implements WeightedGraphInterface
{
  public static int NULL_EDGE = 0;
  private int numVertices;
  private int maxVertices;
  private Object[] vertices;
  private int[][] edges;
  private boolean[] marks;  // marks[i] is marked for vertices[i]

  public WeightedGraph()
  // Post: Arrays of size 50 are dynamically allocated for
  //       marks and vertices, and of size 50 X 50 for edges
  //       numVertices is set to 0; maxVertices is set to 50
  {
    numVertices = 0;
    maxVertices = 50;
    vertices = new Object[50];
    marks = new boolean[50];
    edges = new int[50][50];
  }

  public WeightedGraph(int maxV)
  // Post: Arrays of size maxV are dynamically allocated for
  //       marks and vertices, and of size maxV X maxV for edges
  //       numVertices is set to 0; maxVertices is set to maxV
```

```
  {
    numVertices = 0;
    maxVertices = maxV;
    vertices = new Object[maxV];
    marks = new boolean[maxV];
    edges = new int[maxV][maxV];
  }
...
}
```

Note that the class constructors have to allocate the space for `vertices` and `marks` (the `boolean` array indicating whether a vertex has been marked or not). The default constructor sets up space for 50 `vertices` and `marks`. The parameterized constructor lets the user specify the maximum number of vertices.

The `addVertex` operation puts a vertex into the next free space in the array of vertices. Because the new vertex has no edges defined yet, we also initialize the appropriate row and column of edges to contain `NULL_EDGE` (0 in this case).

```
public void addVertex(Object vertex)
// Post: vertex has been stored in vertices
//       Corresponding row and column of edges has been set to NULL_EDGE.
//       numVertices has been incremented
{
  vertices[numVertices] = vertex;
  for (int index = 0; index < numVertices; index++)
  {
    edges[numVertices][index] = NULL_EDGE;
    edges[index][numVertices] = NULL_EDGE;
  }
  numVertices++;
}
```

To add an edge to the graph, we must first locate the `fromVertex` and `toVertex` that define the edge we want to add. These become the parameters to `addEdge` and are of the general `Object` class. Of course, the client really passes references to the vertex objects, since that is how we manipulate objects in Java. We are implementing our graphs "by reference" so this should not be a problem for the client. To index the correct matrix slot, we need the index in the `vertices` array that corresponds to each vertex. Once we know the indexes, it is a simple matter to set the weight of the edge in the matrix. Here is the algorithm:

addEdge(fromIndex, toIndex, weight)

Set fromIndex to index of fromVertex in V(graph)
Set toIndex to index of toVertex in V(graph)
Set edges[fromIndex, toIndex] to weight

To find the index of each vertex, let's write a little search method that receives a vertex and returns its location (index) in `vertices`. Because the precondition of `addEdge` states that `fromVertex` and `toVertex` are in V(graph), the search method is very simple. We code it as helper method `indexIs`. Here is the code for `indexIs` and `addEdge`:

```
private int indexIs(Object vertex)
// Post: Returns the index of vertex in vertices
{
  int index = 0;
  while (vertex != vertices[index])
    index++;
  return index;
}

public void addEdge(Object fromVertex, Object toVertex, int weight)
// Post: Edge (fromVertex, toVertex) is stored in edges
{
  int row;
  int column;

  row = indexIs(fromVertex);
  column = indexIs(toVertex);
  edges[row][column] = weight;
}
```

The `weightIs` operation is the mirror image of `addEdge`.

```
public int weightIs(Object fromVertex, Object toVertex)
// Post: Returns the weight associated with the edge
//       (fromVertex, toVertex)
{
  int row;
```

```
  int column;

  row = indexIs(fromVertex);
  column = indexIs(toVertex);
  return edges[row][column];
}
```

The last graph operation that we address is `getToVertices`. This method receives a vertex, and returns a queue of vertices that are adjacent from the designated vertex. That is, it returns a queue of all the vertices that you can get to from this vertex in one step. Using an adjacency matrix to represent the edges, it is a simple matter to determine the nodes to which the vertex is adjacent. We merely loop through the appropriate row in `edges`; whenever a value is found that is not `NULL_EDGE`, we add another vertex to the queue.

```
public QueueInterface getToVertices(Object vertex)
// Returns a queue of the vertices that are adjacent from vertex
{
  QueueInterface adjVertices = new LinkedQueue();
  int fromIndex;
  int toIndex;
  fromIndex = indexIs(vertex);
  for (toIndex = 0; toIndex < numVertices; toIndex++)
    if (edges[fromIndex][toIndex] != NULL_EDGE)
      adjVertices.enqueue(vertices[toIndex]);
  return adjVertices;
}
```

We leave `isFull`, `isEmpty`, and the marking operations (`clearMarks`, `markVertex`, and `isMarked`) as exercises.

Linked Implementation

The advantages of representing the edges in a graph with an adjacency matrix are speed and simplicity. Given the indexes of two vertices, determining the existence (or the weight) of an edge between them is a O(1) operation. The problem with adjacency matrices is that their use of space is $O(N^2)$, where N is the maximum number of vertices in the graph. If the maximum number of vertices is large, adjacency matrices may waste a lot of space. The space needed could be decreased by dynamically allocating larger arrays when needed, but that approach can be inefficient in terms of time. In the past, we have tried to save space by allocating memory as we need it at run time, using linked structures. We can use a similar approach to implementing graphs. Adjacency lists are linked lists, one list per vertex, that identify the vertices to which each vertex is connected. There

Adjacency list A linked list that identifies all the vertices to which a particular vertex is connected; each vertex has its own adjacency list

are several ways to implement adjacency lists. Figure 9.15 shows two different adjacency list representations of the graph in Figure 9.9.

In Figure 9.15(a), the vertices are stored in an array. Each component of this array contains a reference to a linked list of edge nodes. Each node in these linked lists contains an index number, a weight, and a reference to the next node in the adjacency list. Let's look at the adjacency list for Denver. The first node in the list indicates that there is a 1,400-mile flight from Denver to Atlanta (the vertex whose index is 0) and a 1,000-mile flight from Denver to Chicago (the vertex whose index is 2).

No arrays are used in the implementation illustrated in Figure 9.15(b). The list of vertices is implemented as a linked list. Now each node in the adjacency list contains a reference to the vertex information rather than the index of the vertex. Because there are so many of these references in Figure 9.15(b), we have used text to describe the vertex that each reference designates rather than draw them as arrows.

We leave the implementation of the `Graph` class methods using the linked approach as a programming exercise.

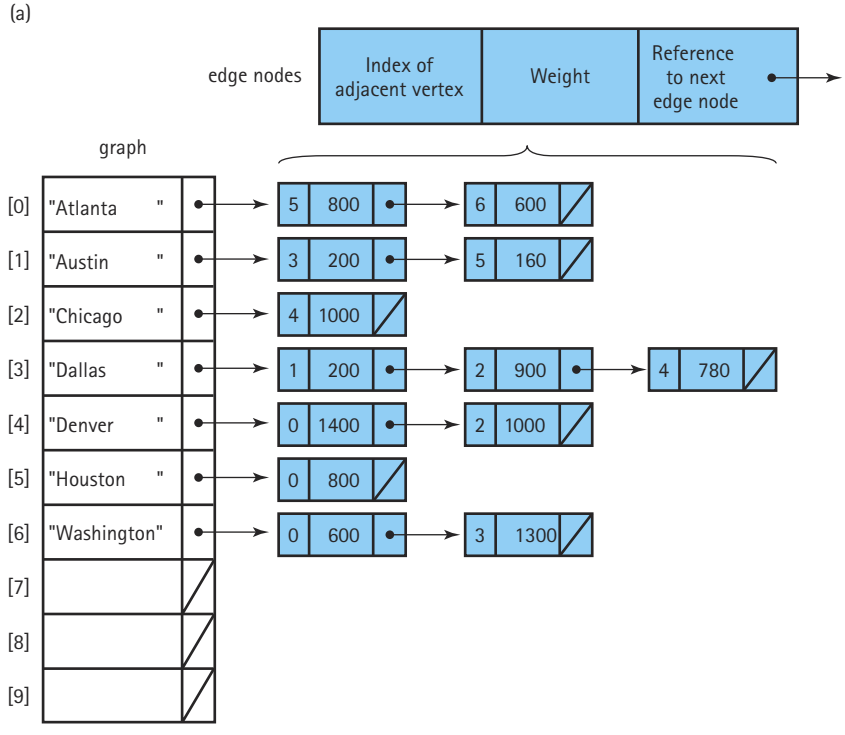


Figure 9.15 *Adjacency list representation of graphs*

(b)

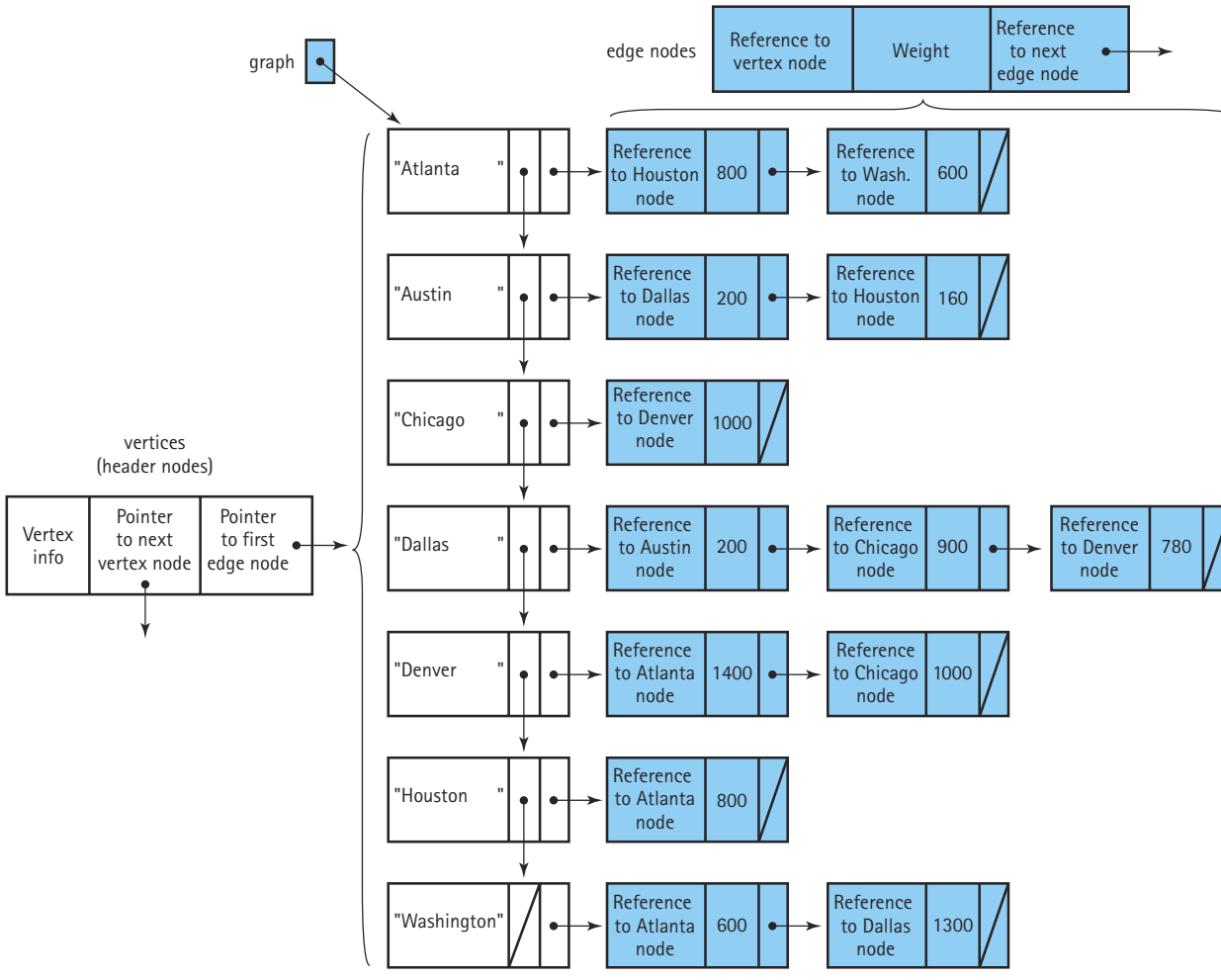


Figure 9.15 *(continued)*

9.4 Storing Objects/Structures in Files

Suppose we want to save a tree or graph between program runs. Our current programs can build structures and use them for processing information, but when the program terminates, the structure is lost. The memory space occupied by the structure, along with all the other memory space used by the program, is returned to the operating system for use by other programs.

Many programs need to save information between program runs. Alternately, we may want one program to save information for later use by another program. In either

case, the information is stored in files. Files are the mechanism for permanently storing information on computers. In this subsection we investigate Java's facilities for saving and retrieving objects and structures using files.

Saving Object Data in Text Files

Any information we need to save can be broken into primitive data. As a very simple example, consider the `Circle` objects defined in Chapter 2. Remember that we used circles to demonstrate the "record" concept. The fields of circles are not private. While we do not encourage this lack of information hiding, it does make it easy to demonstrate the concepts of this section. To simplify our file organization, we place the `Circle` class in a new package called `ch09.circles`.

```
package ch09.circles;

public class Circle
{
  int xValue;       // Horizontal position of center
  int yValue;       // Vertical position of center
  float radius;
  boolean solid;    // True means circle filled
}
```

Although a circle can be viewed as a "nonprimitive" object, when broken into its constituent parts it consists of two `int`s, a `float`, and a `boolean`, all primitive data types. Values of these data types can be saved as strings. We can save a `Circle` object by breaking it into its constituent parts, transforming each part into a string, and writing the strings to a text file. The following program demonstrates this approach:

```
import java.io.*;
import ch09.circles.*;

public class SaveCircle
{
  private static PrintWriter outFile;

  public static void main(String[] args) throws IOException
  {
    Circle c1 = new Circle();
    c1.xValue = 5;
    c1.yValue = 3;
    c1.radius = 3.5f;
    c1.solid = true;

    outFile = new PrintWriter(new FileWriter("circle.dat"));

    outFile.println(c1.xValue);
```

```
    outFile.println(c1.yValue);
    outFile.println(c1.radius);
    outFile.println(c1.solid);

    outFile.close();
  }
}
```

When this program is executed, it creates the following circle.dat file:

```
5
3
3.5
true
```

When we need to retrieve the `Circle` object, we just reverse the process: read the strings, transform them into the primitive data types, and reconstruct the circle.

```
import java.io.*;
import ch09.circles.*;

public class GetCircle
{
  private static BufferedReader inFile;

  public static void main(String[] args) throws IOException
  {
    Circle c2 = new Circle();
    inFile = new BufferedReader(new FileReader("circle.dat"));

    c2.xValue = Integer.parseInt(inFile.readLine());
    c2.yValue = Integer.parseInt(inFile.readLine());
    c2.radius = Float.parseFloat(inFile.readLine());
    c2.solid  = Boolean.getBoolean(inFile.readLine());

    System.out.println("The xValue is " + c2.xValue);
    System.out.println("The yValue is " + c2.yValue);
    System.out.println("The radius is " + c2.radius);
    System.out.println("The solidity is " + c2.solid);

    inFile.close();
  }
}
```

When these two programs are executed back to back, the second program produces the following output:

```
The xValue is 5
The yValue is 3
The radius is 3.5
The solidity is true
```

As you can see, the `Circle` object created and saved by the first program was successfully retrieved and used by the second program. The entire process is depicted in Figure 9.16.

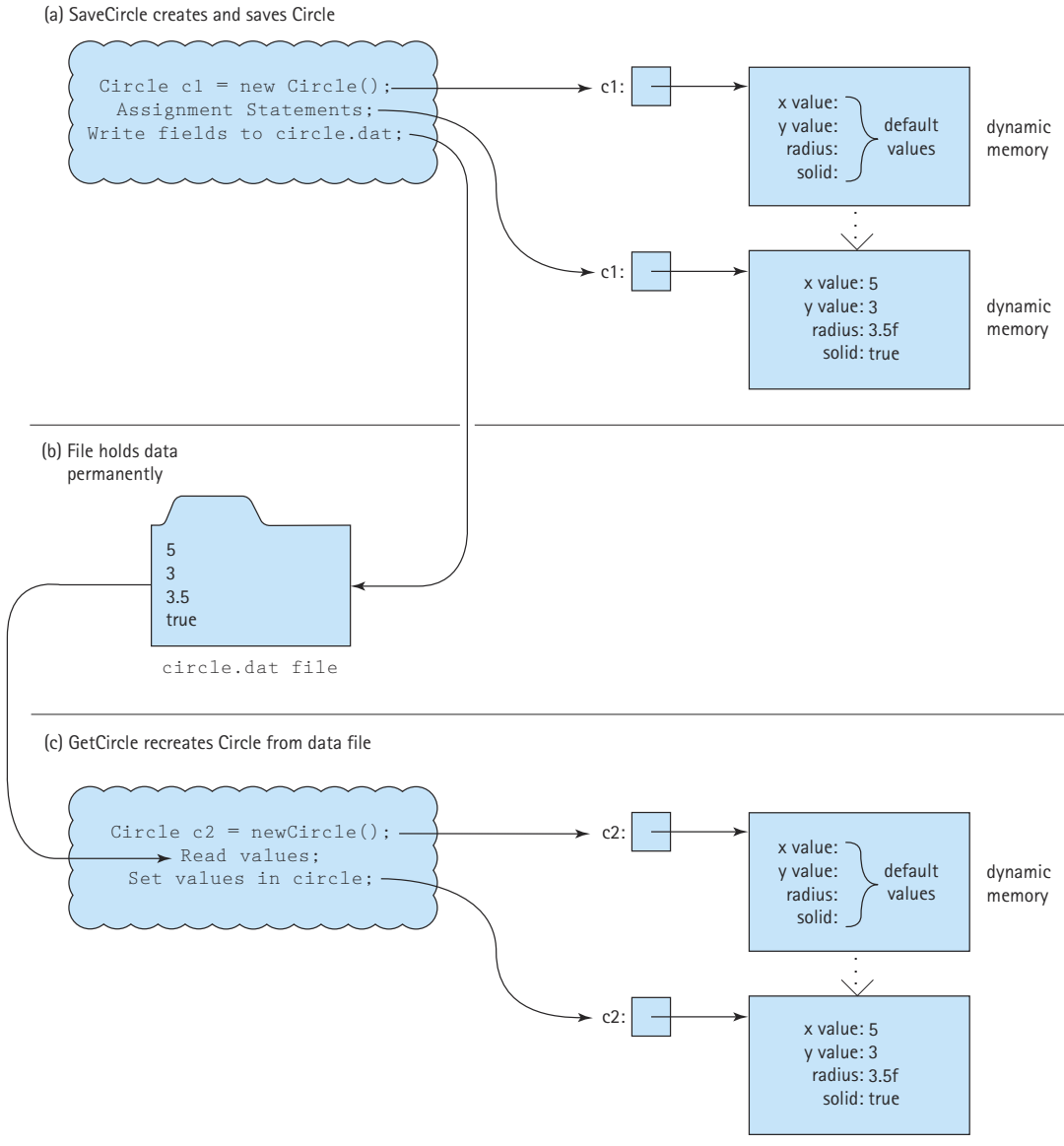


Figure 9.16 *Saving and retrieving a* `Circle` *object*

The circle programs use the same file I/O techniques we use in our test drivers and program examples throughout the textbook. These techniques are explained in the Java Input/Output I feature section at the end of Chapter 1. Note that we used this same approach (transforming an object's primitive attributes into strings, and saving the strings to a file) in the Real Estate program in the Chapter 3 case study. We used it to save and retrieve house information.

This approach can be used even if the objects involved are hierarchical, like our `NewCircle` objects from Chapter 2:

```
public class Point
{
  public int xValue;
  public int yValue;
}

public class NewCircle
{
  public Point location;
  public float radius;
  public boolean solid;
}
```

The programmer just has to work a little harder in saving and restoring the objects. But what about storing and retrieving an entire data structure, such as an array, list, stack, queue, or tree? We examine this question next.

Saving Structures in Text Files

We began the previous subsection by stating that any information we need to save can be broken into primitive data. However, what happens when the structural organization of the data is as important as the data itself? In this case, we would also like to save the structure of the data, or at least be able to recreate the structure when desired.

Simply storing the primitive data of a structure in a specific order, and then carefully rebuilding the structure upon retrieval of the data, can sometimes work. Let's look at several examples:

Arrays

Assuming the array contains homogeneous data, this is easy. Simply save the data one array element at a time, from index 0 through the largest index of the array. Restore the array elements in the same order. For example, to save an array of `Circle` objects just use the approach of the previous subsection combined with loops to repetitively save or restore the information. If the array is not homogeneous, it is not as easy. For example, if the array can contain both `Circle` and `NewCircle` objects, you would somehow have to store a class identifier along with the object information. Upon retrieval, this identifier could be used to determine which type of object to recreate from the data.

Array-Based Lists

No problem. Just save the array information as described above. This is how we handled the list of house information in the Chapter 3 case study.

Reference-Based Lists

We cannot save and restore references as we save and restore primitive data. A reference is really a memory address. When we try to restore information, there is no guarantee that it is placed in the exact same memory location in which it formerly resided—in fact, that would be extraordinarily lucky. However, we usually can reconstruct our list by using our list operations. We should exercise care with this approach—it can lead to inefficient processing. For example, if we store a sorted list in order, from smallest to largest, and then recreate the list by using our Sorted List ADT `insert` operation, it requires $O(N^2)$ steps to recreate the list. (Do you see why?) When processing an item that is larger than any current list element, the `insert` operation visits every node of the current list. That is the worst possible situation. Yet, in the approach just described, that situation would repeat itself for every single `insert` operation when recreating the list. It might be better to read in all of the list elements and store them on a stack, and then remove the elements one at a time from the stack to insert them into the list. This ensures repeated insertion at the start of the list, an efficient operation.

A Linked List as an Array of Nodes

As explained in Section 6.5, using this approach to implementing a linked list makes it easy to save the structure information in a text file. Since the links are not references, but array indices, we can save and restore them just as we would other primitive values.

Binary Search Trees

A binary tree can also be implemented with an array of nodes (see Section 8.9). In that case, as it is for linked lists, it is a simple matter to store and retrieve the tree. But what if the tree is implemented using references? This complicates matters. Certainly we can iterate through the tree and save each node into a file; then later we can construct a binary search tree from the information in the file. But will the tree have the same shape? Do we want it to? The answers depend on the order the tree is traversed to save the nodes, and on the order the nodes are inserted into the new tree. These issues were all addressed in Section 8.8.

Graphs

We discussed both array-based and reference-based implementations of graphs. Following the same reasoning as above, if the array-based approach is used we could safely save and reconstruct our graphs. But the reference approach would require first transforming the references into information that is independent of memory location; for example, storing a reference by storing the names of the nodes that it links. In either case, the transforming, storing, and reconstructing of our graphs is a lot of work. We next discuss an easier approach.

Serialization of Objects

Transforming objects into strings and back again is a lot of work for the programmer. Fortunately, Java provides a way to save objects without requiring all of this work. Saving an object with this approach is called *serializing* the object.

Support Constructs

Before seeing how to serialize objects, you must learn about a new interface and two support classes.

We can write whole objects using the `writeObject` method of the `ObjectOutputStream` class. We can read objects using the `readObject` method of the `ObjectInputStream` class. To set up the output of serialized objects to the file `objects.dat` using the stream variable `out`, we code

```
ObjectOutputStream out = new ObjectOutputStream(new
  FileOutputStream("objects.dat"));
```

Similarly, to set up reading from the same file, but this time using the variable `in`, we code

```
ObjectInputStream in = new ObjectInputStream(new
  FileInputStream("objects.dat"));
```

Finally, any object that you plan to serialize must implement the `Serializable` interface. This interface has no methods! It is merely a way of marking a class as potentially being serialized for I/O, so that the Java runtime engine knows to convert references as needed on output or input of class instances. So to make your objects serializable, you simply have to state that their class implements the interface. The interface is part of the Java `io` package.

Serializing Objects

Here's the code for a serializable version of our circle class, called `SCircle`. As with `Circle`, we place this class in the `ch09.circles` package.

```
package ch09.circles;

import java.io.*;

public class SCircle implements Serializable
{
  public int xValue;
  public int yValue;
  public float radius;
  public boolean solid;
}
```

Now let's look at our program to save a `SCircle` object:

```
import java.io.*;
import ch09.circles.*;

public class SaveSCircle
{
  public static void main(String[] args) throws IOException
  {
    SCircle c1 = new SCircle();
    c1.xValue = 5;
    c1.yValue = 3;
    c1.radius = 3.5f;
    c1.solid = true;

    ObjectOutputStream out = new ObjectOutputStream(new
                                FileOutputStream("objects.dat"));
    out.writeObject(c1);
    out.close();
  }
}
```

As you can see, to save the `SCircle` object we simply write the entire object to the `out` stream. We do not have to handle the individual instance variables. Let's see the corresponding version of retrieving a circle:

```
import java.io.*;
import ch09.circles.*;

public class GetSCircle
{
  public static void main(String[] args) throws IOException
  {
    SCircle c2 = new SCircle();
    ObjectInputStream in = new ObjectInputStream(new
               FileInputStream("objects.dat"));

    try
    {
      c2 = (SCircle)in.readObject();
    }
    catch (Exception e)
    {
      System.out.println("Error in readObject: " + e);
```

```
      System.exit(1);
    }

    System.out.println("The xValue is " + c2.xValue);
    System.out.println("The yValue is " + c2.yValue);
    System.out.println("The radius is " + c2.radius);
    System.out.println("The solidity is " + c2.solid);

    in.close();
  }
}
```

Note that the object read from the file must be cast into a `SCircle` object before being assigned to the `c2` variable. Also note that the `readObject` method throws several checked exceptions, so we must enclose it in a *try-catch* statement. Other than that, it is very easy to read in our `SCircle` object. Java takes care of all the work of rebuilding the object.

Saving hierarchical objects, such as our `NewCircle` objects, is just as easy. In that case, simply ensure that each of the objects involved implements the `Serializable` interface. For example:

```
import java.io.*;
public class SPoint implements Serializable
{
  public int xValue;
  public int yValue;
}
import java.io.*;
public class SNewCircle implements Serializable
{
  public SPoint location;
  public float radius;
  public boolean solid;
}
```

A program that uses `SNewCircle` objects can use the `writeObject` and `readObject` methods to save and retrieve those objects. Java handles the hierarchical object automatically with no problems.

Serializing Structures

The real power of Java's serialization tools is evident when dealing with data structures. For example, you can save or restore an entire array of `SCircle` objects with a single statement. To save the array:

```
SCircle[] circles = new SCircles[100];
...
out.writeObject(circles);
```

And to retrieve it later, perhaps from another program:

```
SCircle[] laterCircles = (SCircle[])in.readObject();
```

What is even more impressive is that Java's serialization works for linked structures. You can save an entire binary search tree or a graph using a single `writeObject` statement, and later restore it using a single `readObject` statement. You can do this even for reference-based implementations. The tree and graph retain both their information and their structure. For example, if the `writeObject` statement is invoked on an object `tree`, Java follows all references that start with `tree` and lead to other objects, and saves those objects along with the `tree` object. Of course, all the objects involved must implement the `Serializable` interface.

How does Java recreate the structure of `tree`? The run-time engine follows a careful approach of numbering (i.e., serializing) each object involved in a write operation and using those numbers in place of references when saving the information. Therefore, when it reloads the information, it can recreate the structure even though it places the information in new memory locations.

Summary

In this chapter we discussed two data structures: priority queues and graphs. For the former we saw an elegant implementation based on a binary tree with special shape and order properties. For the latter we saw a time-efficient array-based implementation and discussed a space-efficient reference-based implementation. Time and space efficiency tradeoffs are often the key to choosing alternate implementations of a data structure.

Graphs are the most complex structure we studied. They are very versatile and are a good way to model many real-world objects and situations. Because there are many different types of applications for graphs, there are all kinds of variations and generalizations of their definitions and implementations. Many advanced algorithms for manipulating and traversing graphs have been discovered. They are generally covered in detail in advanced computer science courses on algorithms.

Finally, we discussed the topic of saving objects and data structures in files and later retrieving them. We introduced Java's serialization tools that support these operations.

Summary of Classes and Support Files

The classes and files are listed in the order in which they appear in the text. Inner classes are not included.

The package a class belongs to, if any, is listed in parenthesis under Notes. The class and support files are available on our web site. They can be found in the `ch09` subdirectory of the `bookFiles` directory.

Classes, Interfaces, and Support Files Defined in Chapter 9

File	1st Ref.	Notes
`PriQueueInterface`	page 613	(`ch09.priorityQueues`) Specification of a Priority Queue ADT
`PriQUnderflowException`	page 614	(`ch09.priorityQueues`) Exception raised when attempt to dequeue from an empty PQ
`PriQOverflowException`	page 614	(`ch09.priorityQueues`) Exception raised when attempt to enqueue to a full PQ
`Heap`	page 620	(`ch09.priorityQueues`) Implements Priority Queue
`WeightedGraphInterface`	page 634	(`ch09.graphs`) Specification of a Weighted Graph ADT
`UseGraph`	page 638	Contains `depthFirstSearch`, `breadthFirst-Search`, and `shortestPaths` methods
`WeightedGraph`	page 649	(`ch09.graphs`) Partial implementation of an array-based Weighted Graph—completion is left as an exercise
`Circle.java`	page 655	(`ch09.circles`) First used in Chapter 2
`SaveCircle.java`	page 655	Saves the information of a `Circle` object in a text file
`GetCircle.java`	page 656	Retrieves the `Circle` object saved by `SaveCircle`
`SCircle.java`	page 660	(`ch09.circles`) A serializable circle
`SaveSCircle.java`	page 661	Saves a `SCircle` object in a file of serialized objects
`GetSCircle.java`	page 661	Retrieves the `SCircle` object saved by `SaveSCircle`

At the top of the next page is a list of the Java Library classes and interfaces that were used in this chapter for the first time in the textbook. The classes are listed in the order in which they are first used. Note that in some classes the methods listed might not be defined directly in the class; they might be defined in one of its superclasses. With the methods we also list constructors, if appropriate. For more information about the library classes and methods, the reader can check Sun's Java documentation.

Library Classes Used in Chapter 9 for the First Time

Class/Interface Name	Package	Overview	Methods Used	Where Used
`Serializable.java`	`io`	Objects of classes that implement this interface can be easily saved to files	None	Section 9.4
`ObjectOutputStream.java`	`io`	Used for saving serializable objects	`writeObject`	Section 9.4
`ObjectInputStream.java`	`io`	Used for retrieving serializable objects	`readObject`	Section 9.4

Exercises

9.1 Priority Queues

1. A priority queue is implemented as a linked list, sorted from largest to smallest element.
   a. Write the declarations for the instance variables, inner classes, and so forth, needed for this implementation.
   b. Write the `enqueue` operation, using this implementation.
   c. Write the `dequeue` operation, using this implementation.
2. A priority queue is implemented as a binary search tree.
   a. Write the declarations for the instance variables, inner classes, and so forth, needed for this implementation.
   b. Write the `enqueue` operation, using this implementation.
   c. Write the `dequeue` operation, using this implementation.
3. A priority queue is implemented as a sequential array-based list. The highest-priority item is in the first array position, the second-highest priority item is in the second array position, and so on.
   a. Write the declarations for the instance variables, inner classes, and so forth, needed for this implementation.
   b. Write the `enqueue` operation, using this implementation.
   c. Write the `dequeue` operation, using this implementation.
4. A stack is implemented using a priority queue. Each element is time-stamped as it is put into the stack. (The time stamp is a number between 0 and `Integer.MAX_VALUE`. Each time an element is pushed onto the stack, it is assigned the next largest number.)

a. What is the highest-priority element?

b. Write the `push`, `top`, and `pop` algorithms, using the specifications in Chapter 4.

5. A FIFO queue is implemented using a priority queue. Each element is time-stamped as it is put into the queue. (The time stamp is a number between 0 and `Integer.MAX_VALUE`. Each time an element is enqueued, it is assigned the next largest number.)

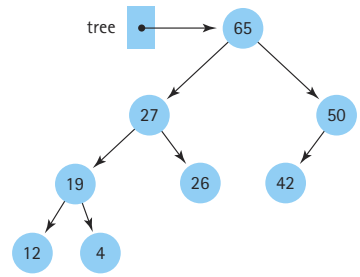
a. What is the highest-priority element?

b. Write the `enqueue` and `dequeue` operations, using the specifications in Chapter 4.

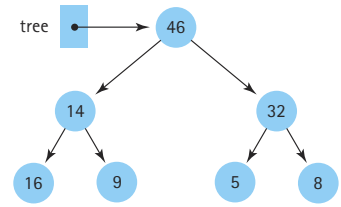
9.2 Heaps

6. Which of the following trees are heaps?

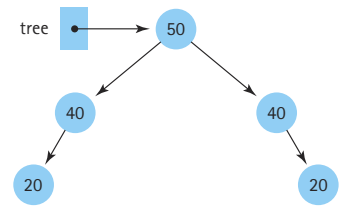
(a)



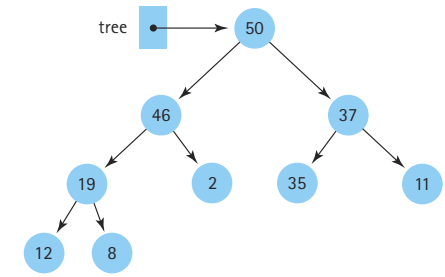
(b)



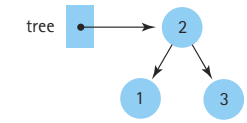
(c)



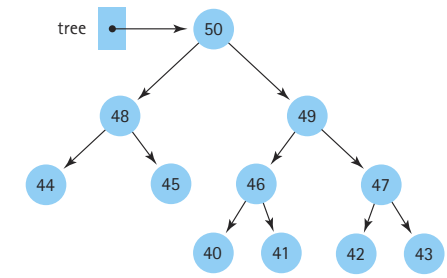
(d)



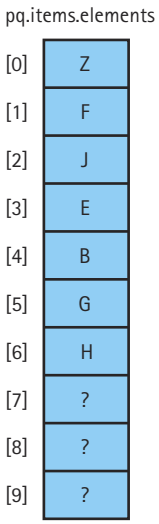
(e)



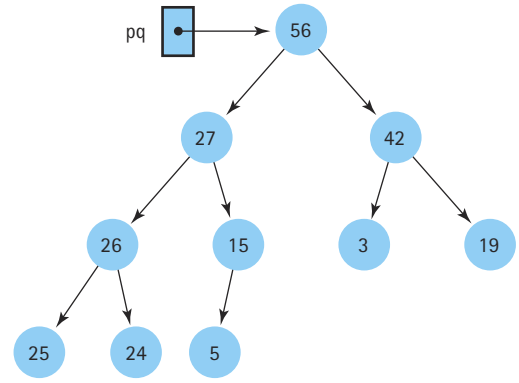
(f)



7. Draw a tree that satisfies both the binary search property and the order property of heaps.
8. A minimum heap has the following order property: The value of each element is less than or equal to the value of each of its children. What changes must be made in the heap operations given in this chapter?
9. We created iterative versions of the heap helper methods `reheapDown` and `reheapUp` in this chapter.
   a. Write a recursive version of `reheapDown`.
   b. Write a recursive version of `reheapUp`.
   c. Describe the recursive versions of these operations in terms of Big-O.
10. A priority queue containing characters is implemented as a heap stored in an array. The precondition states that this priority queue cannot contain duplicate elements. There are ten elements currently in the priority queue below. What values might be stored in array positions 7–9 so that the properties of a heap are satisfied?



11. A priority queue is implemented as a heap:

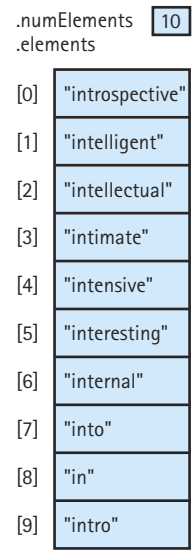


a. Show how the heap above would look after this series of operations:

```
pq.Enqueue(28);
pq.Enqueue(2);
pq.Enqueue(40);
pq.Dequeue(x);
pq.Dequeue(y);
pq.Dequeue(z);
```

b. What would the values of `x`, `y`, and `z` be after the series of operations in part (a)?

12. A priority queue of strings is implemented using a heap. The heap contains the following elements:



a. What feature of these strings is used to determine their priority in the priority queue?

b. Show how this priority queue is affected by adding the string "interviewing".

13. Write and compare two implementations of a priority queue whose highest priority element is the one with the smallest key value. The first implementation uses a minimum heap. You need to modify the heap operations to keep the minimum, rather than maximum, element in the root. The comparison function should compare key values. The second implementation uses a linear linked list whose elements are ordered by key value. Create a data set that contains 50 items with priorities generated by a random-number generator. To compare the operations, you must modify the `enqueue` and `dequeue` operations to count how many elements are accessed (compared or swapped, in the case of reheaping) during its execution. Write a driver to Enqueue and Dequeue the 50 test items and print out the number of elements accessed for the operations. Run your driver once with each implementation.

Deliverables

- A listing of specification and implementation files for both priority queue implementations
- A listing of your driver
- A listing of your test data
- A listing of the output from both runs
- A report comparing the number of elements accessed in executing each operation.

9.3 Graphs

Use the following description of an undirected graph in Exercises 14–17.

EmployeeGraph = (V, E)

V(EmployeeGraph) = {Susan, Darlene, Mike, Fred, John, Sander, Lance, Jean, Brent, Fran}

E(EmployeeGraph) = {(Susan, Darlene), (Fred, Brent), (Sander, Susan), (Lance, Fran), (Sander, Fran), (Fran, John), (Lance, Jean), (Jean, Susan), (Mike, Darlene), (Brent, Lance), (Susan, John)}

14. Draw a picture of `EmployeeGraph`.

15. Draw `EmployeeGraph` implemented as an adjacency matrix. Store the vertex values in alphabetical order.

16. Using the adjacency matrix for `EmployeeGraph` from Exercise 14, describe the path from Susan to Lance

a. using a breadth-first strategy.

b. using a depth-first strategy.

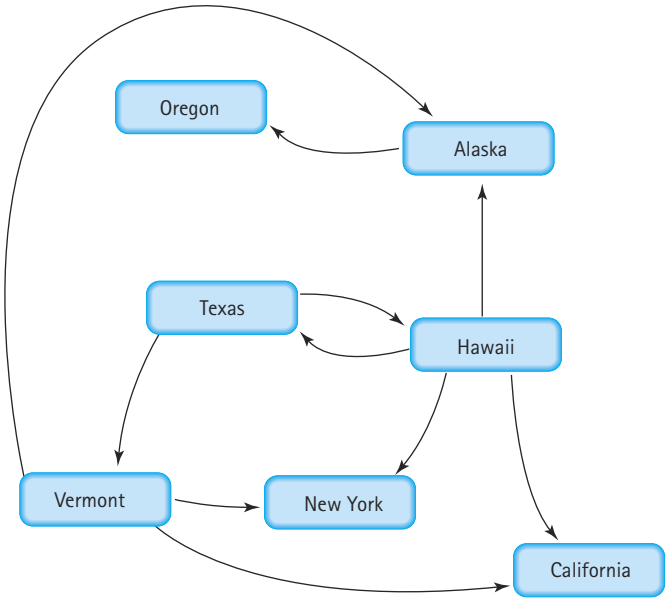
17. Which one of the following phrases best describes the relationship represented by the edges between the vertices in `EmployeeGraph`?
    a. "works for"
    b. "is the supervisor of"
    c. "is senior to"
    d. "works with"

Use the following specification of a directed graph in Exercises 18–21.

ZooGraph = (V, E)
V(ZooGraph) = {dog, cat, animal, vertebrate, oyster, shellfish, invertebrate, crab, poodle, monkey, banana, dalmatian, dachshund}
E(ZooGraph) = {(vertebrate, animal), (invertebrate, animal), (dog, vertebrate), (cat, vertebrate), (monkey, vertebrate), (shellfish, invertebrate), (crab, shellfish), (oyster, shellfish), (poodle, dog), (dalmatian, dog), (dachshund, dog)}

18. Draw a picture of `ZooGraph`.
19. Draw the adjacency matrix for `ZooGraph`. Store the vertices in alphabetical order.
20. To tell if one element in `ZooGraph` has relation X to another element, you look for a path between them. Show whether the following statements are true, using the picture or adjacency matrix.
    a. dalmatian X dog
    b. dalmatian X vertebrate
    c. dalmatian X poodle
    d. banana X invertebrate
    e. oyster X invertebrate
    f. monkey X invertebrate
21. Which of the following phrases best describes relation X in the previous question?
    a. "has a"
    b. "is an example of"
    c. "is a generalization of"
    d. "eats"

Use the following graph for Exercises 22–24:



22. Describe the graph pictured above, using the formal graph notation.

    V(StateGraph) =

    E(StateGraph) =

23. In the states graph:
    a. Is there a path from Oregon to any other state in the graph?
    b. Is there a path from Hawaii to every other state in the graph?
    c. From which state(s) in the graph is there a path to Hawaii?

24. Graphs can be implemented using arrays or references.
    a. Show the adjacency matrix that would describe the edges in this graph. Store the vertices in alphabetical order.
    b. Show the array-of-references adjacency lists that would describe the edges in this graph.

25. Complete the implementation of the Weighted Graph that we began in this chapter by providing bodies for the methods `isEmpty`, `isFull`, `clearMarks`, `markVertex`, and `isMarked` in the `WeightedGraph.java` file. Test the completed implementation using the `UseGraph` class.

26. Class `WeightedGraph` in this chapter is to be extended to include a `boolean edgeExists` operation, which determines whether two vertices are connected by an edge.
    a. Write the declaration of this method. Include adequate comments.
    b. Using the adjacency matrix implementation developed in the chapter and the declaration from part (a), implement the body of the method.
27. Class `WeightedGraph` in this chapter is to be extended to include a `deleteEdge` operation, which deletes a given edge.
    a. Write the declaration of this method. Include adequate comments.
    b. Using the adjacency matrix implementation developed in the chapter and the declaration from part (a), implement the body of the method.
28. Class `WeightedGraph` in this chapter is to be extended to include a `deleteVertex` operation, which deletes a vertex from the graph. Deleting a vertex is more complicated than deleting an edge from the graph. Discuss the reasons.
29. The `depthFirstSearch` operation can be implemented without a stack by using recursion.
    a. Name the base case(s). Name the general case(s).
    b. Write the algorithm for a recursive depth-first search.

9.4 Storing Objects/Structures in Files

30. If you wanted to traverse a binary search tree, writing all the elements to a file, and later (the next time you run the program) rebuild the tree by reading the elements and inserting them into the tree using the `insert` method, would an inorder traversal be appropriate? Why or why not?
31. We want to serialize our binary search trees.
    a. What changes would you make to the binary search tree file `BinarySearchTree.java` to allow binary search trees to be serialized?
    b. How could you test your changes?
    c. Make and test the changes.
32. Expand the Word Frequency Generator program from the case study in Chapter 8, and any associated files that also have to be changed, so that it accepts a third file name as a parameter and uses that file to hold a serialized copy of the binary search tree between runs of the program. This means that you can run the program on one input file of text to create the tree, and later run it on another input file of text and have the second file's words added to the tree containing the first file's words. Discuss rules for using this program.

Chapter 10

Sorting and Searching Algorithms

Goals

Measurable goals for this chapter include that you should be able to

- design and implement the following sorting algorithms:
  - straight selection sort
  - bubble sort (two versions)
  - insertion sort
  - quick sort
  - merge sort
  - heap sort
- compare the efficiency of the sorting algorithms, in terms of Big-O time and space requirements
- discuss other efficiency considerations: sorting small numbers of elements, programmer time
- use the Java `Comparator` interface to define multiple sort orders for objects of a class
- discuss the performances of the following search algorithms:
  - sequential search of an unsorted list
  - sequential search of a sorted list
  - binary search
  - searching a high-probability sorted list
- define the following terms:
  - hashing
  - rehashing
  - collisions
  - linear probing
  - clustering
- design and implement an appropriate hashing function for an application
- design and implement a collision-resolution algorithm for a hash table
- discuss the efficiency considerations for the searching and hashing algorithms, in terms of Big-O

At many points in this book, we have gone to great trouble to keep lists of elements in sorted order: real estate information sorted by house ID number, airline routes sorted by distance, integers sorted from smallest to largest, words sorted alphabetically. One goal of keeping sorted lists, of course, is to facilitate searching; given an appropriate implementation structure, a particular list element can be found faster if the list is sorted.

In this chapter we directly examine strategies for both sorting and searching, two tasks that are fundamental to a variety of computing problems.

10.1 Sorting

Putting an unsorted list of data elements into order—sorting—is a very common and useful operation. Whole books have been written about various sorting algorithms, as well as algorithms for searching a sorted list to find a particular element. The goal is to come up with better, more efficient, sorts. Because sorting a large number of elements can be extremely time consuming, a good sorting algorithm is very desirable. This is one area in which programmers are sometimes encouraged to sacrifice clarity in favor of speed of execution.

How do we describe efficiency? We pick an operation central to most sorting algorithms: the operation that compares two values to see which is smaller. In our study of sorting algorithms, we relate the number of comparisons to the number of elements in the list (N) as a rough measure of the efficiency of each algorithm. The number of element swaps is another measure of sorting efficiency. In the exercises we ask you to analyze the sorting algorithms developed in this chapter in terms of data movements.

Another efficiency consideration is the amount of memory space required. In general, memory space is not a very important factor in choosing a sorting algorithm. We look at only two sorts in which space would be a consideration. The usual time versus space tradeoff applies to sorts—more space often means less time, and vice versa.

Because processing time is the factor that applies most often to sorting algorithms, we consider it in detail here. Of course, as in any application, the programmer must determine goals and requirements before selecting an algorithm and starting to code.

We first discuss the straight selection sort, the bubble sort, and the insertion sort, three simple sorts that students sometimes write in their first course. Then we introduce three more complex (but more efficient) sorting algorithms: merge sort, quick sort, and heap sort. So that we can concentrate on the algorithms, during the initial discussions, we assume that our goal is to sort a given list of integers, held in an array. At the logical level, sorting algorithms take an unsorted list and convert it into a sorted list. At the implementation level, the algorithms take an array and reorganize the values in the array so that they are in order by key. Although we may say that we are "sorting the array" or that we have a "sorted array," remember that it is actually the list represented by the array that is being sorted. In Section 10.4 we address issues related to sorting objects in general.

A Test Harness

To facilitate our study of sorting we develop a standard test harness, a driver program that we can use to test each of our sorting algorithms. Since we are using this program just to test our implementations and facilitate our study, we keep it simple. It consists of a single class called `Sorts`. The class defines an array of integers of size 50. The array is named `values`. Several static methods are defined:

Test harness A standalone program designed to facilitate testing of the implementations of algorithms

- `initValues`: Initializes the `values` array with random numbers between 0 and 99; uses the `abs` method from the Java library's `Math` class (absolute value) and the `nextInt` method from the `Random` class.
- `isSorted`: Returns a `boolean` value indicating whether or not the `values` array is currently sorted.
- `swap`: Swapping data values between two array locations is common in many sorting algorithms—this method swaps the integers between `values[index1]` and `values[index2]`, where `index1` and `index2` are parameters of the method.
- `printValues`: Prints the contents of the `values` array to the `System.out` stream; the output is arranged evenly in ten columns.

Here is the code for the test harness:

```
//--------------------------------------------------------------------------
// Sorts.java               by Dale/Joyce/Weems                  Chapter 10
//
// Test harness used to run sorting algorithms
//--------------------------------------------------------------------------

import java.io.*;
import java.util.*;
import java.text.DecimalFormat;

public class Sorts
{
  static final int SIZE = 50;            // Size of array to be sorted
  static int[] values = new int[SIZE];   // Values to be sorted

  static void initValues()
  // Initializes the values array with random integers from 0 to 99
  {
    Random rand = new Random();
    for (int index = 0; index < SIZE; index++)
      values[index] = Math.abs(rand.nextInt()) % 100;
  }
```

```
static public boolean isSorted()
// Determines whether the array values are sorted
{
  boolean sorted = true;
  for (int index = 0; index < (SIZE - 1); index++)
    if (values[index] > values[index + 1])
      sorted = false;
  return sorted;
}

static public void swap(int index1, int index2)
// Swaps the integers at locations index1 and index2 of array values
// Precondition: index1 and index2 are less than size
{
  int temp = values[index1];
  values[index1] = values[index2];
  values[index2] = temp;
}

static public void printValues()
// Prints all the values integers
{
  int value;
  DecimalFormat fmt = new DecimalFormat("00");
  System.out.println("the values array is:");
  for (int index = 0; index < SIZE; index++)
  {
    value = values[index];
    if (((index + 1) % 10) == 0)
      System.out.println(fmt.format(value));
    else
      System.out.print(fmt.format(value) + " ");
  }
  System.out.println();
}

public static void main(String[] args) throws IOException
{
  initValues();
  printValues();
  System.out.println("values is sorted: " + isSorted());
  System.out.println();

  swap(0, 1);
```

```
    printValues();
    System.out.println("values is sorted: " + isSorted());
    System.out.println();
  }
}
```

In this version of `Sorts` the main method initializes the `values` array, prints it, prints the value of `isSorted`, swaps the first two values of the array, and again prints information about the array. The output from this class as currently defined would look something like this:

```
the values array is:
20 49 07 50 45 69 20 07 88 02
89 87 35 98 23 98 61 03 75 48
25 81 97 79 40 78 47 56 24 07
63 39 52 80 11 63 51 45 25 78
35 62 72 05 98 83 05 14 30 23

values is sorted: false

the values array is:
49 20 07 50 45 69 20 07 88 02
89 87 35 98 23 98 61 03 75 48
25 81 97 79 40 78 47 56 24 07
63 39 52 80 11 63 51 45 25 78
35 62 72 05 98 83 05 14 30 23

values is sorted: false
```

As we proceed in our study of sorting algorithms, we add methods that implement the algorithms to the `Sorts` class and change the main method to invoke those methods. We can use the `isSorted` and `printValues` methods to help us check the results.

Since our sorting methods are implemented for use with this test harness, they can directly access the static `values` array. In the general case, we could modify each method to accept a reference, to an array-based list to be sorted, as a parameter.

10.2 Simple Sorts

In this section we present three "simple" sorts, so called because they use an unsophisticated brute force approach. This means they are not very efficient; but they are easy to understand and to implement.

Straight Selection Sort

If you were handed a list of names and asked to put them in alphabetical order, you might use this general approach:

1. Find the name that comes first in alphabetical order, and write it on a second sheet of paper.
2. Cross the name out on the original list.
3. Continue this cycle until all the names on the original list have been crossed out and written onto the second list, at which point the second list is sorted.

This algorithm is simple to translate into a computer program, but it has one drawback: It requires space in memory to store two complete lists. Although we have not talked a great deal about memory-space considerations, this duplication is clearly wasteful. A slight adjustment to this manual approach does away with the need to duplicate space, however. As you cross a name off the original list, a free space opens up. Instead of writing the minimum value on a second list, you can exchange it with the value currently in the position where the crossed-off item should go. Our "by-hand list" is represented in an array. Let's look at an example—sorting the five-element array shown in Figure 10.1(a). Because of this algorithm's simplicity, it is usually the first sorting method that students learn.

SelectionSort

```
Set current to the index of first item in the array
while more items in unsorted part of array
    Find the index of the smallest unsorted item
    Swap the current item with the smallest unsorted one
    Shrink the unsorted part of the array by incrementing current
```

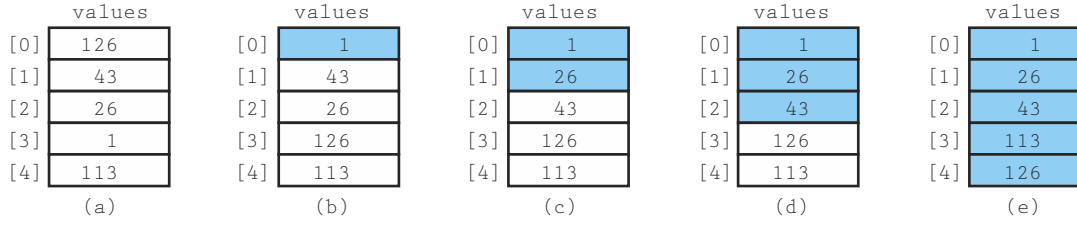


Figure 10.1 *Example of a straight selection sort (sorted elements are shaded)*

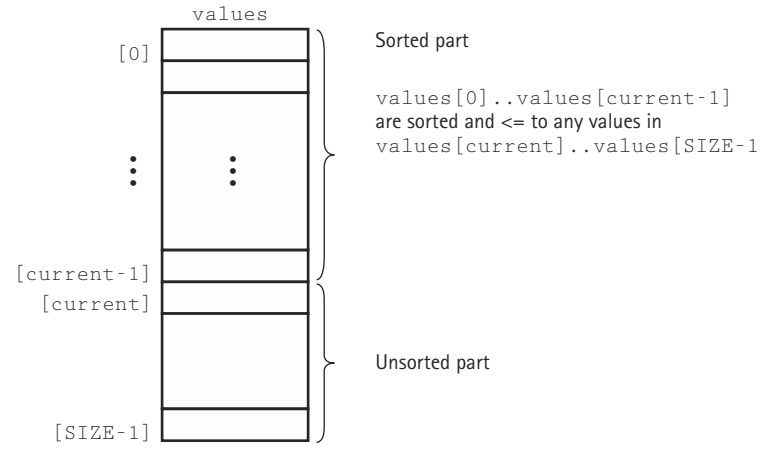


Figure 10.2 *A snapshot of the selection sort algorithm*

We implement the algorithm with a method `selectionSort` that is part of our `Sorts` class. This method sorts the `values` array, declared in that class, and therefore has access to the `SIZE` constant that indicates the number of elements in the array. Within the `selectionSort` method we use a variable, `current`, to mark the beginning of the unsorted part of the array. This means that the unsorted part of the array goes from index `current` to index `SIZE - 1`. We start out by setting `current` to the index of the first position (0).

The main sort processing is in a loop. In each iteration of the loop body, the smallest value in the unsorted part of the array is swapped with the value in the `current` location. After the swap, `current` is in the sorted part of the array, so we shrink the size of the unsorted part by incrementing `current`. The loop body is now complete.

Back at the top of the loop body, the unsorted part of the array goes from the (now incremented) `current` index to position `SIZE - 1`. We know that every value in the unsorted part is greater than or equal to any value in the sorted part of the array.

How do we know when "there are more elements in the unsorted part"? As long as `current <= SIZE - 1`, the unsorted part of the array (`values[current] .. values[SIZE - 1]`) contains values. In each iteration of the loop body, `current` is incremented, shrinking the unsorted part of the array. When `current = SIZE - 1`, the "unsorted" part contains only one element, and we know that this value is greater than or equal to any value in the sorted part. So the value in `values[SIZE - 1]` is in its correct place, and we are done. The condition for the *while* loop is `current < SIZE - 1`. A snapshot of the selection sort algorithm is illustrated in Figure 10.2.

Now all we have to do is locate the smallest value in the unsorted part of the array. Let's write a method to do this task. The `minIndex` method receives first and last indexes of the unsorted part, and returns the index of the smallest value in this part of the array.

minIndex(startIndex, endIndex): return int

```
Set indexOfMin to startIndex
for index going from startIndex + 1 to endIndex
    if values[index] < values[indexOfMin]
        Set indexOfMin to index
return indexOfMin
```

Now that we know where the smallest unsorted element is, we swap it with the element at index `current`. We use the `swap` method that is part of our test harness. Here is the code for the `minIndex` and `selectionSort` methods. Since they are placed directly in our test harness class, a class with a main method, they are declared as static methods.

```
static int minIndex(int startIndex, int endIndex)
// Post: Returns the index of the smallest value in
//       values[startIndex]..values[endIndex]
{
  int indexOfMin = startIndex;
  for (int index = startIndex + 1; index <= endIndex; index++)
    if (values[index] < values[indexOfMin])
      indexOfMin = index;
  return indexOfMin;
}

static void selectionSort()
// Post: The elements in the array values are sorted
{
  int endIndex = SIZE - 1;
  for (int current = 0; current < endIndex; current++)
    swap(current, minIndex(current, endIndex));
}
```

If we now change the main body of the test harness to:

```
initValues();
printValues();
System.out.println("values is sorted: " + isSorted());
System.out.println();

selectionSort();
```

```
printValues();
System.out.println("values is sorted: " + isSorted());
System.out.println();
```

we get an output from the program that looks like this:

```
the values array is:
92 66 38 17 21 78 10 43 69 19
17 96 29 19 77 24 47 01 97 91
13 33 84 93 49 85 09 54 13 06
21 21 93 49 67 42 25 29 05 74
96 82 26 25 11 74 03 76 29 10

values is sorted: false

the values array is:
01 03 05 06 09 10 10 11 13 13
17 17 19 19 21 21 21 24 25 25
26 29 29 29 33 38 42 43 47 49
49 54 66 67 69 74 74 76 77 78
82 84 85 91 92 93 93 96 96 97

values is sorted: true
```

We can test all of our sorting methods using this same approach.

Analyzing Selection Sort

Now let's try measuring the amount of "work" required by this algorithm. We describe the number of comparisons as a function of the number of items in the array, i.e., `SIZE`. To be concise, in this discussion we refer to `SIZE` as *N*.

The comparison operation is in the `minIndex` method. We know from the loop condition in the `selectionSort` method that `minIndex` is called $N - 1$ times. Within `minIndex`, the number of comparisons varies, depending on the values of `startIndex` and `endIndex`:

```
for (int index = startIndex + 1; index <= endIndex; index++)
  if (values[index] < values[indexOfMin])
    indexOfMin = index;
```

In the first call to `minIndex`, `startIndex` is 0 and `endIndex` is `SIZE - 1`, so there are $N - 1$ comparisons; in the next call there are $N - 2$ comparisons, and so on, until in the last call, when there is only one comparison. The total number of comparisons is

$$(N - 1) + (N - 2) + (N - 3) + \ldots + 1 = N(N - 1)/2$$

Table 10.1 *Number of Comparisons Required to Sort Arrays of Different Sizes Using Selection Sort*

Number of Items	Number of Comparisons
10	45
20	190
100	4,950
1,000	499,500
10,000	49,995,000

To accomplish our goal of sorting an array of N elements, the straight selection sort requires $N(N - 1)/2$ comparisons. Note that the particular arrangement of values in the array does not affect the amount of work done at all. Even if the array is in sorted order before the call to `selectionSort`, the method still makes $N(N - 1)/2$ comparisons. Table 10.1 shows the number of comparisons required for arrays of various sizes. Note that doubling the array size roughly quadruples the number of comparisons.

How do we describe this algorithm in terms of Big-O? If we express $N(N - 1)/2$ as $1/2N^2 - 1/2N$, it is easy to see. In Big-O notation we only consider the term $1/2N^2$, because it increases fastest relative to N. (Remember the elephant and goldfish?) Further, we ignore the constant, $1/2$, making this algorithm $O(N^2)$. This means that, for large values of N, the computation time is *approximately proportional* to N^2. Looking back at the previous table, we see that multiplying the number of elements by 10 increases the number of comparisons by a factor of more than 100; that is, the number of comparisons is multiplied by approximately the square of the increase in the number of elements. Looking at this chart makes us appreciate why sorting algorithms are the subject of so much attention: Using `selectionSort` to sort an array of 1,000 elements requires almost a half million comparisons!

The identifying feature of a selection sort is that, on each pass through the loop, one element is put into its proper place. In the straight selection sort, each iteration finds the smallest unsorted element and puts it into its correct place. If we had made the helper method find the largest value instead of the smallest, the algorithm would have sorted in descending order. We could also have made the loop go down from `SIZE - 1` to 1, putting the elements into the end of the array first. All these are variations on the straight selection sort. The variations do not change the basic way that the minimum (or maximum) element is found.

Bubble Sort

The bubble sort is a sort that uses a different scheme for finding the minimum (or maximum) value. Each iteration puts the smallest unsorted element into its correct place, but it also makes changes in the locations of the other elements in the array. The first itera-

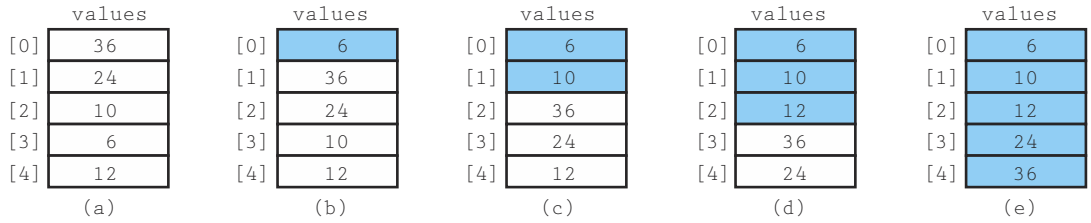


Figure 10.3 *Example of bubble sort (sorted elements are shaded)*

tion puts the smallest element in the array into the first array position. Starting with the last array element, we compare successive pairs of elements, swapping whenever the bottom element of the pair is smaller than the one above it. In this way the smallest element "bubbles up" to the top of the array. The next iteration puts the smallest element in the unsorted part of the array into the second array position, using the same technique. As you look at the example in Figure 10.3, note that in addition to putting one element into its proper place, each iteration causes some intermediate changes in the array.

The basic algorithm for the bubble sort is

BubbleSort

Set current to the index of first item in the array
while more items in unsorted part of array
    "Bubble up" the smallest item in the unsorted part,
        causing intermediate swaps as needed
    Shrink the unsorted part of the array by incrementing current

The structure of the loop is much like that of the `selectionSort`. The unsorted part of the array is the area from `values[current]` to `values[SIZE - 1]`. The value of `current` begins at 0, and we loop until `current` reaches `SIZE - 1`, with `current` incremented in each iteration. On entrance to each iteration of the loop body, the first `current` values are already sorted, and all the elements in the unsorted part of the array are greater than or equal to the sorted elements.

The inside of the loop body is different, however. Each iteration of the loop "bubbles up" the smallest value in the unsorted part of the array to the `current` position. The algorithm for the bubbling task is

bubbleUp(startIndex, endIndex)

```
for index going from endIndex DOWNTO startIndex +1
    if values[index] < values[index − 1]
        Swap the value at index with the value at index − 1
```

A snapshot of this algorithm is shown in Figure 10.4. We use the `swap` method as before. The code for methods `bubbleUp` and `bubbleSort` follows. The code can be tested using our test harness.

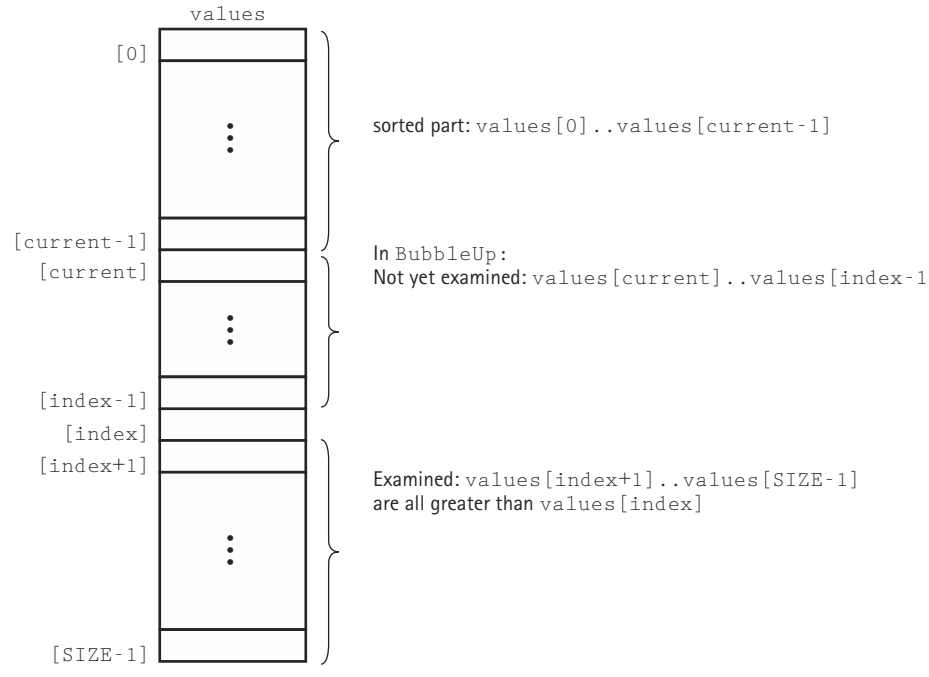


Figure 10.4 *Snapshot of a bubble sort*

```
static void bubbleUp(int startIndex, int endIndex)
  // Post: Adjacent pairs that are out of order have been switched
  //       between values[startIndex]..values[endIndex] beginning at
  //       values[endIndex]
  {
    for (int index = endIndex; index > startIndex; index--)
      if (values[index] < values[index - 1])
        swap(index, index - 1);
  }

  static void bubbleSort()
  // Post: The elements in the array values are sorted
  {
    int current = 0;

    while (current < SIZE - 1)
    {
      bubbleUp(current, SIZE - 1);
      current++;
    }
  }
```

Analyzing Bubble Sort

To analyze the work required by `bubbleSort` is easy. It is the same as for the straight selection sort algorithm. The comparisons are in `bubbleUp`, which is called $N - 1$ times. There are $N - 1$ comparisons the first time, $N - 2$ comparisons the second time, and so on. Therefore, `bubbleSort` and `selectionSort` require the same amount of work in terms of the number of comparisons. `bubbleSort` does more than just make comparisons though; `selectionSort` has only one data swap per iteration, but `bubbleSort` may do many additional data swaps.

What is the purpose of these intermediate data swaps? By reversing out-of-order pairs of data as they are noticed, the method might get the array in order before $N - 1$ calls to `bubbleUp`. However, this version of the bubble sort makes no provision for stopping when the array is completely sorted. Even if the array is already in sorted order when `bubbleSort` is called, this method continues to call `bubbleUp` (which changes nothing) $N - 1$ times.

We could quit before the maximum number of iterations if `bubbleUp` returns a `boolean` flag, to tell us when the array is sorted. Within `bubbleUp`, we initially set a variable `sorted` to `true`; then in the loop, if any swaps are made, we reset `sorted` to `false`. If no elements have been swapped, we know that the array is already in order. Now the bubble sort only needs to make one extra call to `bubbleUp` when the array is in order. This version of the bubble sort is as follows:

```
static boolean bubbleUp2(int startIndex, int endIndex)
// Post: Adjacent pairs that are out of order have been switched
//       between values[startIndex]..values[endIndex] beginning at
//       values[endIndex]
//       Returns false if a swap was made; otherwise, true
{
  boolean sorted = true;
  for (int index = endIndex; index > startIndex; index--)
    if (values[index] < values[index - 1])
    {
      swap(index, index - 1);
      sorted = false;
    }
  return sorted;
}

static void shortBubble()
// Post: The elements in the array values are sorted by key
//       The process stops as soon as values is sorted
{
  int current = 0;
  boolean sorted = false;
  while (current < SIZE - 1 && !sorted)
  {
    sorted = bubbleUp2(current, SIZE - 1);
    current++;
  }
}
```

The analysis of `shortBubble` is more difficult. Clearly, if the array is already sorted to begin with, one call to `bubbleUp` tells us so. In this best-case scenario, `shortBubble` is $O(N)$; only $N - 1$ comparisons are required for the sort. What if the original array was actually sorted in descending order before the call to `shortBubble`? This is the worst possible case: `shortBubble` requires as many comparisons as `bubbleSort` and `selectionSort`, not to mention the "overhead"—all the extra swaps and setting and resetting the `sorted` flag. Can we calculate an average case? In the first call to `bubbleUp`, when current is 0, there are `SIZE` $-$ 1 comparisons; on the second call, when `current` is 1, there are `SIZE` $-$ 2 comparisons. The number of comparisons in any call to `bubbleUp` is `SIZE` $-$ `current` $-$ 1. If we let N indicate `SIZE` and K indicate the number of calls to `bubbleUp` executed before `shortBubble` finishes its work, the total number of comparisons required is

$$\underbrace{(N-1)}_{\text{1st call}} + \underbrace{(N-2)}_{\text{2nd call}} + \underbrace{(N-3)}_{\text{3rd call}} + \cdots + \underbrace{(N-K)}_{K\text{th call}}$$

A little algebra changes this to

$$(2KN - 2K^2 - K)/2$$

In Big-O notation, the term that is increasing the fastest relative to N is $2KN$. We know that K is between 1 and $N - 1$. On average, over all possible input orders, K is proportional to N. Therefore, $2KN$ is proportional to N^2; that is, the `shortBubble` algorithm is also $O(N^2)$.

Why do we even bother to mention the bubble sort algorithm if it is $O(N^2)$ and requires extra data movements? Due to the extra intermediate swaps performed by bubble sort, it can quickly sort an array that is "almost" sorted. If the `shortBubble` variation is used, bubble sort can be very efficient for this situation.

Insertion Sort

In Chapter 3, we created a sorted list by inserting each new element into its appropriate place in an array. We can use a similar approach for sorting an array. The principle of the insertion sort is quite simple: Each successive element in the array to be sorted is inserted into its proper place with respect to the other, already sorted elements. As with the previous sorts, we divide our array into a sorted part and an unsorted part. (Unlike the previous sorts, there may be values in the unsorted part that are less than values in the sorted part.) Initially, the sorted portion contains only one element: the first element in the array. Now we take the second element in the array and put it into its correct place in the sorted part; that is, `values[0]` and `values[1]` are in order with respect to each other. Now the value in `values[2]` is put into its proper place, so `values[0]..values[2]` are in order with respect to each other. This process continues until all the elements have been sorted. Figure 10.5 illustrates this process, which we describe in the following algorithm, and Figure 10.6 shows a snapshot of the algorithm.

In Chapter 3, our strategy was to search for the insertion point from the beginning of the array and shift the elements from the insertion point down one slot to make room for the new element. We can combine the searching and shifting by beginning at the end of the sorted part of the array. We compare the item at `values[current]` to the one before it. If it is less, we swap the two items. We then compare the item at `values[current - 1]` to the one before it, and swap if necessary. The process stops when

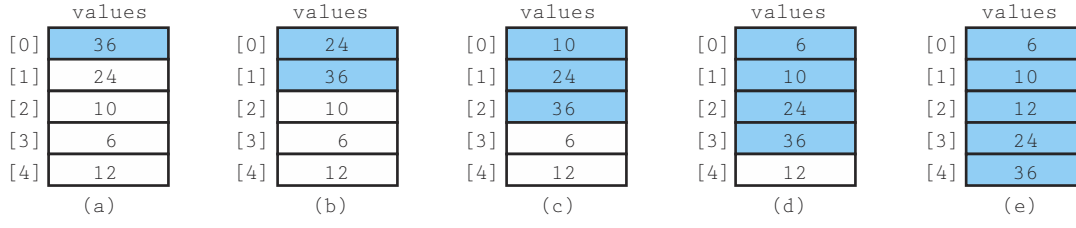


Figure 10.5 *Example of the insertion sort algorithm*

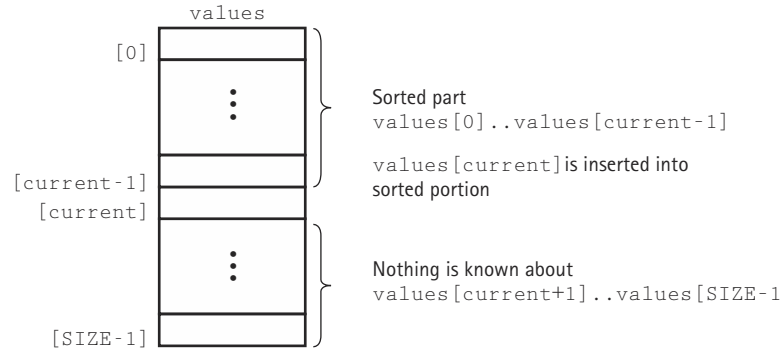


Figure 10.6 *A snapshot of the insertion sort algorithm*

the comparison shows that the values are in order or we have swapped into the first place in the array.

insertionSort

```
for count going from 1 through SIZE − 1
    insertItem(0, count)
```

InsertItem(startIndex, endIndex)

```
Set finished to false
Set current to endIndex
Set moreToSearch to true
while moreToSearch AND NOT finished
    if values[current] < values[current − 1]
        swap(values[current], values[current − 1])
        Decrement current
        Set moreToSearch to (current does not equal startIndex)
    else
        Set finished to true
```

Here are the coded versions of `insertItem` and `insertionSort`.

```
static void insertItem(int startIndex, int endIndex)
// Post: values[0]..values[endIndex] are now sorted
{
  boolean finished = false;
  int current = endIndex;
  boolean moreToSearch = true;
  while (moreToSearch && !finished)
  {
    if (values[current] < values[current - 1])
    {
      swap(current, current - 1);
      current--;
      moreToSearch = (current != startIndex);
    }
    else
      finished = true;
  }
}

static void insertionSort()
// Post: The elements in the array values are sorted by key
{
  for (int count = 1; count < SIZE; count++)
    insertItem(0, count);
}
```

Analyzing Insertion Sort

The general case for this algorithm mirrors the `selectionSort` and the `bubbleSort`, so the general case is O(N^2). But like `shortBubble`, `insertionSort` has a best case: The data are already sorted in ascending order. When the data are in ascending order, `insertItem` is called N times, but only one comparison is made each time and no swaps are necessary. The maximum number of comparisons is made only when the elements in the array are in reverse order.

If we know nothing about the original order of the data to be sorted, `selectionSort`, `shortBubble`, and `insertionSort` are all O(N^2) sorts and are very time consuming for sorting large arrays. Thus, we need sorting methods that work better when N is large.

10.3 O(N log$_2$N) Sorts

Considering how rapidly N^2 grows as the size of the array increases, can't we do better? We note that N^2 is a lot larger than $(1/2N)^2 + (1/2N)^2$. If we could cut the array into two pieces, sort each segment, and then merge the two back together, we should end up

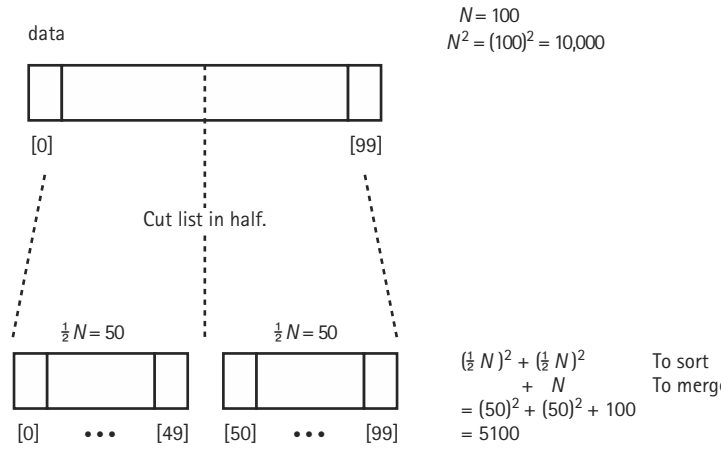


Figure 10.7 *Rationale for divide-and-conquer sorts*

sorting the entire array with a lot less work. An example of this approach is shown in Figure 10.7.

The idea of "divide and conquer" has been applied to the sorting problem in different ways, resulting in a number of algorithms that can do the job much more efficiently than $O(N^2)$. In fact, there is a category of sorting algorithms that are $O(N\log_2 N)$. We examine three of these algorithms here: `mergeSort`, `quickSort`, and `heapSort`. As you might guess, the efficiency of these algorithms is achieved at the expense of the simplicity seen in the straight selection, bubble, and insertion sorts.

Merge Sort

The merge sort algorithm is taken directly from the idea presented above.

mergeSort

Cut the array in half
Sort the left half
Sort the right half
Merge the two sorted halves into one sorted array

Merging the two halves together is a $O(N)$ task: We merely go through the sorted halves, comparing successive pairs of values (one in each half) and putting the smaller value into the next slot in the final solution. Even if the sorting algorithm used for each half is $O(N^2)$, we should see some improvement over sorting the whole array at once.

Actually, because `mergeSort` is itself a sorting algorithm, we might as well use it to sort the two halves. That's right—we can make `mergeSort` a recursive method and let it call itself to sort each of the two subarrays:

mergeSort—Recursive

```
Cut the array in half
mergeSort the left half
mergeSort the right half
Merge the two sorted halves into one sorted array
```

This is the general case, of course. What is the base case, the case that does not involve any recursive calls to `mergeSort`? If the "half" to be sorted doesn't have more than one element, we can consider it already sorted and just return.

Let's summarize `mergeSort` in the format we used for other recursive algorithms. The initial method call would be `mergeSort(0, SIZE - 1)`.

Method mergeSort(first, last)

Definition:	Sorts the array items in ascending order.
Size:	last − first + 1
Base Case:	If size less than 2, do nothing.
General Case:	Cut the array in half.
	mergeSort the left half.
	mergeSort the right half.
	Merge the sorted halves into one sorted array.

Cutting the array in half is simply a matter of finding the midpoint between the first and last indexes:

```
middle = (first + last) / 2;
```

Then, in the smaller-caller tradition, we can make the recursive calls to `mergeSort`:

```
mergeSort(first, middle);
mergeSort(middle + 1, last);
```

So far this is simple enough. Now we only have to merge the two halves and we're done.

Merging the Sorted Halves

Obviously, all the serious work is in the merge step. Let's first look at the general algorithm for merging two sorted arrays, and then we can look at the specific problem of our subarrays.

To merge two sorted arrays, we compare successive pairs of elements, one from each array, moving the smaller of each pair to the "final" array. We can stop when one array runs out of elements, and then move all the remaining elements from the other array to the final array. Figure 10.8 illustrates the general algorithm. We use a similar approach in our specific problem, in which the two "arrays" to be merged are actually subarrays of the original array (Figure 10.9). Just as in Figure 10.8, where we merged

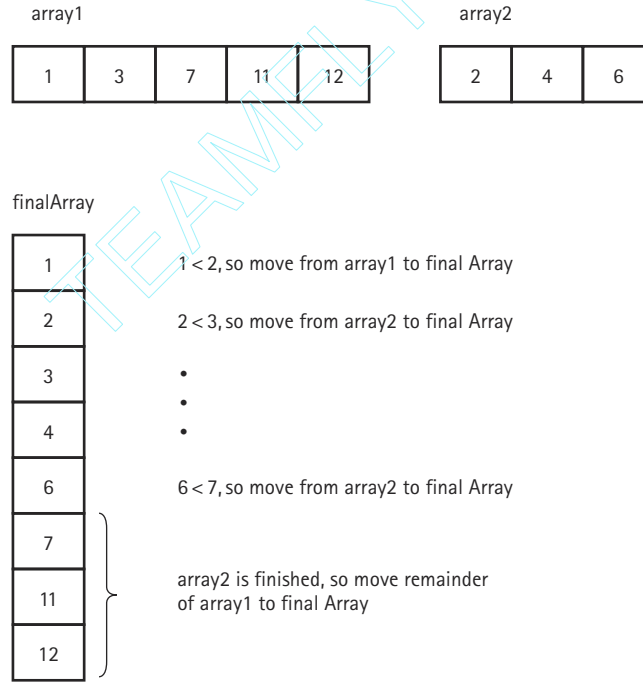


Figure 10.8 *Strategy for merging two sorted arrays*

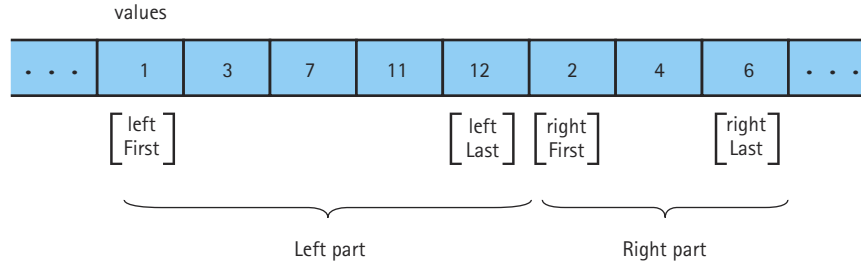


Figure 10.9 *Two subarrays*

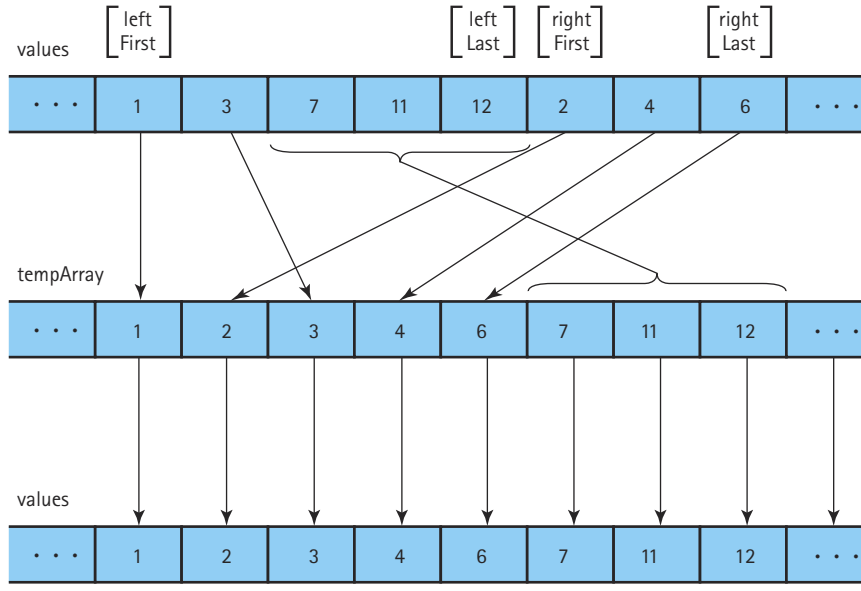


Figure 10.10 *Merging sorted halves*

array1 and array2 into a third array, we need to merge our two subarrays into some auxiliary structure. We only need this structure, another array, temporarily. After the merge step, we can copy the now-sorted elements back into the original array. The whole process is shown in Figure 10.10.

Let's specify a method, `merge`, to do this task:

merge(int leftFirst, int leftLast, int rightFirst, int rightLast)

Method:	Merges two sorted subarrays into a single sorted piece of the array
Preconditions:	values[leftFirst]..values[leftLast] are sorted; values[rightFirst]..values[rightLast] are sorted.
Postcondition:	values[leftFirst]..values[rightLast] are sorted.

Here is the algorithm for Merge:

merge (leftFirst, leftLast, rightFirst, rightLast)

(uses a local array, tempArray)

```
Set saveFirst to leftFirst      // To know where to copy back
Set index to leftFirst
while more items in left half AND more items in right half
      if values[leftFirst] < values[rightFirst]
            Set tempArray[index] to values[leftFirst]
            Increment leftFirst
      else
            Set tempArray[index] to values[rightFirst]
            Increment rightFirst
      Increment index
Copy any remaining items from left half to tempArray
Copy any remaining items from right half to tempArray
Copy the sorted elements from tempArray back into values
```

In the coding of method `merge`, we use `leftFirst` and `rightFirst` to indicate the "current" position in the left and right halves, respectively. Because these are values of the primitive type `int` and not objects, copies of these parameters are passed to method `merge`, rather than references to the parameters. These copies are changed in the method; but changing the copies does not affect the original values. Note that both of the "copy any remaining items" loops are included. During the execution of this method, one of these loops never executes. Can you explain why?

```
static void merge (int leftFirst, int leftLast, int rightFirst, int
rightLast)
// Post: values[leftFirst]..values[leftLast] and
//       values[rightFirst]..values[rightLast] have been merged.
//       values[leftFirst]..values[rightLast] are now sorted
{
  int[] tempArray = new int [SIZE];
  int index = leftFirst;
  int saveFirst = leftFirst;

  while ((leftFirst <= leftLast) && (rightFirst <= rightLast))
  {
    if (values[leftFirst] < values[rightFirst])
    {
      tempArray[index] = values[leftFirst];
```

```
      leftFirst++;
    }
    else
    {
      tempArray[index] = values[rightFirst];
      rightFirst++;
    }
    index++;
  }

  while (leftFirst <= leftLast)
  // Copy remaining items from left half

  {
    tempArray[index] = values[leftFirst];
    leftFirst++;
    index++;
  }

  while (rightFirst <= rightLast)
  // Copy remaining items from right half
  {
    tempArray[index] = values[rightFirst];
    rightFirst++;
    index++;
  }

  for (index = saveFirst; index <= rightLast; index++)
    values[index] = tempArray[index];
}
```

As we said, most of the work is in the merge task. The actual `mergeSort` method is short and simple:

```
static void mergeSort(int first, int last)
// Post: The elements in values are sorted by key
{
  if (first < last)
  {
    int middle = (first + last) / 2;
    mergeSort(first, middle);
    mergeSort(middle + 1, last);
    merge(first, middle, middle + 1, last);
  }
}
```

Analyzing mergeSort

The `mergeSort` method splits the original array into two halves. It first sorts the first half of the array, using the divide and conquer approach; then it sorts the second half of the array using the same approach; then it merges the two halves. To sort the first half of the array it follows the same approach, splitting and merging. During the sorting process the splitting and merging operations are all intermingled. However, analysis is simplified if we imagine that all of the splitting occurs first—we can view the process this way without affecting the correctness of the algorithm.

We view the `mergeSort` algorithm as continually dividing the original array (of size *N*) in two, until it has created *N* one element subarrays. Figure 10.11 shows this point of view for an array with an original size of 16. The total work needed to divide the array in half, over and over again until we reach subarrays of size 1, is O(*N*). After all, we end up with *N* subarrays of size 1.

Each subarray of size 1 is obviously a sorted subarray. The real work of the algorithm involves merging the smaller sorted subarrays back into the larger sorted subarrays. To merge two sorted subarrays of size *X* and size *Y* into a single sorted subarray using the `merge` operation requires O(*X* + *Y*) steps. We can see this because each time through the *while* loops of the `merge` method we either advance the `leftFirst` index or the `rightFirst` index by 1. Since we stop processing when these indexes become greater than their "last" counterparts, we know that we take a total of `(leftLast - leftFirst + 1) + (rightLast - rightFirst + 1)` steps. This expression represents the sum of the lengths of the two subarrays being processed.

How many times must we perform the `merge` operation? And what are the sizes of the subarrays involved? Let's work from the bottom up. The original array of size *N* is eventually split into *N* subarrays of size 1. Merging two of those subarrays, into a

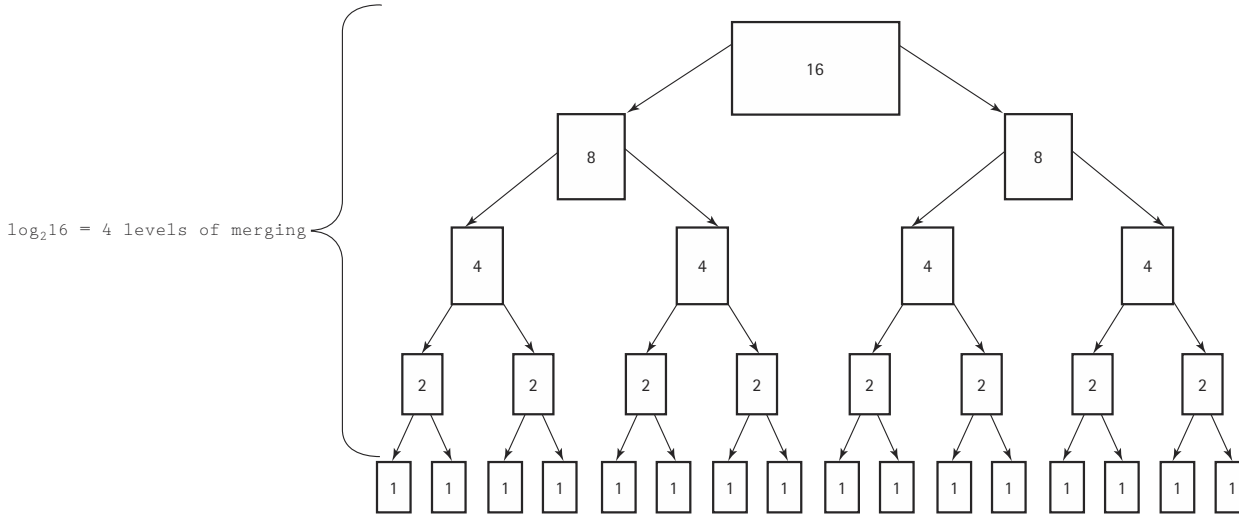


Figure 10.11 *Analysis of Merge Sort algorithm with N = 16*

subarray of size 2, requires O(1 + 1) = O(2) steps based on the analysis of the preceding paragraph. That is, it requires a small constant number of steps in each case. But, we must perform this merge operation a total of ½*N* times (we have *N* one-element subarrays and we are merging them two at a time). So the total number of steps to create all the sorted two-element subarrays is O(*N*). Now we repeat this process to create four-element subarrays. It takes four steps to merge two two-element subarrays. We must perform this merge operation a total of ¼*N* times (we have ½*N* two-element subarrays and we are merging them two at a time). So the total number of steps to create all the sorted four-element subarrays, is also O(*N*) (4 * ¼*N* = *N*). The same reasoning leads us to conclude that each of the other levels of merging also requires O(*N*) steps—at each level the sizes of the subarrays double, but the number of subarrays is cut in half, balancing out.

We now know that it takes O(*N*) total steps to perform merging at each "level" of merging. How many levels are there? The number of levels of merging is equal to the number of times we can split the original array in half. If the original array is size *N*, we have log$_2$*N* levels. (This is just like the analysis of the binary search algorithm.) For example, in Figure 10.11 the size of the original array is 16 and the number of levels of merging is 4. Since we have log$_2$*N* levels, and we require O(*N*) steps at each level, the total cost of the merge operation is: O(*N*log$_2$*N*). And since the splitting phase was only O(*N*), we conclude that Merge Sort algorithm is O(*N*log$_2$*N*). Table 10.2 illustrates that, for large values of *N*, O(*N*log$_2$*N*) is a big improvement over O(N^2).

The disadvantage of `mergeSort` is that it requires an auxiliary array that is as large as the original array to be sorted. If the array is large and space is a critical factor, this sort may not be an appropriate choice. Next we discuss two sorts that move elements around in the original array and do not need an auxiliary array.

Table 10.2 *Comparing N^2 and N log$_2$ N*

N	log$_2$N	N^2	Nlog$_2$N
32	5	1,024	160
64	6	4.096	384
128	7	16,384	896
256	8	65,536	2,048
512	9	262,144	4,608
1024	10	1,048,576	10,240
2048	11	4,194,304	22,528
4096	12	16,777,216	49,152

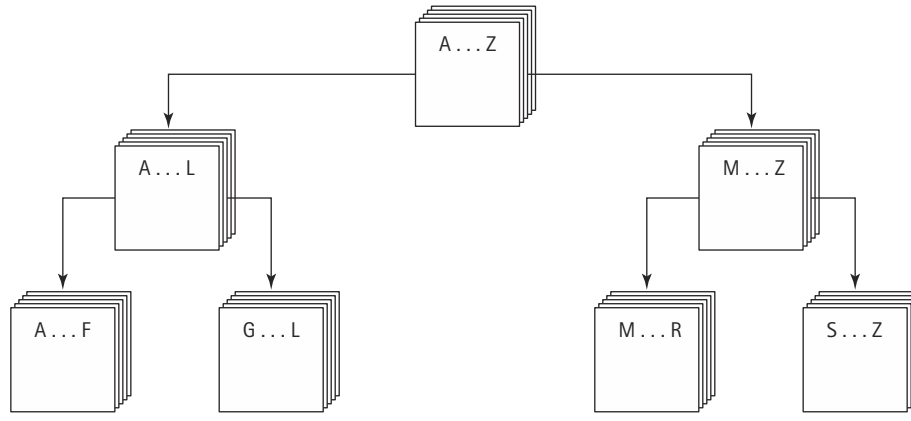


Figure 10.12 *Ordering a list using the Quick Sort algorithm*

Quick Sort

Like Merge Sort, Quick Sort is a divide-and-conquer algorithm, which is inherently recursive. If you were given a large stack of final exams to sort by name, you might use the following approach: pick a splitting value, say, L and divide the stack of tests into two piles, A–L and M–Z. (Note that the two piles do not necessarily contain the same number of tests.) Then take the first pile and subdivide it into two piles, A–F and G–L. The A–F pile can be further broken down into A–C and D–F. This division process goes on until the piles are small enough to be easily sorted. The same process is applied to the M–Z pile.

Eventually all the small sorted piles can be collected one on top of the other to produce a sorted set of tests. (See Figure 10.12.)

This strategy is recursive—on each attempt to sort the pile of tests, the pile is divided, and then the same approach is used to sort each of the smaller piles (a smaller case). This process goes on until the small piles do not need to be further divided (the base case). The parameter list of the `quickSort` method reflects the part of the list that is currently being processed; we pass the first and last indexes that define the part of the array to be processed on this call. The initial call to `quickSort` is

```
quickSort(0, SIZE - 1);
```

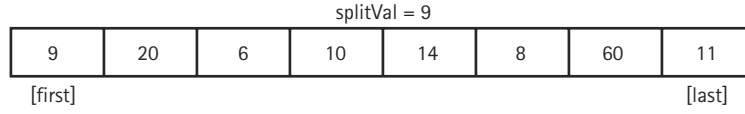
Method quickSort (first, last)

Definition:	Sorts the items in sub array values[first]..values[last].
Size:	last − first + 1
Base Case:	If size less than 2 , do nothing.
General Case:	Split the array according to splitting value. quickSort the elements <= splitting value. quickSort the elements > splitting value.

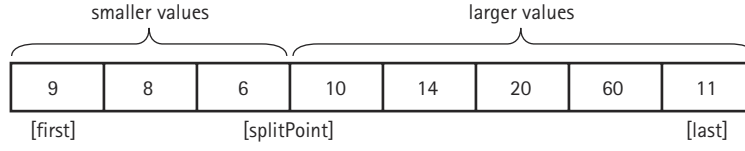
quickSort

```
if there is more than one element in values[first]..values[last]
    Select splitVal
    Split the array so that
        values[first]..values[splitPoint – 1] <= splitVal
        values[splitPoint] = splitVal
        values[splitPoint + 1]..values[last] > splitVal
    quickSort the left half
    quickSort the right half
```

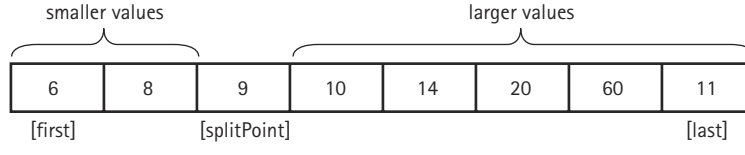
How do we select `splitVal`? One simple solution is to use the value in `values[first]` as the splitting value. (We show a better solution later.)



We create a helper method `split`, to rearrange the array elements as planned. After the call to `split`, all the items less than or equal to `splitVal` are on the left side of the array and all of those greater than `splitVal` are on the right side of the array.



The two "halves" meet at `splitPoint`, the index of the last item that is less than or equal to `splitVal`. Note that we don't know the value of `splitPoint` until the splitting process is complete. Its value is returned by `split`. We can then swap `splitVal` with the value at `splitPoint`.



Our recursive calls to `quickSort` use this index (`splitPoint`) to reduce the size of the problem in the general case.

`quickSort(first, splitPoint - 1)` sorts the left "half" of the array. `quickSort(splitPoint + 1, last)` sorts the right "half" of the array. (The "halves" are not necessarily the same size.) `splitVal` is already in its correct position in `values[splitPoint]`.

What is the base case? When the segment being examined has less than two items, we do not need to go on. So "there is more than one item in `values[first]..values[last]`" can be translated into "`if (first < last)`". We can now code our `quickSort` method.

```
static void quickSort(int first, int last)
{
  if (first < last)
  {
    int splitPoint;

    splitPoint = split(first, last);
    // values[first]..values[splitPoint - 1] <= splitVal
    // values[splitPoint] = splitVal
    // values[splitPoint+1]..values[last] > splitVal

    quickSort(first, splitPoint - 1);
    quickSort(splitPoint + 1, last);
  }
}
```

Now we must develop our splitting algorithm. We must find a way to get all of the elements equal to or less than `splitVal` on one side of `splitVal` and the elements greater than `splitVal` on the other side.

We do this by moving the indexes, `first` and `last`, toward the middle of the array, looking for items that are on the wrong side of the split point and swapping them (Figure 10.13). While this is proceeding, the `splitVal` remains in the `first` position of the subarray being processed. As a final step, we swap it with the value at the `splitPoint`; therefore, we save the original value of `first` in a local variable, `saveF`. (See Figure 10.13a.)

We start out by moving `first` to the right, toward the middle, comparing `values[first]` to `splitVal`. If `values[first]` is less than or equal to `splitVal`, we keep incrementing `first`; otherwise we leave `first` where it is and begin moving `last` toward the middle. (See Figure 10.13b.)

Now `values[last]` is compared to `splitVal`. If it is greater, we continue decrementing `last`; otherwise, we leave `last` in place. (See Figure 10.13c.) At this point, it is clear that `values[last]` and `values[first]` are each on the wrong side of the array. Note that the elements to the left of `values[first]` and to the right of `values[last]` are not necessarily sorted; they are just on the correct side of the array with respect to

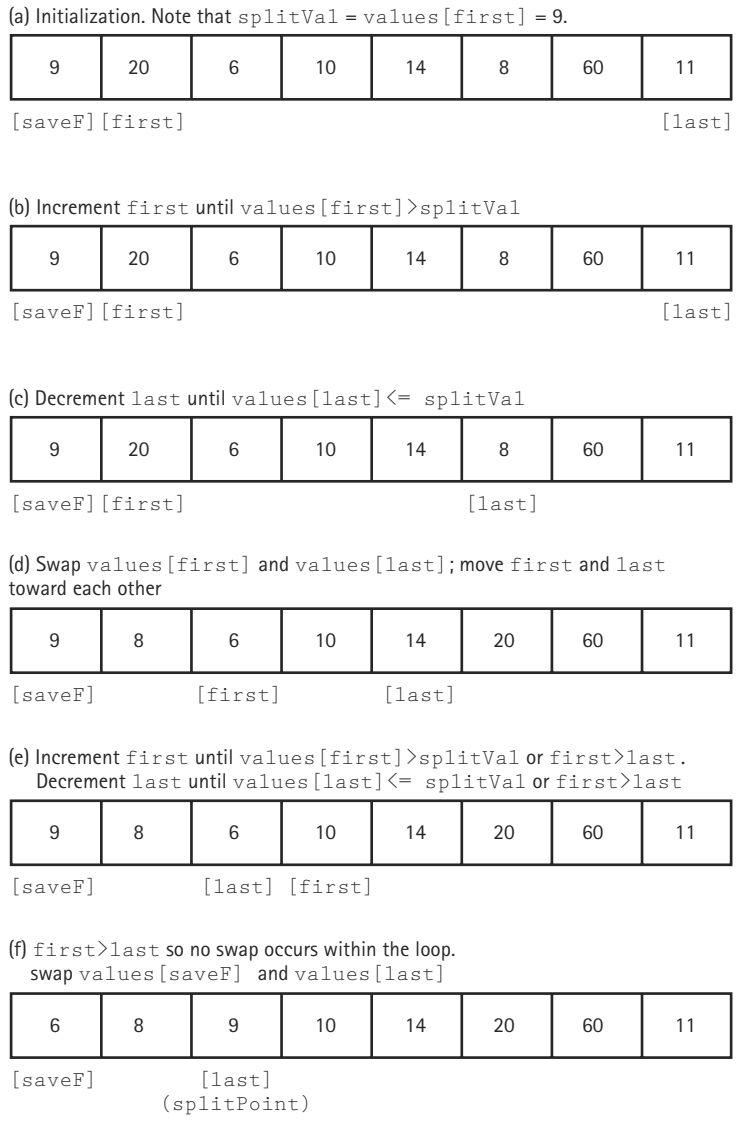


Figure 10.13 *The* `split` *operation*

`splitVal`. To put `values[first]` and `values[last]` into their correct sides, we merely swap them, then increment `first` and decrement `last`. (See Figure 10.13d.)

Now we repeat the whole cycle, incrementing `first` until we encounter a value that is greater than `splitVal`, then decrementing `last` until we encounter a value that is less than or equal to `splitVal`. (See Figure 10.13e.)

When does the process stop? When `first` and `last` meet each other, no further swaps are necessary. Where they meet determines the `splitPoint`. This is the location where `splitVal` belongs, so we swap `values[saveF]`, which contains `splitVal`, with the element at `values[splitPoint]` (Figure 10.13f). The index `splitPoint` is returned from the method, to be used by `quickSort` to set up the next pair of recursive calls.

```
static int split(int first, int last)
{
  int splitVal = values[first];
  int saveF = first;
  boolean onCorrectSide;

  first++;
  do
  {
    onCorrectSide = true;
    while (onCorrectSide)              // Move first toward last
      if (values[first] > splitVal)
        onCorrectSide = false;
      else
      {
        first++;
        onCorrectSide = (first <= last);
      }

    onCorrectSide = (first <= last);
    while (onCorrectSide)              // Move last toward first
      if (values[last] <= splitVal)
        onCorrectSide = false;
      else
       {
        last--;
        onCorrectSide = (first <= last);
       }

    if (first < last)
    {
      swap(first, last);
      first++;
      last--;
    }
  } while (first <= last);

  swap(saveF, last);
  return last;
}
```

What happens if our splitting value is the largest or the smallest value in the segment? The algorithm still works correctly, but because the split is lopsided, it is not so quick.

Is this situation likely to occur? That depends on how we choose our splitting value and on the original order of the data in the array. If we use `values[first]` as the splitting value and the array is already sorted, then *every* split is lopsided. One side contains one element, while the other side contains all but one of the elements. Thus, our `quickSort` is not a quick sort. Our splitting algorithm works best for an array in random order.

It is not unusual, however, to want to sort an array that is already in nearly sorted order. If this is the case, a better splitting value would be the middle value,

```
values[(first + last) / 2]
```

This value could be swapped with `values[first]` at the beginning of the method.

Analyzing quickSort

The analysis of `quickSort` is very similar to that of `mergeSort`. On the first call, every element in the array is compared to the dividing value (the "split value"), so the work done is O(N). The array is divided into two parts (not necessarily halves), which are then examined.

Each of these pieces is then divided in two, and so on. If each piece is split approximately in half, there are O($\log_2 N$) levels of splits. At each level, we make O(N) comparisons. So Quick Sort is also an O($N \log_2 N$) algorithm, which is quicker than the O(N^2) sorts we discussed at the beginning of this chapter.

But Quick Sort isn't always quicker. Note that there are $\log_2 N$ levels of splits if each split divides the segment of the array approximately in half. As we've seen, the array division of Quick Sort is sensitive to the order of the data, that is, to the choice of the splitting value.

What happens if the array is already sorted when our version of `quickSort` is called? The splits are very lopsided, and the subsequent recursive calls to `quickSort` break into a segment of one element and a segment containing all the rest of the array. This situation produces a sort that is not at all quick. In fact, there are $N - 1$ levels; in this case Quick Sort is O(N^2).

Such a situation is very unlikely to occur by chance. By way of analogy, consider the odds of shuffling a deck of cards and coming up with a sorted deck. On the other hand, in some applications you may know that the original array is likely to be sorted or nearly sorted. In such cases you would want to use either a different splitting algorithm or a different sort—maybe even `shortBubble`!

What about space requirements? Quick Sort does not require an extra array, as Merge Sort does. Are there any extra space requirements, besides the few local variables? Yes-remember that Quick Sort uses a recursive approach. There can be many levels of recursion "saved" on the system stack at any time. On average, the algorithm requires O($\log_2 N$) extra space to hold this information and in the worst case requires O(N) extra space, the same as Merge Sort.

Heap Sort

In each iteration of the selection sort, we searched the array for the next-smallest element and put it into its correct place in the array. Another way to write a selection sort is to find the maximum value in the array and swap it with the last array element, then find the next-to-largest element and put it into its place, and so on. Most of the work in this sorting algorithm comes from searching the remaining part of the array in each iteration, looking for the maximum value.

In Chapter 9, we discussed the *heap*, a data structure with a very special feature: We always know where to find its greatest element. Because of the order property of heaps, the maximum value of a heap is in the root node. We can take advantage of this situation by using a heap to help us sort. The general approach of the heap sort is as follows:

1. Take the root (maximum) element off the heap, and put it into its place.
2. Reheap the remaining elements. (This puts the next-largest element into the root position.)
3. Repeat until there are no more elements.

The first part of this algorithm sounds a lot like the straight selection sort. What makes the heap sort fast is the second step: finding the next-largest element. Because the shape property of heaps guarantees a binary tree of minimum height, we make only $O(\log_2 N)$ comparisons in each iteration, as compared with $O(N)$ comparisons in each iteration of the selection sort.

Building a Heap

By now you are probably protesting that we are dealing with an unsorted array of elements, not a heap. Where does the original heap come from? Before we go on, we have to convert the unsorted array, `values`, into a heap.

Let's take a look at how the heap relates to our array of unsorted elements. In Chapter 9 we saw how heaps can be represented in an array with implicit links. Because of the shape property, we know that the heap elements take up consecutive positions in the array. In fact, the unsorted array of data elements already satisfies the shape property of heaps. Figure 10.14 shows an unsorted array and its equivalent tree.

We also need to make the unsorted array elements satisfy the order property of heaps. First let's see if there's any part of the tree that already satisfies the order property. All of the leaf nodes (subtrees with only a single node) are heaps. In Figure 10.15(a) the subtrees whose roots contain the values 19, 7, 3, 100, and 1 are heaps because they consist solely of root nodes.

Now let's look at the first *nonleaf* node, the one containing the value 2 (Figure 10.15b). The subtree rooted at this node is not a heap, but it is *almost* a heap—all of the nodes *except the root node* of this subtree satisfy the order property. We know how to fix this problem. In Chapter 9 we developed a heap utility method, `reheapDown`, that can be used to correct this exact situation. Given a tree whose elements satisfy the order property of heaps except that the tree has an empty root, and a value to insert into the heap, `reheapDown` rearranges the

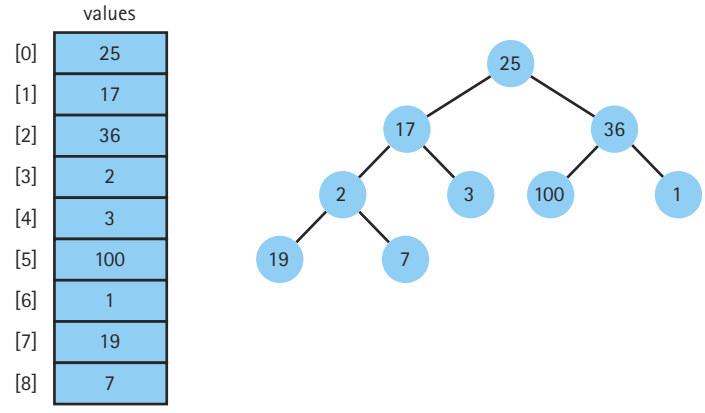


Figure 10.14 *An unsorted array and its tree*

nodes, leaving the (sub)tree containing the new element as a heap. We can invoke `reheapDown` on the subtree, passing it the current root value of the subtree.

We apply this method to all the subtrees on this level, and then we move up a level in the tree and continue reheaping until we reach the root node. After `reheapDown` has been called for the root node, the whole tree should satisfy the order property of heaps. This heap-building process is illustrated in Figure 10.15; the changing contents of the array are shown in Figure 10.16.

In Chapter 9, we defined `reheapDown` as a private method of the `Heap` class. There, the method had only one parameter, the item being inserted into the heap. It always worked on the entire tree, that is, it always started with an empty node at index 0 and assumed that the last tree index of the heap was `lastIndex`. Here, we use a slight variation: `reheapDown` is a static method of our `Sorts` class that takes a second parameter—the index of the node that is the root of the subtree that is to be made into a heap. This is an easy change; if we call the parameter `root` we simply add the following statement to the beginning of the `reheapDown` method:

```
int hole = root;    // Current index of hole
```

The algorithm for building a heap is summarized here:

buildHeap

for index going from first nonleaf node up to the root node
    reheapDown(values[index], index)

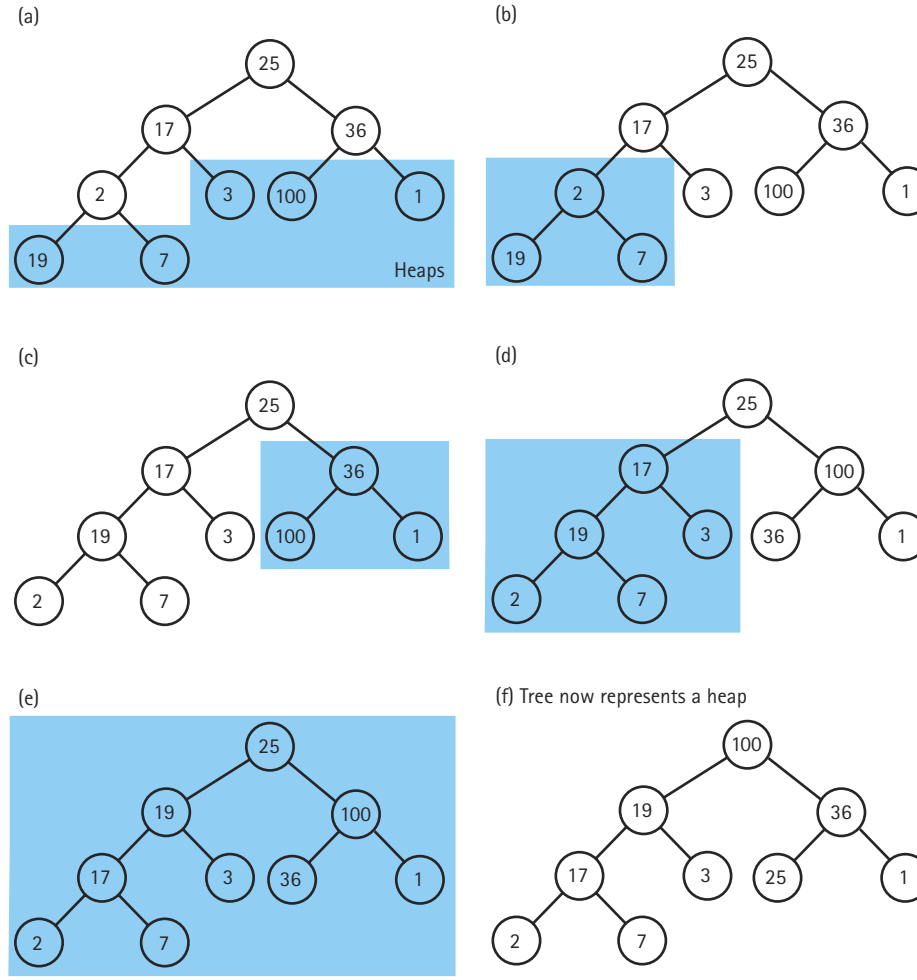


Figure 10.15 *The heap-building process*

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
Original values	25	17	36	2	3	100	1	19	7
After reheapDown index = 3	25	17	36	19	3	100	1	2	7
After index = 2	25	17	100	19	3	36	1	2	7
After index = 1	25	19	100	17	3	36	1	2	7
After index = 0	100	19	36	17	3	25	1	2	7
Tree is a heap.									

Figure 10.16 *Changing contents of the array*

We know where the root node is stored in our array representation of heaps—it's in `values[0]`. Where is the first nonleaf node? Because half the nodes of a complete binary tree are leaves (prove this yourself), the first nonleaf node may be found at position `SIZE/2 - 1`.

Sorting Using the Heap

Now that we are satisfied that we can turn the unsorted array of elements into a heap, let's take another look at the sorting algorithm.

We can easily access the largest element from the original heap—it's in the root node. In our array representation of heaps, that is `values[0]`. This value belongs in the last-used array position `values[SIZE - 1]`, so we can just swap the values in these two positions. Because `values[SIZE - 1]` now contains the largest value in the array (its correct sorted value), we want to leave this position alone. Now we are dealing with a set of elements, from `values[0]` through `values[SIZE - 2]`, that is almost a heap. We know that all of these elements satisfy the order property of heaps, except (perhaps) the root node. To correct this condition, we call our heap utility, `reheapDown`. (But, our original `reheapDown` method assumed that the heap's tree ends at position `lastIndex`. We must again redefine `reheapDown`, so that it now accepts three parameters, the third being the ending index of the heap. And again the change is easy; the new code for `reheapDown` is included in the `Sorts` class file on our web site.)

At this point we know that the next-largest element in the array is in the root node of the heap. To put this element in its correct position, we swap it with the element in `values[SIZE - 2]`. Now the two largest elements are in their final correct positions, and the elements in `values[0]` through `values[SIZE - 3]` are almost a heap. So we call `reheapDown` again, and now the third-largest element is in the root of the heap.

This process is repeated until all of the elements are in their correct positions; that is, until the heap contains only a single element, which must be the smallest item in the array, in `values[0]`. This is its correct position, so the array is now completely sorted from the smallest to the largest element. Notice that at each iteration the size of the unsorted portion (represented as a heap) gets smaller and the size of the sorted portion gets larger. At the end of the algorithm, the size of the sorted portion is the size of the original array.

The heap sort algorithm, as we have described it, sounds like a recursive process. Each time we swap and reheap a smaller portion of the total array. Because it uses tail recursion, we can code the repetition just as clearly using a simple for loop. The node-sorting algorithm is as follows:

Sort Nodes

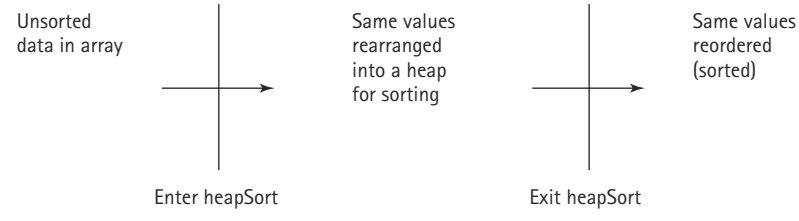
```
for index going from last node up to next-to-root node
    Swap data in root node with values[index]
    reheapDown(values[0], 0, index — 1)
```

Method `heapSort` first builds the heap and then sorts the nodes, using the algorithms just discussed.

```
static void heapSort()
// Post: The elements in the array values are sorted by key
{
  int index;
  // Convert the array of values into a heap
  for (index = SIZE/2 - 1; index >= 0; index--)
    reheapDown(values[index], index, SIZE - 1);

  // Sort the array
  for (index = SIZE - 1; index >=1; index--)
  {
    swap(0, index);
    reheapDown(values[0], 0, index - 1);
  }
}
```

Figure 10.17 shows how each iteration of the sorting loop (the second for loop) would change the heap created in Figure 10.16. Each line represents the array after one operation. The sorted elements are shaded.

We entered the `heapSort` method with a simple array of unsorted values and returned with an array of the same values sorted in ascending order. Where did the heap go? The heap in `heapSort` is just a temporary structure, internal to the sorting algorithm. It is created at the beginning of the method to aid in the sorting process, and then is methodically diminished element by element as the sorted part of the array grows. At the end of the method, the sorted part fills the array and the heap has completely disappeared. When we used heaps to implement priority queues in Chapter 9, the heap structure stayed around for the duration of the use of the queue. The heap in `heapSort`, in contrast, is not a retained data structure. It only exists temporarily, during the execution of the `heapSort` method.



Analyzing Heap Sort

The code for method `heapSort` is very short—only a few lines of new code plus the helper method `reheapDown`, which we developed in Chapter 9 (albeit, slightly revised).

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
values	100	19	36	17	3	25	1	2	7
swap	7	19	36	17	3	25	1	2	100
reheapDown	36	19	25	17	3	7	1	2	100
swap	2	19	25	17	3	7	1	36	100
reheapDown	25	19	7	17	3	2	1	36	100
swap	1	19	7	17	3	2	25	36	100
reheapDown	19	17	7	1	3	2	25	36	100
swap	2	17	7	1	3	19	25	36	100
reheapDown	17	3	7	1	2	19	25	36	100
swap	2	3	7	1	17	19	25	36	100
reheapDown	7	3	2	1	17	19	25	36	100
swap	1	3	2	7	17	19	25	36	100
reheapDown	3	1	2	7	17	19	25	36	100
swap	2	1	3	7	17	19	25	36	100
reheapDown	2	1	3	7	17	19	25	36	100
swap	1	2	3	7	17	19	25	36	100
reheapDown	1	2	3	7	17	19	25	36	100
Exit from sorting loop	1	2	3	7	17	19	25	36	100

Figure 10.17 *Effect of* `heapSort` *on the array*

These few lines of code, however, do quite a bit. All of the elements in the original array are rearranged to satisfy the order property of heaps, moving the largest element up to the top of the array, only to put it immediately into its place at the bottom. It's hard to believe from a small example such as the one in Figure 10.17 that `heapSort` is very efficient.

In fact, for small arrays, `heapSort` is not very efficient because of all the "overhead." For large arrays, however, `heapSort` is very efficient. Let's consider the sorting loop. We loop through $N - 1$ times, swapping elements and reheaping. The comparisons occur in `reheapDown` (actually in its helper method `newHole`). A complete binary tree with N nodes has $O(\log_2(N + 1))$ levels. In the worst cases, then, if the root element had to be bumped down to a leaf position, the `reheapDown` method would make $O(\log_2 N)$ comparisons. So method `reheapDown` is $O(\log_2 N)$. Multiplying this activity by the $N - 1$ iterations shows that the sorting loop is $O(N\log_2 N)$.

Combining the original heap build, which is $O(N)$, and the sorting loop, we can see that Heap Sort requires $O(N\log_2 N)$ comparisons. Note that, unlike Quick Sort, Heap Sort's efficiency is not affected by the initial order of the elements. Heap Sort is just as efficient in terms of space; only one array is used to store the data. Heap Sort only requires constant extra space.

10.4 More Sorting Considerations

In this section we wrap up our coverage of sorting by revisiting efficiency, considering the "stability" of sorting algorithms, and discussing special concerns involved with sorting objects rather than primitive types.

Testing

All of our sorts were implemented within the test harness presented in Section 10.1. That test harness program, `Sorts`, allows us to generate a random array of size 50, sort it with one of our algorithms, and view the sorted array. It is easy to see if the sort was successful. If we do not want to verify success by eyeballing the output, we can always use a call to the `isSorted` method of the `Sorts` class.

`Sorts` is a useful tool for quickly evaluating the correctness of our sorting methods. However, to thoroughly test them, we should vary the size of the array they are sorting. A small revision to `Sorts`, allowing the user to pass the array size as a command-line parameter, would facilitate this process. We should also vary the original order of the array—test an array that is in reverse order, one that is almost sorted, and one that has all identical elements (to make sure we do not generate an array index out of bounds error).

Besides validating that our sort methods create a sorted array, we can check on their performance. At the start of the sorting phase we can initialize two variables, `numSwaps` and `numCompares`, to 0. By carefully placing statements incrementing these variables throughout our code, we can use them to track how many times the code performs swaps and comparisons. Now we output these values and compare them to the predicted theoretical values. Inconsistencies would require further review of the code (or maybe the theory!).

Efficiency

When N Is Small

As we have stressed throughout this chapter, we have based our analysis of efficiency on the number of comparisons made by a sorting algorithm. This number gives us a rough estimate of the computation time involved. The other activities that accompany the comparison (swapping, keeping track of `boolean` flags, and so forth) contribute to the "constant of proportionality" of the algorithm.

In comparing Big-O evaluations, we ignored constants and smaller-order terms, because we wanted to know how the algorithm would perform for large values of N. In general, a $O(N^2)$ sort requires few extra activities in addition to the comparisons, so its constant of proportionality is fairly small. On the other hand, a $O(N\log_2 N)$ sort may be more complex, with more overhead and thus a larger constant of proportionality. This situation may cause anomalies in the relative performances of the algorithms when the value of N is small. In this case, N^2 is not much greater than $N\log_2 N$, and the constants may dominate instead, causing a $O(N^2)$ sort to run faster than a $O(N\log_2 N)$ sort.

We have discussed sorting algorithms that have complexity either $O(N^2)$ or $(N\log_2 N)$. An obvious question is: Are there algorithms that are better than $(N\log_2 N)$? No, it has been proven theoretically that we cannot do better than $(N\log_2 N)$ for sorting algorithms that are based on comparing keys.

Eliminating Calls to Methods

Sometimes it may be desirable, for efficiency considerations, to streamline the code as much as possible, even at the expense of readability. For instance, we have consistently used

```
swap(index1, index2);
```

when we wanted to swap two items in the `values` array. We would achieve slightly better execution efficiency by dropping the method call and directly coding:

```
tempValue = values[index1];
values[index1] = values[index2];
values[index2] = tempValue;
```

Coding the Swap operation as a method made the code simpler to write and to understand, avoiding a cluttered sort method. But, method calls require extra overhead that you may prefer to avoid during a sort, where the method is called over and over again within a loop.

The recursive sorting methods, `mergeSort` and `quickSort`, have a similar situation: They require the extra overhead involved in executing the recursive calls. You may want to avoid this overhead by coding nonrecursive versions of these methods.

In some cases, an optimizing compiler replaces method calls with the inline expansion of the code of the method. In that case, we get the benefits of both readability and efficiency.

Programmer Time

If the recursive calls are less efficient, why would anyone ever decide to use a recursive version of a sort? The decision involves a choice between types of efficiency. Up until now, we have only been concerned with minimizing computer time. While computers are becoming faster and cheaper, however, it is not at all clear that computer programmers are following that trend. In fact, programmers are becoming more expensive. Therefore, in some situations, programmer time may be an important consideration in choosing an algorithm and its implementation. In this respect, the recursive version of Quick Sort is more desirable than its nonrecursive counterpart, which requires the programmer to simulate the recursion explicitly.

Of course, if a programmer is familiar with a language's support library, the programmer can use the sort routines provided there. The Java `Arrays` class in the library's `util` package defines a number of sorts for sorting arrays. Likewise, the Java collections framework, introduced in Section 4.3, provides methods for sorting many of its collection objects.

Space Considerations

Another efficiency consideration is the amount of memory space required. In small applications, memory space is not a very important factor in choosing a sorting algorithm. In large applications, such as a database with many gigabytes of data, space may be a serious concern. We looked at only two sorts, `mergeSort` and `quickSort`, which required more than constant extra space. The usual time-versus-space tradeoff applies to sorts—more space often means less time, and vice versa.

Because processing time is the factor that applies most often to sorting algorithms, we have considered it in detail here. Of course, as in any application, the programmer must determine goals and requirements before selecting an algorithm and starting to code.

Sorting Objects

So that we could concentrate on the algorithms, we limited our implementations to sorting arrays of integers. Do the same approaches work for sorting objects? Of course. But there are a few extra considerations.

References

Keep in mind that when sorting an array of objects we are manipulating references to the objects, and not the objects themselves. See Figure 10.18. This does not affect any of our algorithms, but it is still important to understand. For example, if we decide to swap the objects at index 0 and index 1 of an array, it is actually the references to the objects that we swap and not the objects themselves. At an abstract level, we view objects, and the references to the objects, as identical.

Comparisons

When sorting objects we must have a way to compare two objects and decide which is "larger." There are two basic approaches used when dealing with Java objects.

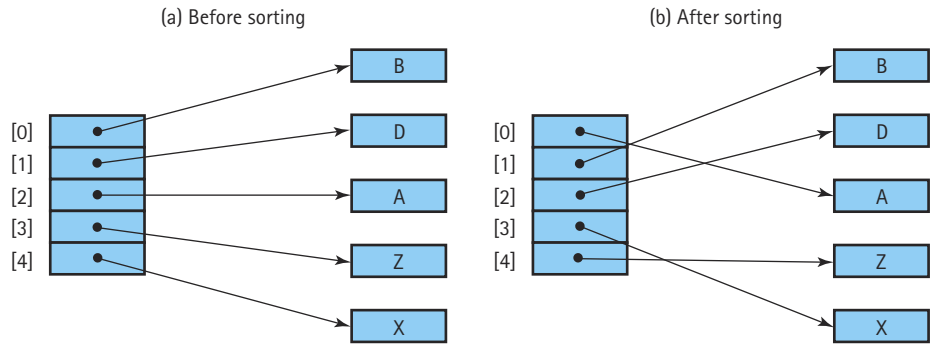


Figure 10.18 *Sorting arrays with references*

Using the Comparable Interface The first approach you are familiar with from previous chapters. If the object class exports a `compareTo` operation, or something similar, it can be used to provide the needed comparison. This is the approach we have used throughout the text. For example, our sorted lists contained objects that implemented the `Listable` interface, which in turn requires a `compareTo` method. For our Binary Search Trees and Priority Queues we used objects that implemented Java's `Comparable` interface. In fact, the only requirement for a `Comparable` object is that it provides a `compareTo` operation. It is common for Java programmers, when creating methods that need to compare objects, to insist that all arguments be of type `Comparable`.

For example, here is what the `bubbleUp` implementation looks like when defined to sort objects of type `Comparable`, rather than just integers:

```
static void bubbleUp(int startIndex, int endIndex)
// Post: Adjacent pairs that are out of order have been switched
//       between values[startIndex]..values[endIndex] beginning at
//       values[endIndex]
{
  for (int index = endIndex; index > startIndex; index--)
    if (values[index].compareTo(values[index - 1]) < 0)
      swap(index, index - 1);
}
```

A limitation of this approach is that a class can only have one `compareTo` method. What if we have a class of objects, for example student records, that we want to sort in many various ways: by name, by grade, by zip code, in increasing order, in decreasing order? In this case we need to use the next approach.

Using the Comparator Interface The second common approach allows more flexibility. The Java Library provides another interface related to comparing objects called `Comparator`. The interface defines two abstract methods:

```
public abstract int compare(Object o1, Object o2);
// Returns a negative integer, zero, or a positive integer to
//  indicate that o1 is less than, equal to, or greater than o2

public abstract boolean equals(Object obj);
// Returns true if this Object equals obj; false, otherwise
```

The first method, `compare`, is very similar to the familiar `compareTo` method. However, it takes two arguments, rather than one. The second method, `equals`, is specified the same as the `equals` method of the `Object` class. We do not have to provide a concrete method for `equals` when creating a `Comparator` class, since every class inherits

the `equals` method from `Object`. We do not address the `equals` method again in this discussion.

Any sort implementation must compare elements. Our methods so far have used built-in integer comparison operations such as "<" or "<=". If we sort `Comparable` objects, instead of integers, we could use the `compareTo` method that is guaranteed to exist by that interface. Alternately, we could use a versatile approach supported by the `Comparator` interface. If we pass a `Comparator` object `comp` to a sorting method as a parameter, the method can use `comp.compare` to determine the relative order of two objects and base its sort on that relative order. Passing a different `Comparator` object results in a different sorting order. Perhaps one `Comparator` object defines an increasing order, and another defines a decreasing order. Or, the different `Comparator` objects could define order based on different attributes of the objects. Now, with a single sorting method, we can produce many different sort orders.

Let's look at an example. As we did in Chapter 9, to allow us to concentrate on the topic of discussion, we use a simple circle class. The fact that the fields of circles are not private makes it easy to demonstrate the concepts of this section. We redefine circles slightly here as follows:

```
package ch10.circles;

public class SortCircle
{
  public int xValue;
  public int yValue;
  public int radius;
  public boolean solid;
}
```

Here is the definition of a `Comparator` object that orders `SortCircles` based on their `xValue`:

```
Comparator xComp = new Comparator()
{
  public int compare(Object a, Object b)
  {
    SortCircle circleA = (SortCircle)a;
    SortCircle circleB = (SortCircle)b;
    return (circleA.xValue - circleB.xValue);
  }
};
```

And here is a `selectionSort` method, along with its helper method `minIndex`, that accepts and uses a `Comparator` object (the changes from the previous version of `selectionSort` are emphasized):

```
static int minIndex(int startIndex, int endIndex, Comparator comp)
// Post: Returns the index of the smallest value in
//       values[startIndex]..values[endIndex]
//       based on the Comparator comp
{
  int indexOfMin = startIndex;
  for (int index = startIndex + 1; index <= endIndex; index++)
    if (comp.compare(values[index],values[indexOfMin]) < 0)
      indexOfMin = index;
  return indexOfMin;
}

static void selectionSort(Comparator comp)
// Post: The elements in the array values are sorted
{
  int endIndex = SIZE - 1;
  for (int current = 0; current < endIndex; current++)
    swap(current, minIndex(current, endIndex, comp));
}
```

Note how passing a different `Comparator` object to `selectionSort` would result in a different sort order. This makes the sort operation extremely versatile. Just by passing it different `Comparator` objects it can sort circles in increasing or decreasing order based on any of the circle fields, or even any mathematical combination of circle fields. The following program, `Sorts2`, demonstrates our new flexibility. It generates an array of six random `SortCircle` objects, prints them, sorts them by `xValue`, prints them, sorts them by `yValue`, and then prints them again. Study the program carefully. The output from an execution of the program is:

```
the values array is:

 x  y  r solid
-- -- -- -----
37 83 82 true
46 25 71 false
43 73 62 true
08 67 40 false
69 68 70 true
20 95 15 false
```

```
the values array is:

 x  y  r solid
-- -- -- ---
08 67 40 false
20 95 15 false
37 83 82 true
43 73 62 true
46 25 71 false
69 68 70 true

the values array is:

 x  y  r solid
-- -- -- ---
46 25 71 false
08 67 40 false
69 68 70 true
43 73 62 true
37 83 82 true
20 95 15 false
```

```
//-------------------------------------------------------------------------
// Sorts2.java              by Dale/Joyce/Weems                    Chapter 10
//
// Test harness used to run sorting algorithms that use Comparator
//-------------------------------------------------------------------------

import java.io.*;
import java.util.*;
import java.text.DecimalFormat;
import ch10.circles.*;

public class Sorts2
{
  static final int SIZE = 6;                        // Size of array to be sorted
  static SortCircle[] values = new SortCircle[SIZE];   // Values to be sorted

  static void initValues()
  // Initializes the values array with random circles
  {
```

```
    Random rand = new Random();
    for (int index = 0; index < SIZE; index++)
    {
      values[index] = new SortCircle();
      values[index].xValue = Math.abs(rand.nextInt()) % 100;
      values[index].yValue = Math.abs(rand.nextInt()) % 100;
      values[index].radius = Math.abs(rand.nextInt()) % 100;
      values[index].solid = ((Math.abs(rand.nextInt()) % 2) == 0);
    }
  }

  static public void swap(int index1, int index2)
  // Swaps the SortCircles at locations index1 and index2 of array values
  // Precondition: index1 and index2 are less than SIZE
  {
    SortCircle temp = values[index1];
    values[index1] = values[index2];
    values[index2] = temp;
  }

  static public void printValues()
  // Prints all the values integers
  {
    SortCircle value;
    DecimalFormat fmt = new DecimalFormat("00");
    System.out.println("the values array is:");
    System.out.println();
    System.out.println(" x  y  r solid");
    System.out.println("-- -- -- ----");
    for (int index = 0; index < SIZE; index++)
    {
      value = values[index];
      System.out.print(fmt.format(value.xValue) + " ");
      System.out.print(fmt.format(value.yValue) + " ");
      System.out.print(fmt.format(value.radius) + " ");
      System.out.print(value.solid);
      System.out.println();
    }
    System.out.println();
  }

  static int minIndex(int startIndex, int endIndex, Comparator comp)
  // Post: Returns the index of the smallest value in
  //       values[startIndex]..values[endIndex]
  //       based on the Comparator comp
```

```
  {
    int indexOfMin = startIndex;
    for (int index = startIndex + 1; index <= endIndex; index++)
      if (comp.compare(values[index],values[indexOfMin]) < 0)
        indexOfMin = index;
    return indexOfMin;
  }

  static void selectionSort(Comparator comp)
  // Post: The elements in the array values are sorted
  {
    int endIndex = SIZE - 1;
    for (int current = 0; current < endIndex; current++)
      swap(current, minIndex(current, endIndex, comp));
  }

  public static void main(String[] args) throws IOException
  {
    Comparator xComp = new Comparator()
    {
      public int compare(Object a, Object b)
      {
        SortCircle circleA = (SortCircle)a;
        SortCircle circleB = (SortCircle)b;
        return (circleA.xValue - circleB.xValue);
      }
    };

    Comparator yComp = new Comparator()
    {
      public int compare(Object a, Object b)
      {
        SortCircle circleA = (SortCircle)a;
        SortCircle circleB = (SortCircle)b;
        return (circleA.yValue - circleB.yValue);
      }
    };

    initValues();
    printValues();
    selectionSort(xComp);
    printValues();
    selectionSort(yComp);
    printValues();
  }
}
```

Remember that using the `Comparator` approach does require revising our sorting routines slightly; they must accept a `Comparator` as a parameter and use it appropriately. With similar changes, we could use this approach for any of our ADTs that involve comparing elements: our lists, binary search trees, and priority queues. If our goal is to make our ADTs as generally usable as possible, we should certainly consider this new approach.

The added flexibility of `Comparator` comes with a cost in performance. Just as we said that writing the swap operation within a separate method adds overhead to execution time, performing the comparison within a method also takes more time than a direct comparison. The `Comparable` interface also places the `compare` operation within a method, but most optimizing compilers are able to automatically extract the code from the method and place it directly in the sort to avoid this cost. With `Comparator`, however, different methods are used at different times, and so the same optimization can't be applied.

Stability

The stability of a sorting algorithm is based on what it does with duplicate values. Of course, the duplicate values all appear consecutively in the final order. For example, if we sort the list A B B A, we get A A B B. But is the relative order of the duplicates the same in the final order as it was in the original order? If that property is guaranteed, we have a stable sort.

Stable sort A sorting algorithm that preserves the order of duplicates

In our descriptions of the various sorts, we showed examples of sorting arrays of integers. Stability is not important when sorting primitive types. If instead we sort objects, the stability of a sorting algorithm can become more important. We may want to preserve the original order of objects considered identical by the comparison operation.

Suppose the items in our array are student objects with instance values representing their names, zip codes, and identification numbers. The list may normally be sorted by the unique identification numbers. For some purposes we might want to see a listing in order by name. In this case the comparison would be based on the name variable. To sort by zip code, we would sort on that instance variable.

If the sort is stable, we can get a listing by zip code, with the names in alphabetical order within each zip code, by sorting twice: the first time by name and the second time by zip code. A stable sort preserves the order of the elements when there is a match. The second sort, by zip code, produces many such matches, but the alphabetical order imposed by the first sort is preserved.

Of the sorts that we have discussed in this book, only `heapSort` and `quickSort` are inherently unstable. The stability of the other sorts depends on what the code does with duplicate values. In some cases, stability depends on whether a `<` or a `<=` comparison is used in some crucial comparison statement. In the exercises, you are asked to examine the code for the other sorts as we have coded them and determine whether they are stable.

Of course, if you can directly control the comparison operation used by your sort method, you can allow more than one variable to be used in determining a sort order. So another, more efficient approach to sorting our students by zip code and name is to define an appropriate `compareTo` method for determining sort order as follows (for simplicity, this assumes we can directly compare the name values):

```
if (zipcode < other.zipcode)
  return -1;
else
if (zipcode > other.zipcode)
  return +1;
else
// Zipcodes are equal
if (name < other.name)
  return -1;
else
if (name > other.name)
  return +1;
else
  return 0;
```

With this approach we need to sort the array only once.

10.5 Searching

As we have seen throughout the text, for each particular structure used to hold data, the methods that allow access to elements in the structure must be defined. In some cases access is limited to the elements in specific positions in the structure, such as the top element in a stack or the front element in a queue. Often, when data are stored in a list or a table, we want to be able to access any element in the structure.

Sometimes the retrieval of a specified element can be performed directly. For instance, the fifth element of the list stored *sequentially* in an array-based list called `list` is found in `list[4]`. Often, however, you want to access an element according to some key value. For instance, if a list contains student records, you may want to find the record of the student named Suzy Brown or the record of the student whose ID number is 203557. In cases like these, some kind of *searching technique* is needed to allow retrieval of the desired record.

For each of the techniques we review or introduce, our search operation must meet the following specifications. Note that we are talking about techniques within the class, not client code.

FindItem(item) return location

Effect:	Determines whether an element in the list has a key that matches item's, and if so returns its location
Preconditions:	List has been initialized. Item's key has been initialized.
Postcondition:	Location = position of element whose key matches item's key, if it exists; otherwise, location = NULL.

This specification has been written to apply to both array-based and linked lists, where `location` would be either an index in an array-based list or a reference in a linked list, and `NULL` would be either −1 in an array-based list or the `null` reference in a linked list.

Linear Searching

We cannot discuss efficient ways to find an element in a list without considering how the elements were inserted into the list. Therefore, our discussion of search algorithms is related to the issue of the list's `insert` operation. Suppose that we want to insert elements as quickly as possible, and we are not as concerned about how long it takes to find them. We would put the element into the last slot in an array-based list and the first slot in a linked list. These are O(1) insertion algorithms. The resulting list is sorted according to the time of insertion, not according to key value.

To search this list for the element with a given key, we must use a simple *linear* (or *sequential*) *search*. Beginning with the first element in the list, we search for the desired element by examining each subsequent item's key until either the search is successful or the list is exhausted:

Linear Search (unsorted data): returns location

```
Initialize location to position of first item
Set found to false
Set moreToSearch to (have not examined last.info( ))
while moreToSearch AND NOT found
      if item equals location.info( )
            Set found to true
      else
            Set location to location.next( )
            Set moreToSearch to (have not examined last.info( ))
if not found
      Set location to NULL
return location
```

Based on the number of comparisons, it should be obvious that this search is O(N), where N represents the number of elements. In the worst case, in which we are looking for the last element in the list or for a nonexistent element, we have to make N key comparisons. On the average, assuming that there is an equal probability of searching for any item in the list, we make $N/2$ comparisons for a successful search; that is, on the average we have to search half of the list.

High-Probability Ordering

The assumption of equal probability for every element in the list is not always valid. Sometimes certain list elements are in much greater demand than others. This observation suggests a way to improve the search: Put the most-often-desired elements at the beginning of the list. Using this scheme, you are more likely to make a hit in the first few tries, and rarely do you have to search the whole list.

If the elements in the list are not static or if you cannot predict their relative demand, you need some scheme to keep the most frequently used elements at the front of the list. One way to accomplish this goal is to move each element accessed to the front of the list. Of course, there is no guarantee that this element is later frequently used. If the element is not retrieved again, however, it drifts toward the end of the list as other elements are moved to the front. This scheme is easy to implement for linked lists, requiring only a couple of pointer changes, but it is less desirable for lists kept sequentially in arrays, because of the need to move all the other elements down to make room at the front.

A second approach, which causes elements to move toward the front of the list gradually, is appropriate for either linked or array-based list representations. As an element is found, it is swapped with the element that precedes it. Over many list retrievals, the most frequently desired elements tend to be grouped at the front of the list. To implement this approach, we only need to modify the end of the algorithm to exchange the found element with the one before it in the list (unless it is the first element). This change should be documented; it is an unexpected side effect of searching the list.

Keeping the most active elements at the front of the list does not affect the worst case; if the search value is the last element or is not in the list, the search still takes N comparisons. This is still a $O(N)$ search. The *average* performance on successful searches should be better, however. Both of these algorithms depend on the assumption that some elements in the list are used much more often than others. If this assumption is not applicable, a different ordering strategy is needed to improve the efficiency of the search technique.

Lists in which the relative positions of the elements are changed in an attempt to improve search efficiency are called *self-organizing* or *self-adjusting* lists.

Key Ordering

If a list is sorted according to the key value, we can write more efficient search routines. To support a sorted list, we must either insert the elements in order, or we must sort the list before searching it. (Note that inserting the elements in order is an $O(N^2)$ process, as each insertion is $O(N)$.) If we insert each element into the next free slot, and then sort the list with a "good" sort, the process is $O(N \log_2 N)$.)

If the list is sorted, a sequential search no longer needs to search the whole list to discover that an element does *not* exist. It only needs to search until it has passed the element's logical place in the list—that is, until an element with a larger key value is encountered. Versions of the Sorted List ADT in Chapters 3 and 5 implement this search technique.

One advantage of linear searching of a sorted list is the ability to stop searching before the list is exhausted if the element does not exist. Again, the search is O(N)—the worst case, searching for the largest element, still requires N comparisons. The average number of comparisons for an unsuccessful search is now $N/2$, however, instead of a guaranteed N.

Another advantage of linear searching is its simplicity. The disadvantage is its performance: In the worst case you have to make N comparisons. If the list is sorted and stored in an array, you can improve the search time to a worst case of O($\log_2 N$) by using a binary search. However, efficiency is improved at the expense of simplicity.

Binary Searching

We know of a way to improve searching from O(N) to O($\log_2 N$). If the data elements are sorted and stored sequentially in an array, we can use a `binary` search. The binary search algorithm improves the search efficiency by limiting the search to the area where the element might be. The binary search algorithm takes a divide-and-conquer approach. It continually pares down the area to be searched until either the element is found or the search area is gone (the element is not in the list). We developed the binary Search algorithm in Chapter 3, and converted it to a recursive approach in Chapter 7.

The binary search, however, is not guaranteed to be faster for searching very small lists. Notice that even though the binary search generally requires fewer comparisons, each comparison involves more computation. When N is very small, this extra work (the constants and smaller terms that we ignore in determining the Big-O approximation) may dominate. Although fewer comparisons are required, each involves more processing. For instance, in one assembly-language program, the linear search required 5 time units per comparison, whereas the binary search took 35. For a list size of 16 elements, therefore, the worst-case linear search would require 5 * 16 = 80 time units. The worst-case binary search requires only 4 comparisons, but at 35 time units each, the comparisons take 140 time units. In cases where the number of elements in the list is small, a linear search is certainly adequate and sometimes faster than a binary search.

As the number of elements increases, however, the disparity between the linear search and the binary search grows very quickly. Look back at Table 3.2 to compare the rates of growth for the two algorithms.

Note that the binary search discussed here is appropriate only for list elements stored in a sequential array-based representation. After all, how can you efficiently find the midpoint of a linked list? However, you already know of a structure that allows you to perform a binary search on a linked data representation, the binary search tree. The operations used to search a binary tree were discussed in Chapter 8.

10.6 Hashing

So far, we have succeeded in paring down our O(N) search to O($\log_2 N$) by keeping the list sorted sequentially with respect to the key value. That is, the key in the first element is less than (or equal to) the key in the second element, which is less than the key in the

third, and so on. Can we do better than that? Is it possible to design a search of O(1); that is, one that has a constant search time, no matter where the element is in the list?

In theory, that is not an impossible dream. Let's look at an example, a list of employees of a fairly small company. Each of the 100 employees has an ID number in the range 0 to 99, and we want to access the employee information by the key `idNum`. If we store the elements in an array that is indexed from 0 to 99, we can directly access any employee's information through the array index. There is a one-to-one correspondence between the element keys and the array index; in effect, the array index functions as the key of each element.

In practice, however, this perfect relationship between the key value and the location of an element is not easy to establish or maintain. Consider a similar small company that uses its employees' five-digit ID number as the primary key. Now the range of key values is from 00000 to 99999. Obviously, it is impractical to set up an array of 100,000 elements, of which only 100 are needed, just to make sure that each employee's element is in a perfectly unique and predictable location.

What if we keep the array size down to the size that we actually need (an array of 100 elements) and use just the last two digits of the key to identify each employee? For instance, the element of employee 53374 is in `employeeList[74]`, and the element of employee 81235 will be in `employeeList[35]`. Note that the elements are not sorted according to the *value* of the key as they were in our earlier discussion; the position of employee 81235's information precedes that of employee 53374 in the array, even though the value of its key is larger. Instead, the elements are sorted with respect to *some function of the key value*.

Hash function A function used to manipulate the key of an element in a list to identify its location in the list

Hashing The technique used for ordering and accessing elements in a list in a relatively constant amount of time by manipulating the key to identify its location in the list

Hash table Term used to describe the data structure used to store and retrieve elements using hashing

This function is called a hash function, and the search technique we are using is called hashing. The underlying data structure is often called a hash table. In the case of the employee list, the hash function is `(key % 100)`. The key (`idNum`) is divided by 100, and the remainder is used as an index into the array of employee elements, as illustrated in Figure 10.19. This function assumes that the array is indexed from 0 to 99 (`MAX_ITEMS = 100`). The method to perform the conversion of key values to indexes is very simple:

```
int hash()
// Post: Returns an integer between 0 and MAX_ITEMS -1
{
  return (idNum % MAX_ITEMS);
}
```

Here we assume that `hash` is a public method of `itemType`, the type (class) of the items in the list, and that `idNum` is an instance variable of `itemType`. For example, to use hashing to facilitate access to the lists we defined in Chapter 3 we must create a new interface `Hashable`:

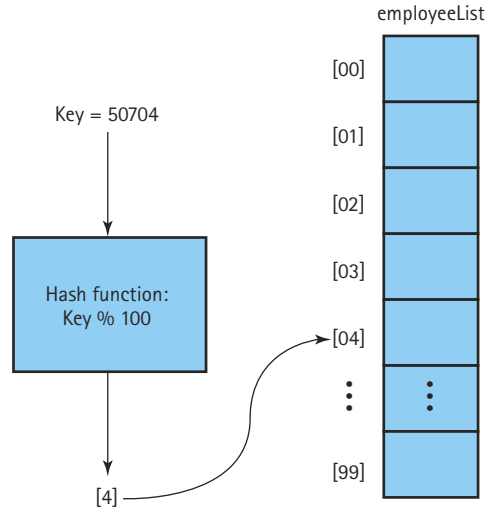


Figure 10.19 *Using a hash function to determine the location of the element in an array*

```
public interface Hashable extends Listable
// Objects of classes that implement this interface can be used
//  with lists based on hashing
{
  // A mathematical function used to manipulate the key of an element
  //  in a list to identify its location in the list
  public abstract int hash();
}
```

We can manipulate objects that implement the `Hashable` interface on our hash-based lists. In Chapter 3 we required such objects to implement the `Listable` interface. (`Listable` defines abstract methods `compareTo` and `copy`.) Note that the `Hashable` interface extends the `Listable` interface—objects that implement `Hashable` must export `hash`, `compareTo`, and `copy` methods.

The hash function has two uses. As we have seen, it is used to access a list element. The result of the hash function tells us where to *look* for a particular element—information we need to retrieve, modify, or delete the element. Here, for example, is a simple version of our list method `retrieve`, which assumes that the element is in the list.

```
public Hashable retrieve(Hashable item)
// Returns a copy of the list element in the array at position
//  item.Hash()
```

```
{
  int location;
  location = item.hash();
  return (Hashable)list[location].copy();
}
```

There is a second use of the hash function. It determines where in the array to store the element. If the employee list elements were inserted into the list using an `insert` operation from Chapter 3—into sequential array slots or into slots with their relative order determined by the key value—we could not use the hash function to retrieve them. We have to create a version of an `insert` operation that puts each new element into the correct slot according to the hash function. Here is a simple version of `insert`, which assumes that the array slot at the index returned from the hash function is not in use:

```
public void insert (Hashable item)
// Adds a copy of item to this list at position item.Hash()
{
  int location;
  location = item.hash();
  list[location] = item;
  numItems++;
}
```

Figure 10.20(a) shows an array whose elements—information for the employees with the key values (unique ID numbers) 12704, 31300, 49001, 52202, and 65606—were added using `insert`. Note that this function does not fill the array positions sequentially. Because we have not yet inserted any elements whose keys produce the hash values 3 and 5, the array slots [3] and [5] are logically "empty." This is different from the approach we used in Chapter 3 to create a sorted list. In Figure 10.20(b), the same

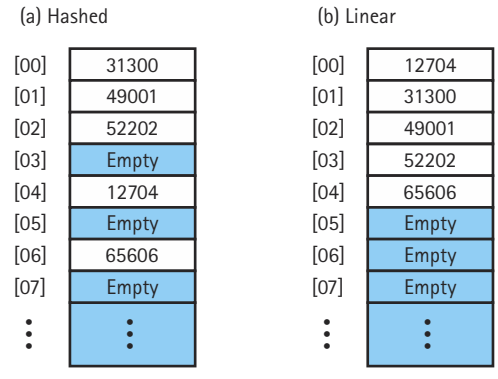


Figure 10.20 *Comparing hashed and sequential lists of identical elements*

employee records have been inserted into a sorted list using the `insert` operation from Chapter 3. Note that, unless the hash function was used to determine where to insert an element, the hash function is useless for finding the element.

Collisions

By now you are probably objecting to this scheme on the grounds that it does not guarantee unique hash locations. ID number 01234 and ID number 91234 both "hash" to the same location: `list[34]`. The problem of avoiding these collisions is the biggest challenge in designing a good hash function. A good hash function *minimizes collisions* by spreading the elements uniformly throughout the array. We say "minimizes collisions" because it is extremely difficult to avoid them completely.

Collision The condition resulting when two or more keys produce the same hash location

Assuming that there are some collisions, where do you store the elements that cause them? We briefly describe several popular collision-handling algorithms in the next section. Note that the scheme that is used to find the place to store an element determines the method subsequently used to retrieve it.

Linear Probing

A simple approach to resolving collisions is to store the colliding element into the next available space. This technique is known as linear probing. In the situation in Figure 10.21, we want to add the employee element

Linear probing Resolving a hash collision by sequentially searching a hash table beginning at the location returned by the hash function

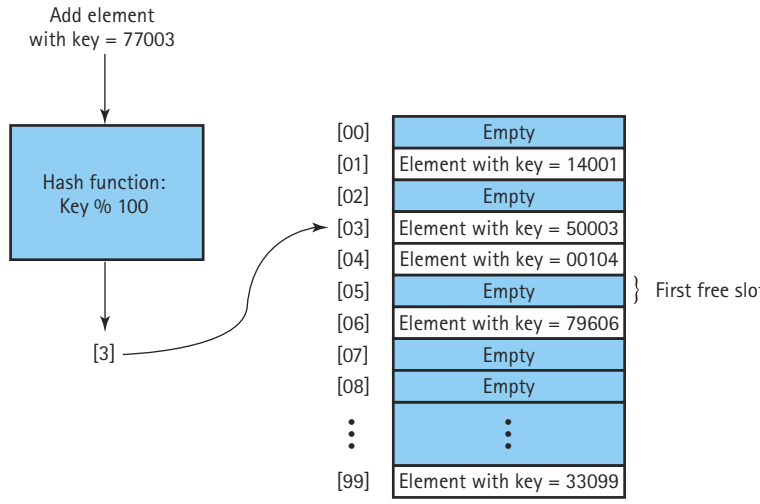


Figure 10.21 *Handling collisions with linear probing*

with the key ID number 77003. The hash function returns 3. But there already is an element stored in this array slot, the record for Employee 50003. We increment `location` to 4 and examine the next array slot. `list[4]` is also in use, so we increment `location` again. This time we find a slot that is empty, so we store the new element into `list[5]`.

What happens if the key hashes to the last index in the array and that space is in use? We can consider the array as a circular structure and continue looking for an empty slot at the beginning of the array. This situation is similar to our circular array-based queue in Chapter 4. There we used the `%` operator when we incremented our index. We can use similar logic here.

How do we know whether an array slot is "empty"? Assuming we have an array of objects, this is easy—just check to see if the value of the array slot is `null`.

Here is a version of `insert` that uses linear probing to find a place to store a new element. It assumes that there is room in the array for another element; that is, the client checks for `isFull` before the method is called. (We have retained the `numItems` instance variable of our lists. Even though it no longer tells us where the end of the list is, it is still useful in determining whether the list is full.)

```
public static void insert (Hashable item)
// Adds a copy of item to this list at position item.Hash()
//  or the next free spot
{
  int location;
  location = item.hash();
  while (list[location] != null)
    location = (location + 1) % list.length;
  list[location] = item;
  numItems++;
}
```

To search for an element using this collision-handling technique, we perform the hash function on the key, then compare the desired key to the actual key in the element at the designated location. If the keys do not match, we use linear probing, beginning at the next slot in the array. Following is a version of the `retrieve` method that uses this approach. Recall that our `retrieve` method for lists assumes that the item being retrieved is on the list.

```
public static Hashable retrieve(Hashable item)
// Returns a copy of the list element that matches item
{
  int location;
  location = item.hash();
  while (list[location].compareTo(item) != 0)
    location = (location + 1) % list.length;

  return (Hashable)list[location].copy();
}
```

We have discussed the insertion and retrieval of elements in a hash table, but we have not yet mentioned how to determine whether an item is in the table (`isThere`) or how to remove an element from the table (`delete`). If we did not need to concern ourselves with collisions, the algorithms would be simple:

isThere (item): return boolean

```
Set location to item.hash( )
Return (list[location] != null)
```

delete (item)

```
Set location to item.hash( )
Set list[location] to null
```

Collisions, however, complicate the matter. We cannot be sure that our item is in location `item.hash()`. For `delete`, since we assume the item to be deleted is in the table, we can use the same approach we used for `retrieve`. We examine every array element, starting with location `item.hash()`, until we find the matching element. For `isThere` we need an extra check to see if we have looped all the way back to our starting position without finding a match, in which case we return `false`.

Let's look at an example. In Figure 10.22, suppose we delete the element with the key 77003 by setting the array slot [5] to `null`. A subsequent search for the element with the key 42504 would begin at the hash location [4]. The element in this slot is not the one we are looking for, so we increment the hash location to [5]. This slot, which

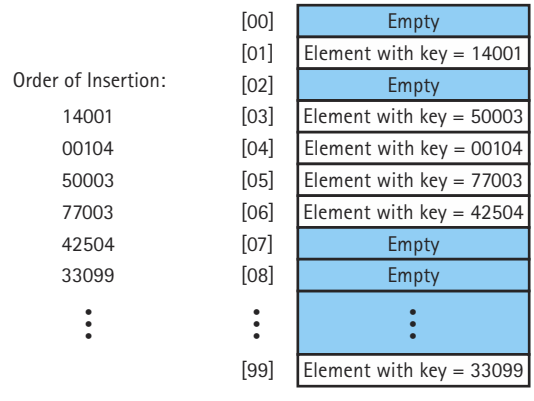


Figure 10.22 *A hash program with linear probing*

formerly was occupied by the element that we deleted, is now empty (contains `null`), but we cannot terminate the search—the record that we are looking for is in the next slot.

Not being able to assume that an empty list element indicates the end of a linear probe severely undermines the efficiency of this approach. Even when the hash table is sparsely populated, we must examine every location before determining that an item is not contained in the table. For the List ADT defined in Chapter 3, this search operation must precede every `insert`, `delete`, and `retrieve` operation. This problem illustrates that hash tables, in the forms that we have studied thus far, are not the most effective data structure for implementing lists whose elements may be deleted.

Clustering The tendency of elements to become unevenly distributed in the hash table, with many elements clustering around a single hash location

Clustering

One problem with linear probing is that it results in a situation called clustering. A good hash function results in a uniform distribution of indexes throughout the array's index range. Initially, therefore, items are inserted throughout the array, each slot equally likely to be filled. Over time, however, after a number of collisions have been resolved, the distribution of elements in the array becomes less and less uniform. The elements tend to cluster together, as multiple keys begin to compete for a single hash location.

Consider the hash table in Figure 10.22. Only an element whose key produces the hash value 8 would be inserted into array slot [8]. However, any elements with keys that produce the hash values 3, 4, 5, 6, or 7 would be inserted into array slot [7]. That is, array slot [7] is five times as likely as array slot [8] to be filled. Clustering results in inconsistent efficiency of list operations.

Rehashing Resolving a collision by computing a new hash location from a hash function that manipulates the original location rather than the element's key

Rehashing

The technique of linear probing discussed here is an example of collision resolution by rehashing. If the hash function produces a collision, the hash value is used as the input to a rehash function to compute a new hash value. In the previous section, we added 1 to the hash value to create a new hash value; that is, we used the rehash function:

```
(HashValue + 1) % MAX_ITEMS
```

For rehashing with linear probing, you can use any function

```
(HashValue + constant) % array-size
```

as long as `constant` and `array-size` are relatively prime—that is, if the largest number that divides both of them evenly is 1. For instance, given the 100-slot array in Figure 10.23, we might use the constant 3 in the rehash function:

```
(HashValue + 3) % 100
```

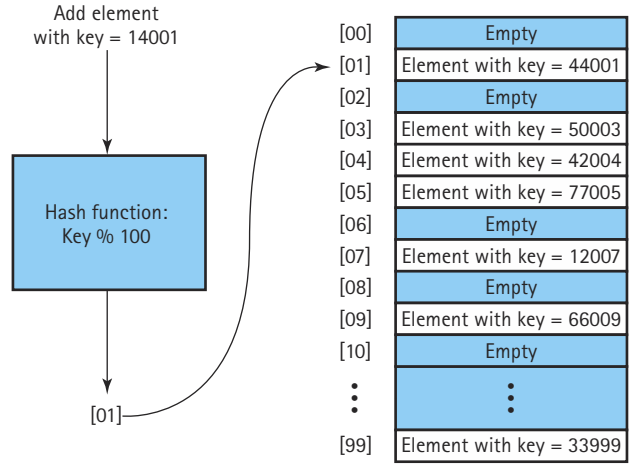


Figure 10.23 *Handling collisions with rehashing*

(Though 100 is not a prime number, 3 and 100 are relatively prime; they have no common factor larger than 1.)

Suppose that we want to add an element with the key 14001 to the hash table in Figure 10.23. The original hash function (Key % 100) returns the hash value 1, but this array slot is in use; it contains the element with the key 44001. To determine the next array slot to try, we apply the rehash function using the results of the first hash function as input: (1 + 3) % 100 = 4. The array slot at index [4] is also in use, so we reapply the rehash function until we get an available slot. Each time, we use the value computed from the previous rehash as input to the rehash function. The second rehash gives us (4 + 3) % 100 = 7; this slot is in use. The third rehash gives us (7 + 3) % 100 = 10; the array slot at index [10] is empty, so the new element is inserted there.

To understand why the constant and the number of array slots must be relatively prime, consider the rehash function

```
(HashValue + 2) % 100
```

We want to add the element with the key 14001 to the hash table pictured in Figure 10.23. The original hash function, key % 100, returns the hash value 1. This array slot is already occupied. We resolve the collision by applying the rehash function above, examining successive odd-numbered indexes until a free slot is found. What happens if all of the slots with odd-numbered indexes are already in use? The search would fail—even though there are free slots with even-numbered indexes. This rehash function does not cover the full index range of the array. However, if the constant and the number of array slots are relatively prime (like 3 and 100), the function produces successive rehashes that eventually cover every index in the array.

Rehash functions that use linear probing do not eliminate clustering (although the clusters are not always visually apparent in a figure). For example, in Figure 10.23, any element with a key that produces the hash value 1, 4, 7, or 10 would be inserted into the slot at index [10].

Quadratic probing Resolving a hash collision by using the rehashing formula `(HashValue + I²) % array-size`, where `I` is the number of times that the rehash function has been applied

Random probing Resolving a hash collision by generating pseudo-random hash values in successive applications of the rehash function

In linear probing, we add a constant (usually 1) in each successive application of the rehash function. Another approach, called quadratic probing, makes the result of rehashing dependent on how many times the rehash function has been applied. In the *I*th rehash, the function is:

```
(HashValue + I²) % array-size
```

The first rehash adds 1 to `HashValue`, the second rehash adds 4, the third rehash adds 9, and so on. Quadratic probing reduces clustering, but it does not necessarily examine every slot in the array. For example, if array-size is a power of 2 (512 or 1024, for example), relatively few array slots are examined. However, if array-size is a prime number of the form (4 * some-integer + 3), quadratic probing does examine every slot in the array.

A third approach uses a pseudo-random number generator to determine the increment to `HashValue` in each application of the rehash function. Random probing is excellent for eliminating clustering, but it tends to be slower than the other techniques we have discussed.

Buckets and Chaining

Another alternative for handling collisions is to allow multiple element keys to hash to the same location. One solution is to let each computed hash location contain slots for multiple elements, rather than just a single element. Each of these multi-element locations is called a bucket. Figure 10.24 shows a hash table with buckets that can contain three elements each. Using this approach, we can allow collisions to produce duplicate entries at the same hash location, up to a point. When the bucket becomes full, we must again deal with handling collisions.

Bucket A collection of elements associated with a particular hash location

Chain A linked list of elements that share the same hash location

Another solution, which avoids this problem, is to use the hash value not as the actual location of the element, but as the index into a linked list of elements. Each array slot accesses a chain of elements that share the same hash location. Figure 10.25 illustrates this solution to the problem of collisions. Rather than rehashing, we simply allow both elements to share hash location [3]. The entry in the array at this location contains a reference to a linked list that includes both elements.

To search for a given element, you first apply the hash function to the key and then search the chain for the element. Searching is not eliminated, but it is limited to elements that actually share a hash value. In contrast, with linear probing you may have to search through many additional elements if the slots following the hash location are filled with elements from collisions on other hash locations.

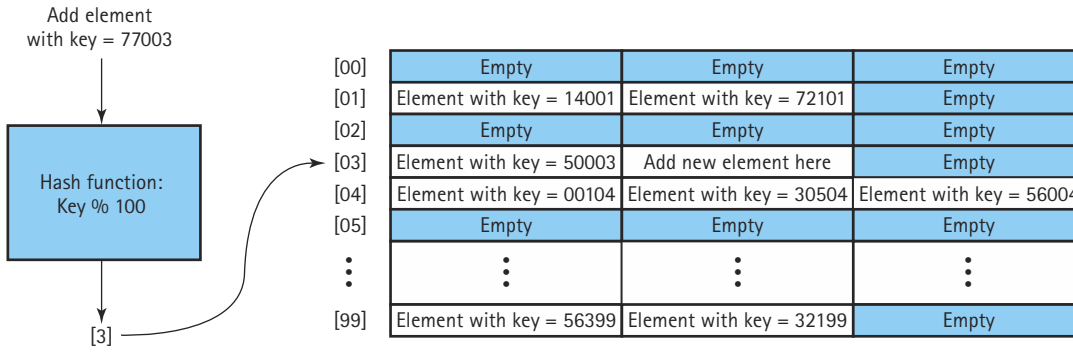


Figure 10.24 *Handling collisions by hashing with buckets*

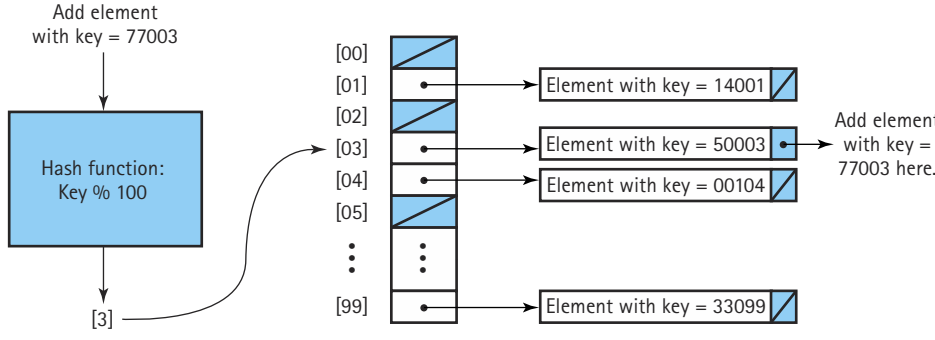


Figure 10.25 *Handling collisions by hashing with chaining*

Figure 10.26 illustrates a comparison of the chaining and hash-and-search schemes. The elements were added in the following order:

45300
20006
50002
40000
25001
13000
65905
30001
95000

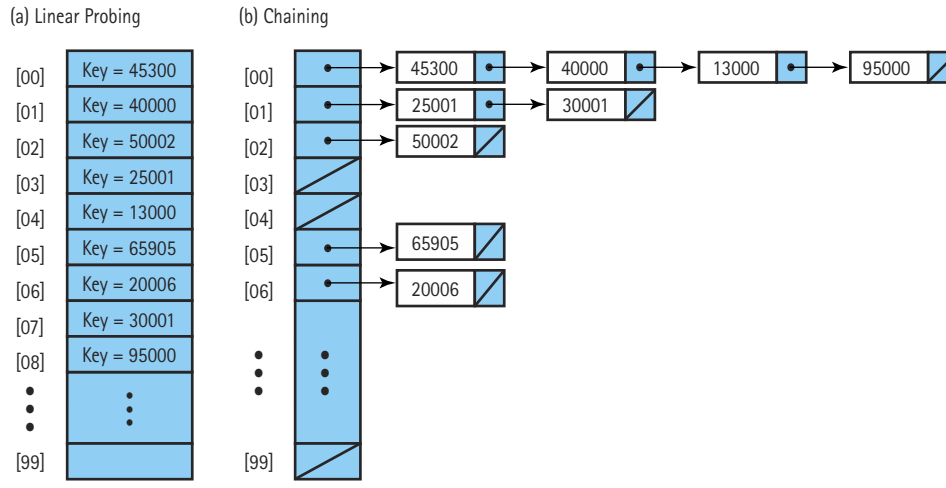


Figure 10.26 *Comparison of linear probing and chaining schemes*

Figure 10.26(a) represents the linear probing approach to collision handling; Figure 10.26(b) shows the result of chaining the colliding elements. Let's search for the element with the key 30001.

Using linear probing, we apply the hash function to get the index [1]. Because `list[1]` does not contain the element with the key 30001, we search sequentially until we find the element in `list[7]`. This requires seven steps.

Using the chaining approach, we apply the hash function to get the index [1]. `list[1]` directs us to a chain of elements whose keys hash to 1. We search this linked list until we find the element with the desired key. This requires only two steps.

Another advantage of chaining is that it simplifies the deletion of elements from the hash table. We apply the hash function to obtain the index of the array slot that contains the pointer to the appropriate chain. The node can then be deleted from this chain using the linked-list algorithm from Chapter 5.

Choosing a Good Hash Function

One way to minimize collisions is to use a data structure that has more space than is actually needed for the number of elements, in order to increase the range of the hash function. In practice it is desirable to have the array size somewhat larger than the number of elements required, in order to reduce the number of collisions. (This is assuming you are not using a method that requires the `isThere` method to continue searching when it encounters an empty array slot.)

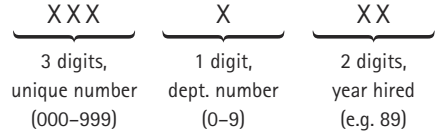
Selecting the table size involves a space-versus-time tradeoff. The larger the range of hash locations, the less likely it is that two keys hash to the same location. However, allocating an array that contains a large number of empty slots wastes space.

More important, you can design your hash function to minimize collisions. The goal is to distribute the elements as uniformly as possible throughout the array. Therefore, you want your hash function to produce unique values as often as possible. Once you admit collisions, you must introduce some sort of searching, either through array or chain searching or through rehashing. The access to each element is no longer direct, and the search is no longer O(1). In fact, if the collisions cause very disproportionate chains, the worst case may be almost O(*N*).

To avoid such a situation, you need to know something about the statistical distribution of keys. Imagine a company whose employee information is sorted according to a company ID six digits long. There are 500 employees, and we decide to use a chained approach to handling collisions. We set up 100 chains (expecting an average of five elements per chain) and use the hash function

```
idNum % 100
```

That is, we use the last two digits of the six-digit ID number as our index. The planned hash scheme is shown in Figure 10.27(a). Figure 10.27(b) shows what happened when the hash scheme was implemented. How could the distribution of the elements have come out so skewed? It turns out that the company's ID number is a concatenation of three values:



The hash scheme depended solely on the year hired to produce hash values. Because the company was founded in 1987, all the elements were crowded very disproportionately into a small subset of the hash locations. A search for an employee element, in this case, is O(*N*). Although this is an exaggerated example, it illustrates the need to understand as completely as possible the domain and predicted values of keys in a hash scheme. In the example situation it is much better to use some combination of the first three digits for the hash function.

Division Method

The most common hash functions use the division method (%) to compute hash values. This is the type of function used in the preceding examples. The general function is

```
Key % TableSize
```

We have already mentioned the idea of making the table somewhat larger than the number of elements required, in order to increase the range of hash values. In addition, it has been found that better results are produced with the division method when the table size is a prime number.

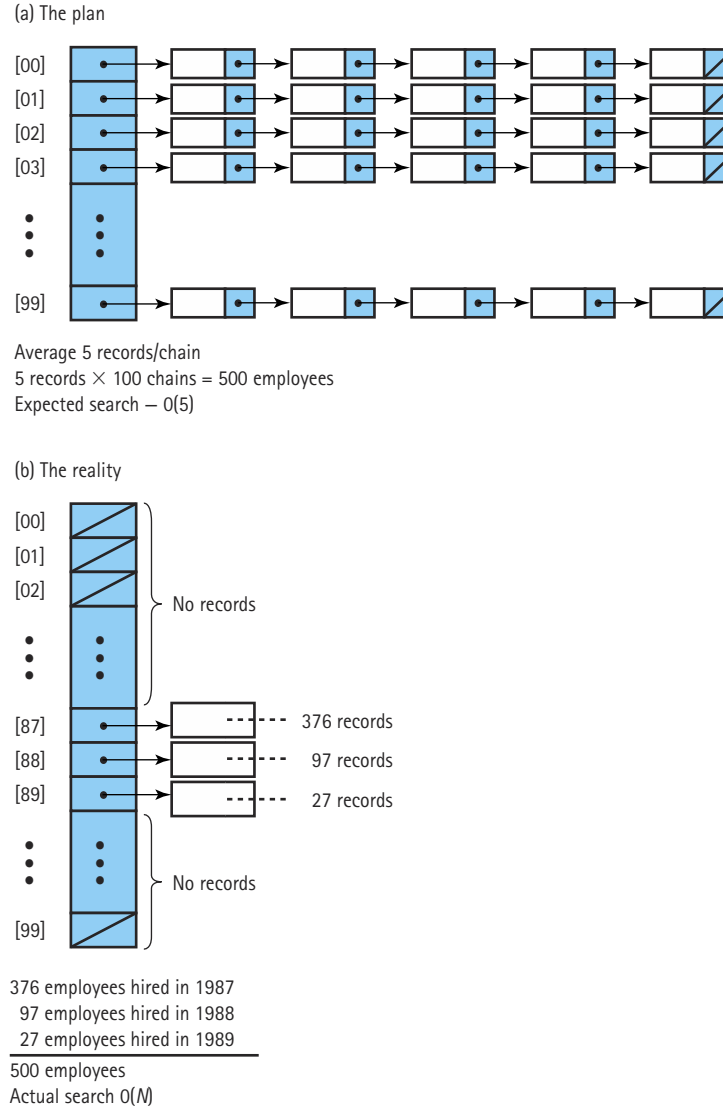


Figure 10.27 *Hash scheme to handle employee elements*

The advantage of the division hash function is simplicity. Sometimes, however, it is necessary to use a more complicated (or even exotic) hash function to get a good distribution of hash values.

Other Hash Methods

How can we use hashing if the element key is a string instead of an integer? One approach is to use an arithmetic combination of the internal representations of the

string's characters to create a number that can be used as an index. (Each Unicode character is represented in memory as an integer.)

Folding A hash method that breaks the key into several pieces and concatenates or exclusive-ORs some of them to form the hash value

A hash method called folding involves breaking the key into several pieces and concatenating or exclusive-OR'ing some of them to form the hash value. Another method is to square the key and then use some of the digits (or bits) of the key as a hash value. There are a number of other techniques, all of which are intended to make the hash location as unique and random (within the allowed range) as possible.

When using an exotic hash function, you should keep two considerations in mind. First, you should consider the efficiency of calculating the function. Even if a hash function always produces unique values, it is not a good hash function if it takes longer to calculate the hash value than to search half the list. Second, you should consider programmer time. An extremely exotic function that somehow produces unique hash values for all of the known key values may fail if the domain of possible key values changes in a later modification. The programmer who has to modify the program may then waste a lot of time trying to find another hash function that is equally clever.

Finally, we should mention that if you know all of the possible keys ahead of time, it is possible to determine a *perfect* hash function. For example, if you needed a list of elements whose keys were the reserved words in a computer language, you could find a hash function that hashes each word to a unique location. In general, it takes a great deal of work to discover a perfect hash function. And usually, we find that its computational complexity is very high, perhaps comparable to the effort required to execute a binary search.

Java's Support for Hashing

The Java Library includes a `HashTable` class that uses hash techniques to support storing objects in a table. In fact, the library includes several other collection classes, such as `HashSet`, that provide an ADT whose underlying implementation uses the approaches described in this section.

The Java `Object` class exports a `hashCode` method that returns an `int` hash code. Since all Java objects ultimately inherit from `Object`, all Java objects have an associated hash code. This is the hash value that Java uses within its hash-based library classes.

The standard Java hash code for an object is a function of the object's memory location. This means it cannot be used to relate separate objects with identical contents. For example, even if `circleA` and `circleB` have identical field values, it is very unlikely that they have the same hash code. Of course, if `circleA` and `circleB` both reference the same circle object, then their hash codes are identical since they hold the same memory reference.

For most applications, hash codes based on memory locations are not usable. Therefore, many of the Java classes that define commonly used objects (such as `String` and `Integer`), override the `Object` class's `hashCode` method with one that *is* based on the contents of the object. If you plan to use hash tables in your programs, you should do likewise.

Complexity

We began the discussion of hashing by trying to find a list implementation where the insertion and deletion were O(1). If our hash function never produces duplicates and the array size is large compared to the expected number of items in the list, then we have reached our goal. In general, this is not the case. Clearly, as the number of elements approaches the array size, the efficiency of the algorithms deteriorates. A precise analysis of the complexity of hashing is beyond the scope of this book. Informally, we can say that the larger the array is relative to the expected number of elements, the more time-efficient the algorithms are.

Summary

We have not attempted in this chapter to describe every known sorting algorithm. We have presented a few of the popular sorts, of which many variations exist. It should be clear from this discussion that no single sort is best for all applications. The simpler, generally $O(N^2)$ sorts work as well, and sometimes better than the more complicated sorts, for fairly small values of N. Because they are simple, these sorts require relatively little programmer time to write and maintain. As you add features to improve sorts, you also add to the complexity of the algorithms, increasing both the work required by the routines and the programmer time needed to maintain them.

Another consideration in choosing a sort algorithm is the order of the original data. If the data are already sorted (or almost sorted), `shortBubble` is $O(N)$, whereas some versions of a `quickSort` are $O(N^2)$.

As always, the first step in choosing an algorithm is to determine the goals of the particular application. This step usually narrows down the options considerably. After that, knowledge of the strong and weak points of the various algorithms assists you in making a choice.

The following table compares the sorts discussed in this chapter, in terms of Big-O.

Table 10.3 *Comparison of Sorting Algorithms*

	Order of Magnitude		
Sort	Best Case	Average Case	Worst Case
`selectionSort`	$O(N^2)$	$O(N^2)$	$O(N^2)$
`bubbleSort`	$O(N^2)$	$O(N^2)$	$O(N^2)$
`shortBubble`	$O(N)$ (*)	$O(N^2)$	$O(N^2)$
`insertionSort`	$O(N)$ (*)	$O(N^2)$	$O(N^2)$
`mergeSort`	$O(N\log_2 N)$	$O(N\log_2 N)$	$O(N\log_2 N)$
`quickSort`	$O(N\log_2 N)$	$O(N\log_2 N)$	$O(N^2)$ (depends on split)
`heapSort`	$O(N\log_2 N)$	$O(N\log_2 N)$	$O(N\log_2 N)$

*Data almost sorted.

Searching, like sorting, is a topic that is closely tied to the goal of efficiency. We speak of a sequential search as an O(N) search, because it may require up to N comparisons to locate an element. (N refers to the number of elements in the list.) Binary searches are considered to be O($\log_2 N$) and are appropriate for arrays only if they are sorted. A binary search tree may be used to allow binary searches on a linked structure. The goal of hashing is to produce a search that approaches O(1) time efficiency. Because of collisions of hash locations, some searching or rehashing is usually necessary. A good hash function minimizes collisions and distributes the elements randomly throughout the table.

It is important to be familiar with several of the basic sorting and searching techniques. These are tools that you use over and over again in a programming environment, and you need to know which ones are appropriate solutions to different problems. Our review of sorting and searching techniques has given us another opportunity to examine a measuring tool—the Big-O approximation—that helps us determine how much work is required by a particular algorithm. Both building and measuring tools are needed to construct sound program solutions.

Summary of Classes and Support Files

The classes and files are listed in the order in which they appear in the text. Inner classes are not included.

The package a class belongs to, if any, is listed in parenthesis under Notes. The class and support files are available on our web site. They can be found in the `ch10` subdirectory of the `bookFiles` directory.

Classes, Interfaces, and Support Files Defined in Chapter 10

File	1st Ref.	Notes
`Sorts.java`	page 675	Test harness for the sorting methods—includes the code for all of the sorting algorithms covered in the chapter
`SortCircle.java`	page 714	(`ch10.circles`) Defines circle objects to use with `Sorts2`
`Sorts2.java`	page 716	Test harness for using the `Comparator` approach with Selection Sort

On the next page is a list of the Java Library Classes and Interfaces that were used in this chapter for the first time in the textbook. The classes are listed in the order in which they are first used. Note that in some classes the methods listed might not be defined directly in the class; they might be defined in one of its superclasses. With the methods we also list constructors, if appropriate. For more information about the library classes and methods, the reader can check Sun's Java documentation.

Library Classes Used in Chapter 10 for the First Time:

Class/Interface Name	Package	Overview	Methods Used	Where Used
`Random.java`	`util`	Supports generation of random numbers	`Random`, `nextInt`	`Sorts` and `Sorts2`
`Comparator.java`	`util`	Interface; Supports generic sort routines	`compare`	`Sorts2`

Exercises

10.1 Sorting

1. A test harness program for testing sorting methods is provided on our web site. It is in the file `Sorts.java`. The program includes a `swap` method that is used by all of the sorting methods to swap array elements.
   a. Describe an approach to modifying the program so that after calling a sorting method the program prints out the number of swaps needed by the sorting method.
   b. Implement your approach.
   c. Test your new program by running the `selectionSort` method. Your program should report 49 swaps.

10.2 Simple Sorts

2. Multiple choice: How many comparisons would be needed to sort an array containing 100 elements using Selection Sort if the original array values were already sorted?
   a. 10,000 b. 9,900 c. 4,950 d. 99 e. None of these
3. Determine the Big-O measure for Selection Sort based on the number of elements moved rather than on the number of comparisons
   a. for the best case.
   b. for the worst case.
4. In what case(s), if any, is the Selection Sort $O(\log_2 N)$?
5. Write a version of the Bubble Sort algorithm that sorts a list of integers in descending order.
6. In what case(s), if any, is the Bubble Sort $O(N)$?
7. How many comparisons would be needed to sort an array containing 100 elements using `shortBubble`
   a. in the worst case?
   b. in the best case?

8. Show the contents of the array

43	7	10	23	18	4	19	5	66	14
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

after the fourth iteration of

a. `selectionSort`

b. `bubbleSort`

c. `insertionSort`

9. A sorting function is called to sort a list of 100 integers that have been read from a file. If all 100 values are zero, what would the execution requirements (in terms of Big-O) be if the sort used was

a. `bubbleSort`?

b. `shortBubble`?

c. `selectionSort`?

d. `insertionSort`?

10. In Exercise 1 you were asked to modify the `Sorts` program so that it would output the number of swaps used by a sorting method. It is a little more difficult to have the program also output the number of comparisons (compares) needed. You must include one or more statements to increment your counter within the sorting methods themselves. For each of the listed methods, make and test the changes needed, and list both the number of swaps and the number of compares needed by the `Sorts` program to sort an array of 50 random integers.

a. `selectionSort` swaps: _____ compares: _____

b. `bubbleSort` swaps: _____ compares: _____

c. `shortBubble` swaps: _____ compares: _____

d. `insertionSort` swaps: _____ compares: _____

10.3 O(N$\log_2$N) Sorts

11. A merge sort is used to sort an array of 1,000 test scores in descending order. Which one of the following statements is true?

a. The sort is fastest if the original test scores are sorted from smallest to largest.

b. The sort is fastest if the original test scores are in completely random order.

c. The sort is fastest if the original test scores are sorted from largest to smallest.

d. The sort is the same, no matter what the order of the original elements.

12. Show how the values in the array in Exercise 8 would be arranged immediately before the execution of method `merge` in the original (nonrecursive) call to `mergeSort`.

13. Determine the Big-O measure for `mergeSort` based on the number of elements moved rather than on the number of comparisons
    a. for the best case.
    b. for the worst case.
14. Use the Three-Question Method to verify `mergeSort`.
15. In what case(s), if any, is Quick Sort $O(N^2)$?
16. Which is true about Quick Sort?
    a. A recursive version executes faster than a nonrecursive version.
    b. A recursive version has fewer lines of code than a nonrecursive version.
    c. A nonrecursive version takes more space on the run-time stack than a recursive version.
    d. It can only be programmed as a recursive function.
17. Determine the Big-O measure for Quick Sort based on the number of elements moved rather than on the number of comparisons
    a. for the best case.
    b. for the worst case.
18. Use the Three-Question Method to verify `quickSort`.
19. Use the algorithms for creating a heap and sorting an array using a heap-based approach.
    a. Show how the values in the array in Exercise 8 would have to be rearranged to satisfy the heap property.
    b. Show how the array would look with four values in the sorted portion after reheaping.
20. A sorting function is called to sort a list of 100 integers that have been read from a file. If all 100 values are zero, what would the execution requirements (in terms of Big-O) be if the sort used was
    a. Merge Sort?
    b. Quick Sort, with the first element used as the split value?
    c. Heap Sort?
21. A list is sorted from smallest to largest when a sort is called. Which of the following sorts would take the longest time to execute and which would take the shortest time?
    a. Quick Sort, with the first element used as the split value
    b. Short Bubble
    c. Selection Sort
    d. Heap Sort
    e. Insertion Sort
    f. Merge Sort

22. A very large array of elements is to be sorted. The program is to be run on a personal computer with limited memory. Which sort would be a better choice: heap sort or merge sort? Why?

23. True or False? Explain your answers.

    a. Merge Sort requires more space to execute than heap sort.

    b. Quick Sort (using the first element as the split value) is better for nearly sorted data than Heap Sort.

    c. The efficiency of Heap Sort is not affected by the original order of the elements.

24. In Exercise 1 you were asked to modify the `Sorts` program so that it would output the number of swaps used by a sorting method. It is a little more difficult to have the program also output the number of comparisons needed. You must include one or more statements to increment your counter within the sorting methods themselves. For each of the listed methods, make and test the changes needed, and list the number of comparisons needed by `Sorts` to sort an array of 50 random integers.

    a. `mergeSort` compares: ____

    b. `quickSort` compares: ____

    c. `heapSort` compares: ____

10.4 More Sorting Considerations

25. For small values of N the number of steps required for a $O(N^2)$ sort might be less than the number of steps required for a sort of a lower degree. For each of the following pairs of mathematical functions f and g below, determine a value N such that if $n > N$, $g(n) > f(n)$. This value represents the cutoff point, above which the $O(n^2)$ function is always larger than the other function.

    a. $f(n) = 4n$ $\quad$ $g(n) = n^2 + 1$

    b. $f(n) = 3n + 20$ $\quad$ $g(n) = \frac{1}{2}n^2 + 2$

    c. $f(n) = 4 \log_2 n + 10$ $\quad$ $g(n) = n^2$

26. Give arguments for and against using methods (such as `swap`) to encapsulate frequently used code in a sorting routine.

27. Many times, in order to simplify our code, we create recursive methods of the form

```
if condition
  do something
```

    If the condition is false, the recursive method does nothing and returns. To avoid the overhead of extra method invocations, it is more efficient to perform the test on the condition before invoking the recursive method. Thus the method becomes

```
do something
```

An example of this is our `quickSort` method. Describe how you would change `quickSort` to avoid the overhead of unneeded invocations. Don't forget to address any needed change to the original invocation of `quickSort`.

28. What is meant by the statement that programmer time is an efficiency consideration? Give an example of a situation in which programmer time is used to justify the choice of an algorithm, possibly at the expense of other efficiency considerations.

29. Suppose a sorting method `doSort` operates on arrays of `Comparable` objects. Consider the `Circle` class defined in Chapter 2. Write a `compareTo` method for the `Circle` class so that the `doSort` method would sort an array of circles
    a. from smallest to largest.
    b. from right to left, based on the position of the center of the circle. (The coordinate system uses increasing values moving from left to right.)
    c. based on the distance of the circle from the point 0,0; shortest distance first.

30. Modify the `Sorts2` class so that it also sorts the array of `SortCircles`
    a. from smallest to largest.
    b. from right to left, based on the position of the center of the circle. (The coordinate system uses increasing values moving from left to right.)
    c. based on the distance of the circle from the point 0,0; shortest distance first.

31. The `Sorts2` class, provided on the CD, is similar to the `Sorts` class we used to study sorting algorithms. Instead of using integers, the `Sorts2` class generates an array of random `SortCircles`, and then sorts them first by `xValue` and then by `yValue`. Create an `isSorted` method for `Sorts2`, similar to the `isSorted` method of `Sorts` except it must accept and use a `Comparator` parameter. Modify `Sorts2` to include this new method; remember to call the method both before and after the calls to the sort routines.

32. Go through the sorting algorithms coded in this chapter and determine which ones are stable as coded. Identify the key statement in the corresponding method that determines the stability. If there are unstable algorithms (other than Quick Sort and Heap Sort), make them stable.

33. We said that Heap Sort is inherently unstable. Explain why.

34. Which sorting algorithm would you *not* use under each of the following conditions?
    a. The sort must be stable.
    b. Space is very limited.

10.5 Searching

35. Fill in the following table, showing the number of comparisons needed either to find the value or to determine that the value is not in the indicated structure based on the given approach and given the following values:

26, 15, 27, 12, 33, 95, 9, 5, 99, 14

Value	Unsorted Array in order shown	Sorted Array, Sequential Search	Sorted Array, Binary Search	Binary Search Tree (entered as shown)
15				
17				
14				
5				
99				
100				
0				

36. If you know the index of an element stored in an array of N unsorted elements, which of the following best describes the order of the algorithm to retrieve the element?

- **a.** $O(1)$
- **b.** $O(N)$
- **c.** $O(\log_2 N)$
- **d.** $O(N^2)$
- **e.** $O(0.5\ N)$

37. The element being searched for is not in an array of 100 elements. What is the *average* number of comparisons needed in a sequential search to determine that the element is not there

- **a.** if the elements are completely unsorted?
- **b.** if the elements are sorted from smallest to largest?
- **c.** if the elements are sorted from largest to smallest?

38. The element being searched for is not in an array of 100 elements. What is the *maximum* number of comparisons needed in a sequential search to determine that the element is not there

- **a.** if the elements are completely unsorted?
- **b.** if the elements are sorted from smallest to largest?
- **c.** if the elements are sorted from largest to smallest?

39. The element being searched for *is* in an array of 100 elements. What is the *average* number of comparisons needed in a sequential search to determine the position of the element
    a. if the elements are completely unsorted?
    b. if the elements are sorted from smallest to largest?
    c. if the elements are sorted from largest to smallest?
40. Choose the answer that correctly completes the following sentence: The elements in an array may be sorted by highest probability of being requested in order to reduce
    a. the average number of comparisons needed to find an element in the list.
    b. the maximum number of comparisons needed to detect that an element is not in the list.
    c. the average number of comparisons needed to detect that an element is not in the list.
    d. the maximum number of comparisons needed to find an element that is in the list.
41. Identify each of the following statements as True or False. Explain your answers.
    a. A binary search of a sorted set of elements in an array is always faster than a sequential search of the elements.
    b. A binary search is an $O(N\log_2 N)$ algorithm.
    c. A binary search of elements in an array requires that the elements be sorted from smallest to largest.
    d. A high-probability ordering scheme would be a poor choice for arranging an array of elements that are equally likely to be requested.
42. How might you order the elements in a list of Java's reserved words to use the idea of high-probability ordering?

10.6 Hashing

For Exercises 43–46, use the following values:

66, 47, 87, 90, 126, 140, 145, 153, 177, 285, 393, 395, 467, 566, 620, 735

43. Store the values into a hash table with 20 positions, using the division method of hashing and the linear probing method of resolving collisions.
44. Store the values into a hash table with 20 positions, using rehashing as the method of collision resolution. Use `key % tableSize` as the hash function, and `(key + 3) % tableSize` as the rehash function.
45. Store the values into a hash table with ten buckets, each containing three slots. If a bucket is full, use the next (sequential) bucket that contains a free slot.
46. Store the values into a hash table that uses the hash function `key % 10` to determine which of ten chains to put the value into.

47. Fill in the following table, showing the number of comparisons needed to find each value using the hashing representations given in Exercises 43–46.

	Number of Comparisons			
Value	Exercise 43	Exercise 44	Exercise 45	Exercise 46
66				
467				
566				
735				
285				
87				

48. Identify each of the following statements as True or False. Explain your answers.
    a. When a hash function is used to determine the placement of elements in an array, the order in which the elements are added does not affect the resulting array.
    b. When hashing is used, increasing the size of the array always reduces the number of collisions.
    c. If we use buckets in a hashing scheme, we do not have to worry about collision resolution.
    d. If we use chaining in a hashing scheme, we do not have to worry about collision resolution.
    e. The goal of a successful hashing scheme is an O(1) search.
49. Choose the answer that correctly completes the following sentence: The number of comparisons required to find an element in a hash table with N buckets, of which M are full
    a. is always 1.
    b. is usually only slightly less than N.
    c. may be large if M is only slightly less than N.
    d. is approximately $\log_2 M$.
    e. is approximately $\log_2 N$.

50. Write a program that repeatedly accepts a string from the user and outputs the hash code for the string, using the `String` class's predefined `hashCode` method.

51. Create a data set with 100 integer values. Use the division method of hashing to store the data values into hash tables with table sizes of 7, 51, and 151. Use the linear probing method of collision resolution. Print out the tables after the data values have been stored. Search for ten different values in each of the three hash tables, counting the number of comparisons necessary. Print out the number of comparisons necessary in each case in tabular form. Turn in a listing of your program and a listing of the output.

Appendix A

Java Reserved Words

```
abstract        do          if          package         synchronized
boolean         double      implements  private         this
break           else        import      protected       throw
byte            extends     instanceof  public          throws
case            false       int         return          transient
catch           final       interface   short           true
char            finally     long        static          try
class           float       native      strictfp        void
const           for         new         super           volatile
continue        goto        null        switch          while
default
```

Appendix B

Operator Precedence

In the following table, the operators are grouped by precedence level (highest to lowest), and a horizontal line separates each precedence level from the next-lower level.

Precedence (highest to lowest)

Operator	Assoc.*	Operand Type(s)	Operation Performed
.	LR	object, member	object member access
[]	LR	array, int	array element access
(args)	LR	method, arglist	method invocation
++, --	LR	variable	post-increment, decrement
++, --	RL	variable	pre-increment, decrement
+, -	RL	number	unary plus, unary minus
~	RL	integer	bitwise complement
!	RL	boolean	boolean NOT
new	RL	class, arglist	object creation
(type)	RL	type, any	cast (type conversion)
*, /, %	LR	number, number	multiplication, division, remainder
+, -	LR	number, number	addition, subtraction
+	LR	string, any	string concatenation
<<	LR	integer, integer	left shift
>>	LR	integer, integer	right shift with sign extension
>>>	LR	integer, integer	right shift with zero extension
<, <=	LR	number, number	less than, less than or equal
>, >=	LR	number, number	greater than, greater than or equal
instanceof	LR	reference, type	type comparison
==	LR	primitive, primitive	equal (have identical values)
!=	LR	primitive, primitive	not equal (have different values)
==	LR	reference, reference	equal (refer to the same object)
!=	LR	reference, reference	not equal (refer to different objects)
&	LR	integer, integer	bitwise AND
&	LR	boolean, boolean	boolean AND
^	LR	integer, integer	bitwise XOR
^	LR	boolean, boolean	boolean XOR

*LR means left to right associativity; RL means right to left associativity.

Precedence (highest to lowest)

Operator	Assoc.*	Operand Types(s)	Operation Performed
|	LR	integer, integer	bitwise OR
|	LR	boolean, boolean	boolean OR
&&	LR	boolean, boolean	conditional AND (short circuit evaluation)
||	LR	boolean, boolean	conditional OR (short circuit evaluation)
?:	RL	boolean, any, any	conditional (ternary) operator
=	RL	variable, any	assignment
*=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=	RL	variable, any	assignment with operation

*LR means left to right associativity; RL means right to left associativity.

Appendix C

Primitive Data Types

Type	Value Stored	Default Value	Size	Range of Values
char	Unicode character	Character code 0	16 bits	0 to 65535
byte	Integer value	0	8 bits	−128 to 127
short	Integer value	0	16 bits	−32768 to 32767
int	Integer value	0	32 bits	−2147483648 to 2147483647
long	Integer value	0	64 bits	−9223372036854775808 to 9223372036854775807
float	Real value	0.0	32 bits	±1.4E-45 to ±3.4028235E+38
double	Real value	0.0	64 bits	±4.9E-324 to ±1.7976931348623157E+308
boolean	true or false	false	1 bit	NA

Appendix D

ASCII Subset of Unicode

The following chart shows the ordering of characters in the ASCII (American Standard Code for Information Interchange) subset of Unicode. The internal representation for each character is shown in decimal. For example, the letter *A* is represented internally as the integer 65. The space (blank) character is denoted by a "□".

Left Digit(s) \ *Right Digit*	*ASCII*									
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
1	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
2	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
3	RS	US	□	!	"	#	$	%	&	'
4	(	)	*	+	,	–	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	DEL		

Codes 00–31 and 127 are the following nonprintable control characters:

NUL	Null character	VT	Vertical tab	SYN	Synchronous idle
SOH	Start of header	FF	Form feed	ETB	End of transmitted block
STX	Start of text	CR	Carriage return	CAN	Cancel
ETX	End of text	SO	Shift out	EM	End of medium
EOT	End of transmission	SI	Shift in	SUB	Substitute
ENQ	Enquiry	DLE	Data link escape	ESC	Escape
ACK	Acknowledge	DC1	Device control one	FS	File separator
BEL	Bell character (beep)	DC2	Device control two	GS	Group separator
BS	Back space	DC3	Device control three	RS	Record separator
HT	Horizontal tab	DC4	Device control four	US	Unit separator
LF	Line feed	NAK	Negative acknowledge	DEL	Delete

Answers to Selected Exercises

Chapter 1

Many of the questions in this chapter's exercises are "thought questions." The answers given here are typical or suggested responses, but they are not the only possible answers.

1. Software engineering is a disciplined approach to the creation and maintenance of computer programs.

3. (d) is correct. Although there is a general order to the activities, and in some cases it is desirable to finish one phase completely before beginning another, often the software phases overlap one another.

4. a. When the program's requirements change; when a better solution is discovered in the middle of the design phase; when an error is discovered in the requirements due to the design effort.

   b. When the program is being debugged, because of compilation errors or errors in the design; when a better solution is found for a part of the program that was already implemented; or when any of the situations in Part (a) occur.

   c. When there are errors that cause the program to crash or to produce wrong answers; or when any of the situations in Parts (a) or (b) occur.

   d. When an error is discovered during the use of the program; when additional functions are added to an existing software system; when a program is being modified to use on another computer system; or when any of the situations in Parts (a), (b), or (c) occur.

10. Top-down: First the problem is broken into several large parts. Each of these parts is in turn divided into sections, then the sections are subdivided, and so on.

    Bottom-up: With this approach the details come first. It is the opposite of the top-down approach. After the detailed components are identified and designed, they are brought together into increasingly high-level components.

Functional decomposition: This is a program design approach that encourages programming in logical action units, called functions. The main module of the design becomes the main program (also called the main function), and subsections develop into functions.

Round-trip gestalt design: First, the tangible items and events in the problem domain are identified and assigned to candidate classes and objects. Next the external properties and relationships of these classes and objects are defined. Finally, the internal details are addressed, and unless these are trivial, the designer must return to the first step for another round of design.

13. A class defines a structure or template for an object or a set of objects. An object is an instance of a class. An example is a blueprint of a building and the building itself. Another example, from the text, of a Java class/object is the `Date` class and the `myDate`, `yourDate`, `ourDate` objects.

15. Customer, bank card, ATM, PIN, account, account number, balance, display

16. a. Legal—`dayIs` is a public method that returns an `int`.
    b. Legal—`yearIs` is a public method that returns an `int`.
    c. Illegal—`increment` is not defined for `Date` objects.
    d. Legal—`increment` is defined for `IncDate` objects.
    e. Legal—Object variables can be assigned to objects of the same class.
    f. Legal—Subclasses are assignment-compatible with the superclasses above them in the class hierarchy.
    g. Illegal—Superclasses are not assignment-compatible with the subclasses below them in the class hierarchy

18. The correction of errors early in the program's life cycle involves less rework. The correction can be incorporated into the program design. Detected late in the life cycle, errors may necessitate redesign, recoding, and/or retesting. The later the error is detected, the more rework one is likely to have to do to correct it.

20. Program verification determines that the program fulfills the specified requirements; program validation determines if the program is as useful as possible for the customer. The former is measured against formal documentation; the latter is determined through observation and a thorough understanding of the problem domain.

23. The body of the *while* loop is not in braces.

    The comment includes the call to increment the `count` variable.

24. A single programmer could use the inspection process as a way to do a structured deskcheck. The programmer would especially benefit from inspection checklists of errors to look for.

25. a. It is appropriate to start planning a program's testing during the earliest phases of program development.

28.

Parameter 1	Parameter 2	Expected Result
0	0	true
5	5	true
−5	−5	true
5	−5	false
−5	5	false
5	0	false
−5	0	false
0	5	false
0	−5	false
27	3	true
−15	−34	true
27	−34	false

33. Life-cycle verification refers to the idea that program verification activities can be performed throughout the program's life cycle, not just by testing the program after it is coded.

Chapter 2

2. Data abstraction refers to the logical picture of the data—what the data represent rather than how they are represented.
3. Data encapsulation is the separation of the physical representation of data from the applications that use the data at a logical (abstract) level. When data abstraction is protected through encapsulation, the data user can deal with the data abstraction but cannot access its implementation, which is encapsulated. The data user accesses data that are encapsulated through a set of operations specified to create, access, and change the data. Data encapsulation is accomplished through a programming language feature.
5. a. Application level (e.g., College of Engineering's enrollment information for 1988)
   b. Abstract level (e.g., list of student academic records)
   c. Implementation level (e.g., array of objects that contain the variables `studentID` [an integer], `lastName` [a string of characters], `firstName` [a string of characters], and so forth)
6. a. Applications of type `GroceryStore` include the Safeway on Main Street, the Piggly Wiggly on Broadway, and the Kroger's on First Street.
   b. User operations include `SelectItem`, `CheckOut`, `PayBill`, and so on.
   c. Specification of `CheckOut` operation:

float CheckOut (Basket)

Effect:	Presents basket of groceries to cashier to check out; returns bill.
Precondition:	Basket is not empty.
Postconditions:	Return value = total charge for all the groceries in Basket.
	Basket contains all groceries arranged in paper sacks.

d. Algorithm for the `CheckOut` operation:

CheckOut

```
InitRegister
Set bill to 0
do
   OpenSack
   while More objects in Basket AND NOT SackFull
      Take object from Basket
      Set bill to bill + cost of this object
      Put object in sack
   Put full Sack aside
while more objects in Basket
   Put full sacks into Basket
return bill
```

e. The customer does not need to know the procedure that is used by the grocery store to check out a basket of groceries and to create a bill. The logical level (c) above provides the correct interface, allowing the customer to check out without knowing the implementation of the process.

7. Java's primitive types are `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, and `short`.

10. a.

```
public String toString()
{
  String temp;
  temp = " (" + xValue + "," + yValue + ")";
  temp = temp + "\n radius: " + radius;
  temp = temp + "\n solid: " + solid;
```

```
    return temp;
  }
```

b.
```
public String toString()
  {
    String temp;
    temp = " (" + location.xValue + "," + location.yValue + ")";
    temp = temp + "\n radius: " + radius;
    temp = temp + "\n solid: " + solid;
    return temp;
  }
```

12.
```
5/5/2000
5/5/2000
5/6/2000
5/6/2000
```

14. Garbage is memory space that has been allocated to a program but that can no longer be accessed by the program. Garbage can be created when a variable that is the only reference to an object is associated with a new object:

```
Circle c1 = new Circle();
c1 = new Circle();
```

or when it is assigned to a different object:

```
Circle c1 = new Circle();
Circle c2 = new Circle();
c1 = c2;
```

16. Final, static variables; abstract methods.

20.
```
import java.io.PrintWriter;
public class Exercise20
{
  static PrintWriter output = new PrintWriter(System.out,true);

  public static void main(String[] args)
    throws Exception
  {
    int[] squares = new int[10];

    for (int i = 0; i < 10; i++)
      squares[i] = i * i;

    for (int i = 0; i < 10; i++)
      output.println(squares[i]);
  }
}
```

24. The class

25. When a record is created, the instance variables are public and can be directly accessed from the class/program that uses the record; when an ADT is created, the instance variables are private and can only be accessed through the public methods of the class.

34. a. array
    b. array list
    c. array
    d. array
    e. array list

37. a.

SquareMatrix ADT Specification (assumes programming by contract)

Structure: An N × N square integer matrix.

Operations:

void MakeEmpty(int n)

Effect: Instantiates this matrix to size $n \times n$ and sets the values to zero.

Precondition: n is less than or equal to 50.

Postcondition: This matrix contains all zero values.

void StoreValue(int i, int j, int value)

Effect: Stores value into the i, jth position in this matrix.

Preconditions: This matrix has been initialized; i and j are between 0 and the size minus 1.

Postcondition: value has been stored into the i, jth position of this matrix.

SquareMatrix Add(SquareMatrixType one)

Effect: Adds this matrix and matrix one and returns the result.

Preconditions: This matrix and matrix one have been initialized and are the same size.

Postcondition: return value = this + one.

SquareMatrix Subtract(SquareMatrixType one)

Effect:	Subtracts matrix one from this matrix and returns the result.
Preconditions:	This matrix and matrix one have been initialized and are the same size.
Postcondition:	return value = this − two.

SquareMatrix Copy()

Effect:	Returns a copy of this matrix.
Precondition:	This matrix has been initialized.
Postcondition:	return value = copy of this matrix.

40. See the feature section Designing ADTs on page 130.

Chapter 3

2. a. Voter Identification Number
   b. A combination of their league name, team name, and team number
   c. Many schools have a student identification number.

3.

`UnsortedStringList`	Constructor
`isFull`	Observer
`lengthIs`	Observer
`isThere`	Observer
`insert`	Transformer
`delete`	Transformer
`reset`, `getNextItem`	Iterator

4. a.
```
private static boolean printLast(PrintWriter outFile,
UnsortedStringList list)
// Effect: Prints the last item on the list
// Pre:    List has been instantiated.
//         outFile is open for writing
// Post:   If the list is not empty
//            the last list item has been written to outFile.
//            return value = true
//         otherwise
//            "List is empty" has been written to the outFile
//            return value = false
```

```
{
  int length;
  String item = "";

  if (list.lengthIs() == 0)
  {
    outFile.println("List is empty");
    return false;
  }
  else
  {
    list.reset();
    length = list.lengthIs();
    for (int counter = 1; counter <= length; counter++)
      item = list.getNextItem();
    outFile.println(item);
    return true;
  }
}
```

b. For testing, the `printLast` method can be included in the `TDUnsortedStringList` application. At the end of the *while* loop of the test driver, after the call to `printList`, we can include the call:

```
exer4 = printLast(outFile, list);
outFile.println(exer4 + " returned from printLast");
outFile.println();
```

This assumes we have declared a `boolean` variable `exer4`. Now we can just run the set of test cases we used to test the ADT and ensure that `printLast` acts as expected in each case.

8. a. Syntax error occurs because the `length` attribute of the `list` array is a final variable and therefore cannot have a value assigned to it.

c. A fatal run-time error "NullPointerException" occurs when we pass the `isThere` method a value that is not on the list. This error occurs because we do not stop searching when the end of the list is reached.

e. Since the `numItems` variable is a crucial component for many of the list methods, many errors can result. A deleted item will still appear to be on the list. The size of the list will be reported incorrectly.

10. a.

boolean isEmpty()

Effect:	Determines whether this list is empty.
Postcondition:	Return value = (this list is empty).

b.
```
public boolean isEmpty()
// Returns whether this list is empty
{
  return (numItems == 0);
}
```

13. a.

boolean tryDelete (String item)

Effect:	If the list contains an element whose key matches item's key, deletes the element and returns true; Otherwise, returns false.
Precondition:	List is instantiated.
Postconditions:	No element on this list has a key matching the argument item's key. Return value = (list did contain an element matching item).

b.
```
public boolean tryDelete (String item)
// If an element on this list matches item
//    deletes it and returns true;
//  otherwise, returns false
{

  int match = -1;

  for (int location = 0; location < numItems; location++)
    if (item.compareTo(list[location]) == 0)  // If they match
      match = location;
```

```
    if (match == -1)  // Item not found
      return false;
    else
    {
      list[match] = list[numItems - 1];
      numItems--;
      return true;
    }
  }
```

18. a. `isThere`, `insert`, `delete`

b. `StringList`, `isFull`, `lengthIs`, `reset`, `getNextItem`

c. An abstract method does not have a method body. In the case of `StringList`, the abstract methods are the ones whose code depends upon whether or not the string list is sorted.

20. An abstract method. If the list is kept sorted, the smallest element is the first element on the list. This is not true for the unsorted case. Therefore, the implementation of this method depends on whether or not the list is sorted. So, it should be an abstract method, with the method body provided in the extension.

22. a.

String smallest ()

Effect:	Finds the smallest element of this list.
Precondition:	List is not empty.
Postcondition:	Return value = (copy of smallest value on this list).

b.

```
public String smallest ()
// Returns a copy of the smallest element from this list
{
 return new String(list[0]);
}
```

28. a. $O(N^2)$

c. $O(N^5)$

e. $O(N^4)$

30. b. $O(N^2)$

31. a. Algorithm 1 is $O(N^3)$; Algorithm 2 is $O(N)$.

b. Algorithm 2

c. $N^3 < 3N + 1000$

Rather than solve the cubic equation we can just use a table to answer the question:

N	N^3	3N + 1000
1	1	1003
2	8	1006
5	125	1015
10	1000	1030
11	1331	1033

Therefore, for $N <= 10$, the "less efficient" algorithm is faster.

32. a. O(*N*)—To print the list, we must visit every item.

c. O(1)—Simply return whether or not the number of items is zero.

d. O(*N*)—Assuming the `UnsortedStringList` is implemented as described in the text … but if we allow any implementation, then it could be O(1). (For example, the list could be kept sorted.)

e. O(*N*)—Since the client can only use the ADT as is, this is the only possible answer.

35. a.
```
public class ListNumber implements Listable
{
  private int value;
  private String word;

  public ListNumber(int x, String y)
  {
    this.value = x;
    this.word = y;
  }

  public int getValue()
  {
    return value;
  }

  public String getWord()
  {
    return word;
  }
```

```
  public int compareTo(Listable otherListNumber)
  {
    ListNumber other = (ListNumber)otherListNumber;
    return (int)(this.value - other.value);
  }

  public Listable copy()
  {
    ListNumber result = new ListNumber(this.value, this.word);
    return result;
  }

  public String toString()
  {
    return word;
  }
}
```

b. Here is one test program:

```
import java.io.*;
import ch03.genericLists.*;

// Test for Exercise 35, Chapter 3
public class TestExercise35
{
  public static void main(String[] args) throws IOException
  {
    SortedList list;
    int size;
    ListNumber temp;

    list = new SortedList();

    System.out.println("Test of Exercise 35 Results");
    System.out.println();

    list = new SortedList(5);;

    temp = new ListNumber(3, "three");
    list.insert(temp);

    temp = new ListNumber(2, "two");
    list.insert(temp);
```

```
        temp = new ListNumber(17, "seventeen");
        list.insert(temp);

        temp = new ListNumber(1, "one");
        list.insert(temp);

        temp = new ListNumber(27, "twenty-seven");
        list.insert(temp);

        list.reset();
        for (int i = 0; i <= 4; i++)
          System.out.println(list.getNextItem());
    }
}
```

We would expect the output from this program to be:

```
Test of Exercise 35 Results

one
two
three
seventeen
twenty-seven
```

37. a. Since the `HouseFile` class "hides" the information in the `houses.dat` file, it is the only class that needs to be changed. In its `getHouse` and `putHouse` methods we would just switch the order of handling first and last names. This is a good example of information hiding limiting the scope of changes.

c. This change would permeate through the entire application. The `ListHouse` class would need to expand its definition of what a house is to include the number of bathrooms. This would require adding a new instance variable, expanding the constrictor and `copy` methods to handle another parameter and adding a new observer method that returns the number of bathrooms. The `HouseFile` class would need one new read statement in the `getNextHouse` method and one new write statement in the `putToFile` method to handle the new information. The main application, `RealEstate`, would need to be changed to include the number of bathrooms in the interface—this requires straightforward additions throughout the program.

Chapter 4

2. Since the methods return different types, the compiler will report that the class is missing a method definition.

4. a. Yes
   b. No
   c. Yes
   d. No
   e. Yes
   f. No
   g. No
   h. Yes

5. a.
```
3 5 4
5
16
5
1
0
```

7. a.
```
{
  myStack.pop();
  secondElement = myStack.top();
  myStack.pop();
}
```

   c.
```
{
  ArrayStack tempStack = new ArrayStack();
  Object tempItem;

  while (!myStack.isEmpty())
  {
      tempItem = myStack.top();
      myStack.pop();
      tempStack.push(tempItem);
  }
  bottom = tempItem;

  // Restore stack
  while (!tempStack.isEmpty())
  {
    tempItem = tempStack.top();
    tempStack.pop();
    myStack.push(tempItem);
  }
}
```

8. a. `evenTop`: top for the stack of even values, initialized to −1 and incremented.
`oddTop`: top for the stack of odd values, initialized to 200 and decremented.

0	1	2	3	4	. . .	197	198	199
6	28	34	8		. . .		27	[129]

`evenTop`: 3
`oddTop`: 198

0	1	2	3	4	. . .	197	198	199
6	28	34	8		. . .		27	129

b.

```
public interface EvenOddStackInterface
{
  public void push(int item) throws StackOverflowException;
  // Effect:          If item is even, adds item to the top of the even
  //                  stack; otherwise, adds item to top of odd stack
  // Postconditions: If (this stack is full)
  //                    an unchecked exception that communicates
  //                    'push on stack full' is thrown
  //                  else
  //                    item is at the top of the appropriate stack

  public void popEven() throws StackUnderflowException;
  // Effect:          Removes top item from the even stack
  // Postconditions: If (the even stack is empty)
  //                   an unchecked exception that communicates
  //                   'pop on stack empty' is thrown
  //                 else
  //                   top element has been removed from the even stack

  public int topEven() throws StackUnderflowException;
  // Effect:          Returns the element on top of the even stack
  // Postconditions: If (the even stack is empty)
  //                    an unchecked exception that communicates
  //                   'top on stack empty' is thrown
  //                 else
  //                   return value = (top element of the even stack)
```

```
  public boolean isEvenEmpty();
  // Effect:         Determines whether the even stack is empty
  // Postcondition:  Return value = (the even stack is empty)

  public boolean isEvenFull();
  // Effect:         Determines whether the even stack is full
  // Postcondition:  Return value = (even stack is full)

  public void popOdd() throws StackUnderflowException;
  // Effect:         Removes top item from the odd stack
  // Postconditions: If (the odd stack is empty)
  //                   an unchecked exception that communicates
  //                   'pop on stack empty' is thrown
  //                 else
  //                   top element has been removed from the odd stack

  public int topOdd() throws StackUnderflowException;
  // Effect:         Returns the element on top of the odd stack
  // Postconditions: If (the odd stack is empty)
  //                    an unchecked exception that communicates
  //                   'top on stack empty' is thrown
  //                 else
  //                   return value = (top element of the odd stack)

  public boolean isOddEmpty();
  // Effect:         Determines whether the odd stack is empty
  // Postcondition:  Return value = (the odd stack is empty)

  public boolean isOddFull();
  // Effect:         Determines whether the odd stack is full
  // Postcondition:  Return value = (odd stack is full)
}
```

11. b.

```
public void pop(int count)
// Removes the top count elements from this stack
{
  for (int i = 0; i < count; i++)
  {
    if (!isEmpty())
    {
      stack[topIndex] = null;
      topIndex--;
    }
```

```
    else
      throw new StackUnderflowException("Pop attempted on an empty stack.");
  }
}
```

12. A number is either printed or put on the stack. Therefore, the numbers must follow a (possibly empty) increasing sequence followed by a (possibly empty) decreasing sequence.
    a. ii) False
    b. i) True
    c. The answers do not change. The implementation of an ADT does not (should not) change its functional behavior.

17. The `Set` class provides a collection that contains no duplicate elements.

19. b.
```
6
4
6
0
6  5  0
```

22. a. No
    b. Yes
    c. No
    d. No
    e. Yes
    f. Yes
    g. No
    h. Yes
    i. No

23. a. Repeatedly dequeue and save the items from the queue, counting as you go, until the queue is empty. Save the items in an array or another queue, and then enqueue them back into the original queue, in the same order in which they were removed.
    b. Simply return the value of the `numItems` instance variable.

25. a. 61
    b. 96

26. a. Too many operands when reach the second '+'
    b. Too many operands when reach '/'

Chapter 5

1. a. True—Array elements can be directly accessed through their indices.
   b. False—It depends on the implementation details; the key-based lists developed in this book are not random access.
   c. False—You must follow the links to access the items in the middle of the structure.
   d. False—For example, it could use dynamic arrays such as the Java Library's `ArrayList` class.
   e. True—In order to access an element in the middle of a stack you must "work through" the elements on top of it.
2. Typically, with an array-based stack, storage is allocated statically, ahead of time.
4. a. The use of `newNode` before its declaration would result in a syntax error.
   b. No functional difference.
   c. The third line would now use an incorrect value of `top`; the link to the rest of the stack would be lost.
6. a.
```
public int sizeIs()
// Returns the number of items on this stack
{
  return (topIndex + 1);
}
```
   b.
```
public int sizeIs()
// Returns the number of items on this stack
{
  int count = 0;
  StackNode temp;
  temp = top;
  while (temp != null)
  {
    count = count + 1;
    temp = temp.link;
  }
  return count;
}
```
   c. The constructor should set `size` to zero. The `push` method should increment `size` by 1 and the `pop` method should decrement `size` by 1. Could change `isEmpty` but it is not necessary.
   d. The `sizeIs` method for the `ArrayStack` class is O(1), since it just consists of a single statement. No other extra work is needed by the class. The `sizeIs` method for the `LinkedStack` class is O(*N*), since it must walk through the entire stack, counting

items, each time it is called. In Part (c) the `LinkedStack` class is revised, creating a little extra work every time `push` or `pop` are called, but allowing a O(1) `sizeIs` method, just like for the `ArrayStack` approach.

9. a. (201 references) × (4 bytes) + (20 ints × 2 bytes) = 844 bytes

d. (41 references) × (4 bytes) + (20 ints × 2 bytes) = 204 bytes

12. You cannot instantiate the `LinkedList` class since it is an abstract class. First you must extend it with a concrete class, such as `SortedLinkedList`, and then instantiate the concrete class.

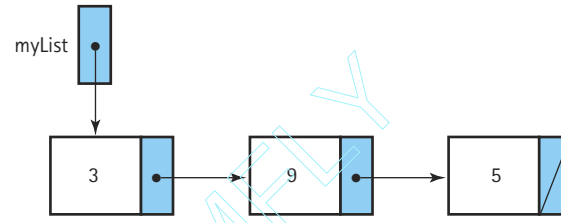
13. In the `List` class, `retrieve` and `delete` are abstract methods, but in the `LinkedList` class they are concrete classes. The `LinkedList` class defines and uses an inner class called `ListNode`.

15.
```
public boolean delete (Listable item)
// Deletes the element of this list whose key matches item's key
// and returns true. If no element on the list matches item it,
// returns false
{
  ListNode location = list;
  boolean found = false;

  if (numItems != 0)
  {
    // Locate node to be deleted
    if (item.compareTo(location.info) == 0)
    {
      list = list.next;                          // Delete first node
      found = true;
      numItems--;
    }
    else
    {
      while ((location.next != null)
            && (item.compareTo(location.next.info) != 0))
        location = location.next;
      if (location.next != null)
      {
        // Delete node at location.next
        location.next = location.next.next;
        found = true;
        numItems--;
      }
    }
  }
  return found;
}
```

17. a. 30
    b. 90
    c. 45

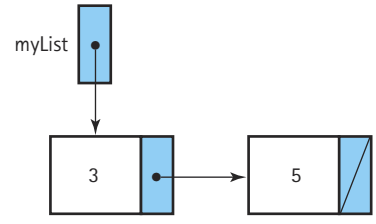
19. a. `list = list.next.next;`
    b. `list = null;`

20. a.



21.
```
int totalPrice(UnsortedLinkedList carList)
{
  Car currCar;
  int total = 0;
  int size = carList.lengthIs();
  carList.reset();
  for (int i = 0; i < size; i++)
  {
    currCar = (Car)carList.getNextItem();
    total = total + currCar.getPrice();
  }
  return total;
}
```

24. b.



26. a. It takes M steps to insert the items from `list1` into `list3`. For each of the N items in `list2` it takes M steps to check to see if they already were in `list1`, so that is $M * N$ steps. The actual insertion of the `list2` items is a little complicated to analyze. In the worst case we would have to traverse the entire `list3` for each insertion of an element of `list2`. Therefore, the first insertion from `list2` would take $M + 1$ steps, the second would take $M + 2$ steps, and so on, with the last taking $M + N$

steps. The total number of steps for this part of the process is therefore $(M + 1) + (M + 2) + \cdots + (M + N) = (M * N) + (1 + 2 + \cdots + N) = (M * N) + (N * (N - 1))/2$. Therefore, the total number of steps is

$$M + (M * N) + (M * N) + N^2/2 - N/2$$

In terms of Big-O, the complexity is $O(M * N + N^2)$.

b. A good algorithm would be to walk down each of the parameter lists, creating the new list as you go. At each step, you would copy the smaller of the elements that you are looking at on the parameter lists and then advance the list pointer of that list. When you reach the end of one of the lists, the remaining elements of the other list can just be copied to the new list. During this processing, if an element is discovered to be on both lists, it is copied to the new list only once. This algorithm advances one of the parameter list pointers at each "step"; therefore, it is $O(M + N)$, much faster than the algorithm described in part (a).

Chapter 6

1. The `isFull`, `lengthIs`, and `retrieve` methods are inherited, but Java does not allow inheritance of constructors. This forces the class designer to create his or her own constructor for each class, or use the default constructor. However, we can (and do) use the `super` operation to invoke the constructor of the superclass from within the constructor of the subclass.

3. The public methods `enqueue` and `dequeue` would have to be changed since they require manipulation of the extra references.

4.
```
public void printReverse()
// Pre:  List is not empty
// Post: List is printed in reverse order
{
  ListNode location = list;
  ArrayStack stack = new ArrayStack();
  Object item;

  do
  {
     stack.push(location.info);
     location = location.next;
  } while (location != list);

  while (!stack.isEmpty())
  {
    item = stack.top();
    stack.pop();
    System.out.println(item);
  }
}
```

6. a. Yes. The constructor must also set the new reference to `null`.
   b. No.
8. a. Instead of the references in the new node pointing to the previous node and the next node on the list, both references will point to the new node itself.
12. a. `C.next.info`
    b. `B.back.info`
    c. `A.next.next`
14. a. Doubly linked
    b. Circular
    c. List with header and trailer
15. Perhaps the school management programs use lists with trailers, and Mary knows that the "larger value" held in the trailer node's name variable is "Zz". But the new family will be inserted into the list after the trailer node. So, they must perform some corrective maintenance on the programs.
16. Initialization has O(N) complexity, where N is the size of the array of items; `getNode` and `freeNode` have O(1) complexity.
18. a.

nodes	.info	.next	.back
[0]		1	null
[1]		2	0
[2]		3	1
[3]		4	2
[4]		5	3
[5]		6	4
[6]		7	5
[7]		8	6
[8]		9	7
[9]		null	8

free	0
list	null

20. a. False
    b. False
    c. False
    d. True
    e. False
    f. False
    g. True

22. No. There is no "proper place" on the list. The list is not kept sorted and allows duplicates.

23. First of all, most of the changes would have to be made to the `SpecializedList` class, since that is the class the defines the nodes used for storage of the digits. If we assume that class can hold multiple bytes per node, the only changes needed to `LargeInt` would be to possibly change `addDigit` to handle more than one digit at a time and to change `toString` (revamp the construction of `largeIntString`).

26. a. The `greaterList` method assumes that a "longer" number is a larger number. So in the example it identified the number 003 as larger than the number 35. The subtraction algorithm changes the sign of the subtrahend and adds, so the program tries to add 35 and −003. The addition algorithm, when presented with two numbers of opposite signs, subtracts the absolute value of the smaller from the larger … so in the example it subtracts 35 from 003. Passing the parameters to the `subtractLists` method in the incorrect order causes calculation errors.

    b. One approach, which we call the Interface approach, would be to have the numbers entered by the user checked for leading zeros, and if they exist, display an error message in the interface and do not perform the calculation. Another approach, which we call the StripZeros approach, would be to have the leading zeros stripped off the numbers before they are stored in the lists. A third approach, called the RobustMath approach, is to rewrite all of the methods related to mathematics to handle situations where there are leading zeros.

    c. The Interface approach is easy to implement. We can test each operand for a single leading zero and, if it exists, throw an appropriate exception. The exception handler can display the error message. A drawback of this approach is that it puts the responsibility on the user. We must be careful to properly handle the case where the user just enters the digit 0 by itself.

    The StripZeros approach is a little more complicated. Instead of just checking for a single leading zero we must check for a string of leading zeros and "remove" them. Removing them just means not passing them to the `addDigit` method. A drawback here is that we are changing the information entered by the user. If later updates to the system require access to the exact information entered by the user, things could get complicated. Again, we must be careful to properly handle the case where the user just enters the digit 0 by itself.

    The RobustMath approach is the most complicated. The `greaterList` helper method is now more complicated. Additionally, the loop controls in the `addLists` and `subtractLists` methods depend on the "smaller" number being the "shorter" number. Since

this is no longer necessarily true, this program logic would need to be revised. If we ever upgrade the application to perform additional operations, they are also likely to be more complicated if this approach is used. The benefit of this approach is that the user is not bothered by any special rules about entering numbers and the user's entry data is not changed.

Chapter 7

1. a. The case for which a solution can be stated nonrecursively; it is a nonrecursive exit from the recursive solution.
   b. The general (or recursive) case is the case for which the solution is expressed in terms of a smaller version of itself.
   c. Indirect recursion is when the solution to one problem is expressed in terms of a second problem, which in turn is expressed in terms of a smaller version of the original problem.
2. See page 487.
5. a. −1
6. a. Yes, `num` must be zero or a negative number.
   b. No
   c. Yes. 0 is returned.
   d. Yes. −15 is returned.
7. a. The base case is when the second argument is equal to zero.

   The general case is when the second argument is not equal to zero.

   The second argument must be greater than or equal to zero.

   When the method is passed a *base* and a nonnegative *exponent*, it returns the value of the *base* raised to a power of the *exponent*. For example `power(4, 3)` would return $4^3 = 64$.

   c. The base case is when the argument, *n*, is less than 10.

   The general case is when the second argument, *n*, is greater than or equal to 10.

   There are no restrictions on the initial argument value (except that it must be of type `int`).

   When the method is passed a negative argument, it returns a −1; otherwise, it returns the number of digits in the positive integral argument. For example `recur(501)` would return 3 and `recur(12345)` would return 5.
8. a. This answer is incorrect. The value 0 is returned; the recursive case is never reached. This solution gets half credit, because it correctly calculates the base case (even if it doesn't reach it).

b. This solution correctly calculates the sum of squares but gets no credit because it is not a recursive solution.

c. This answer is correct and gets full credit.

d. This answer is functionally equivalent to (c); it just avoids the last recursive call (to an empty list) by returning the sum of the last squares as the base case. This answer runs into problems if the list is empty, but the specification states that the list is not empty. This answer gets full credit.

e. This solution is incorrect. The general case does not correctly calculate the sum of the squares. Quarter credit is given for using the correct control structure and for getting the base case correct.

9. a.
```
int fibonacci(int number)
{
  if (number <= 1)
    return number;
  else
    return fibonacci(number - 2) + fibonacci(number - 1);
}
```

b.
```
int fibonacci(int number)
{
  int current;
  int previous;
  int temp;

  if (number <= 1)
    return 1;
  else
  {
    previous = 0;
    current = 1;
    for (int count = 2; count <= number; count++)
    {
      temp = previous;
      previous = current;
      current = temp + previous;
    }
    return current;
  }
}
```

d. The recursive solution is extremely inefficient because many of the intermediate values are calculated more than once.

e. The following version, which uses an auxiliary recursive function, is more efficient. Note that the recursive parameters are used to keep track of the current and previous numbers, rather than recalculating them.

```
int fibonacci(int number)
{
  return fib(number, 1, 1);
}

int fib(int number, int previous, int current)
{
  if (number == 0)
    return previous;
  else
    return fib(number - 1, current, current + previous)
}
```

11. a. It is difficult to establish that the recursive calls satisfy Question 2, that they are moving toward the base case.

b. 16 recursive calls

3 recursive calls

17 recursive calls

13.
```
private void printList(ListNode listRef)
{
  if (listRef != null)
  {
    System.out.println(" " + listRef.info);
    printList(listRef.next);
  }
}
```

The `revPrint` method is a better use of recursion since the list can easily be printed in forward order by using a standard iterative list traversal.

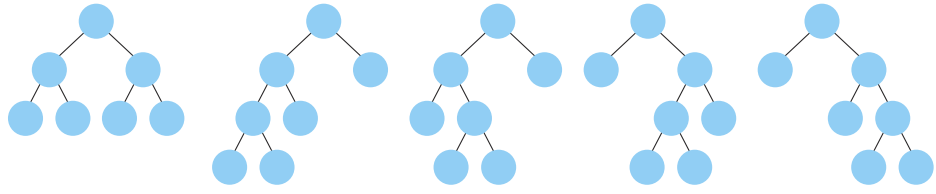
17. a. True—The return values on the runtime stack can take the place of local variables.

d. False—Recursive solutions are often less efficient in terms of computing time.

f. True—Otherwise you would recurse indefinitely and eventually use up all the run time stack space and bomb.

18. A stack.

Chapter 8

1. a. The level of a binary search tree determines the maximum number of comparisons that are required to find an element in the tree.
   b. 100
   c. 7
2. c. is the correct answer
7. 
8. a. Q, K, and M
   b. B, D, J, M, P, and N
   c. 8
   d. 16
   e. BDJKMNPQRTWY
   f. BJDNPMKRWYTQ
   g. QKDBJMPNTRYW
10. a. False
    b. True
    c. False
13. The elements used with our trees would need to support the `copy` operation, in addition to the `compareTo` operation. Since this operation is not really needed to implement the tree operations, requiring it places an unnecessary restriction on the kinds of elements that can be used with trees. Additionally, since the `Comparable` interface is a Java library interface, there are many types of elements that already implement it, that can be used with our trees. This benefit would not occur if we require `Listable` tree elements instead.
16.

```
int countLess(BinarySearchTree tree, Comparable maxValue)
// Effect: returns the number of nodes of tree that contain a value
//         less than or equal to maxValue
// Pre:    tree has been instantiated
```

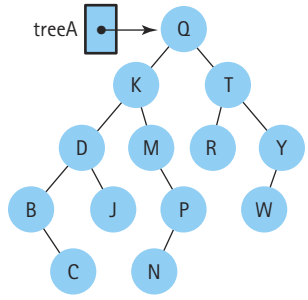
```
{
  int treeSize;
  int numNodes = 0;
  treeSize = tree.reset(BinarySearchTree.INORDER);
  for (int count = 1; count <= treeSize; count++)
  {
    if ((tree.getNextItem(BinarySearchTree.INORDER).compareTo(maxValue))
         <= 0)
      numNodes = numNodes + 1;
  }
  return numNodes;
}
```

19.

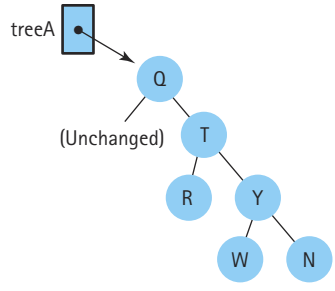
```
public int leafCount()
// Effect: returns number of leaf nodes in the tree
// Calls recursive method countLeaves to count the number
//  of leaf nodes
{
  return countLeaves(root);
}

private int countLeaves(BSTNode tree)
{
  if (tree == null)
    return 0;
  else if (tree.left == null) && (tree.right == null)
    return 1;
  else
    return (countLeaves(tree.left) + countLeaves(tree.right));
}
```

27.

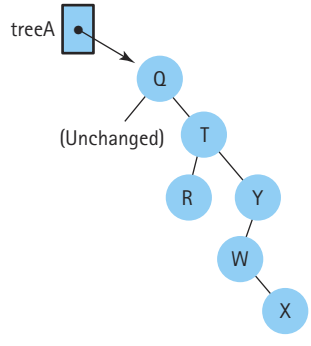
(a)


treeA
Q
K
T
D
M
R
Y
B
J
P
W
C
N

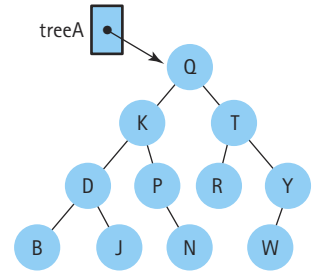
(b)


treeA
Q
(Unchanged)
T
R
Y
W
N

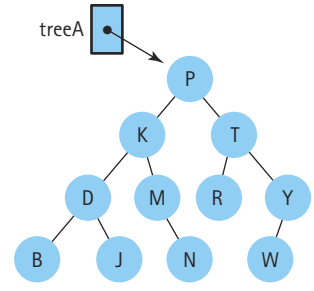
(c)


treeA
Q
(Unchanged)
T
R
Y
W
X

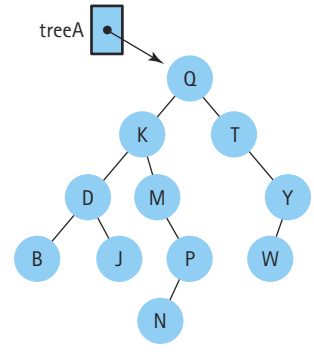
(d)


treeA
Q
K
T
D
P
R
Y
B
J
N
W

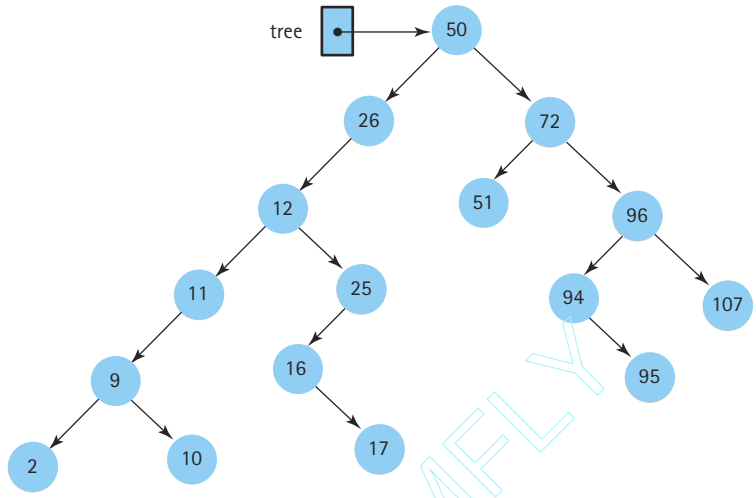
(e)


treeA
P
K
T
D
M
R
Y
B
J
N
W

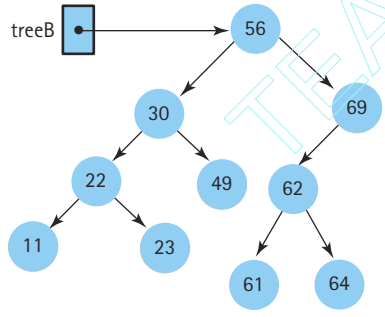
(f)


treeA
Q
K
T
D
M
Y
B
J
P
W
N

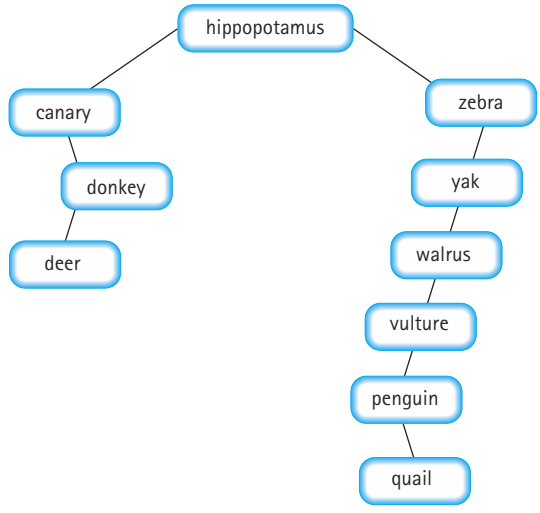
28.



30.



32. a.

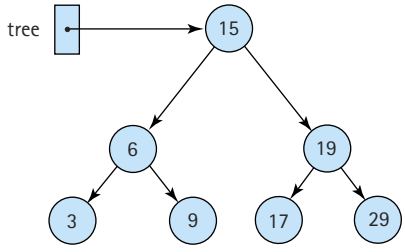


34. Either the value in node 5 or the value in node 6.

36. Preorder

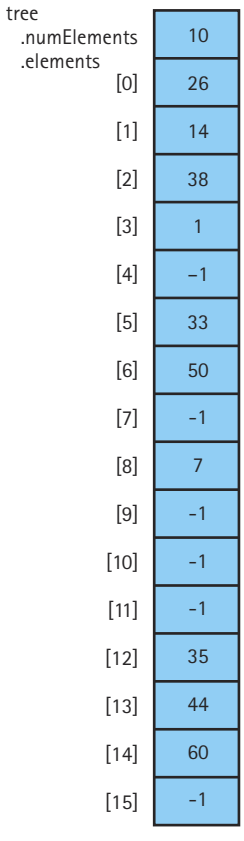
37. Elements inserted in random order:
    Sorted linked list: O(N)
    Binary search tree: O($\log_2 N$)

42. a.



45. a. Any negative value, for example a -1.

b.



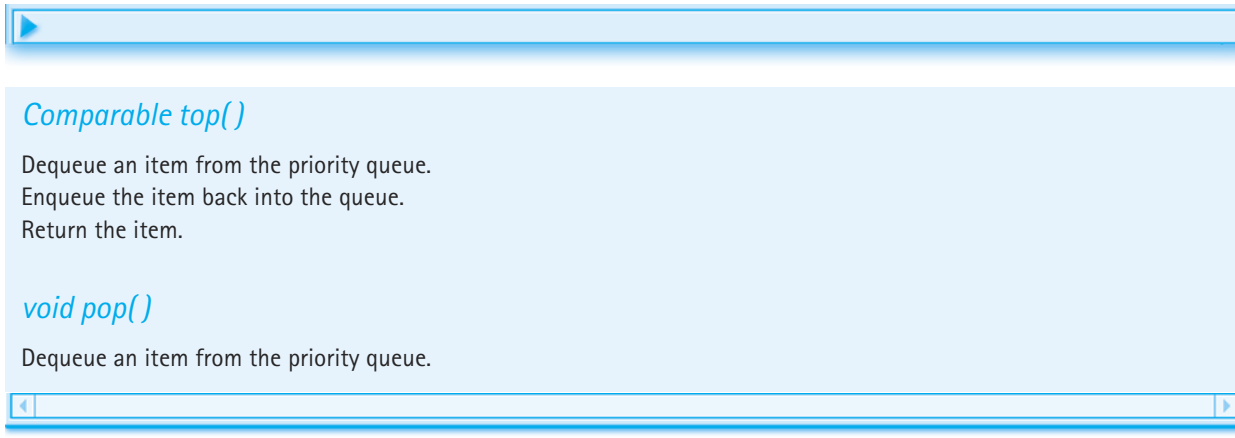
48. a. True—Its smallest child index would be 85, but the tree only contains 85 elements, in locations 0 .. 84.
    b. False—`treeNodes[41]` has two children: `treeNodes[83]` and `treeNodes[84]`.
    c. False—The left child of `treeNodes[12]` is `treeNodes[25]`.
    d. True—Drawing the corresponding tree helps verify this statement.
    e. False—The tree contains six full levels (containing 63 elements), with 22 additional elements on the seventh level.
49. It appears 9 times. See the figure on page 596.

Chapter 9

3. a.
```
private Comparable[] items;
private int numItems;
```
   b. The code is the same as for the `insert` method of the `SortedList` class as presented in Chapter 3, except we must reverse the relational operator so that the items are stored from largest to smallest.
   c.
```
public Comparable dequeue()
{
  Comparable item;
  item = items[0];
  for (int count = 1; count < numItems; count++)
    items[count - 1] = items[count];
  numItems--;
  return item;
}
```
4. a. The highest-priority element is the one with the largest time stamp.
   b.

void push(Comparable item)

Assign the next largest time stamp to the new item.
Enqueue the new item into the priority queue.



Comparable top( )

Dequeue an item from the priority queue.
Enqueue the item back into the queue.
Return the item.

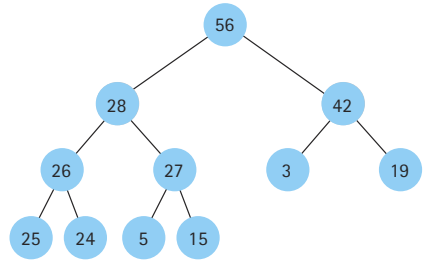
void pop( )

Dequeue an item from the priority queue.

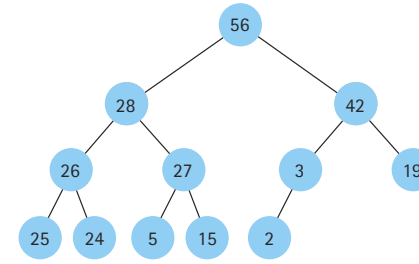
6. Tree d is the only heap. Trees a, c, and f do not satisfy the shape property of heaps. Trees b and e do not satisfy the order property of heaps.
7. Any tree whose root nodes contain the largest value, and whose other nodes are linearly linked through the left pointer member, each containing a smaller value than its parent, correctly answers this question.

11. a.

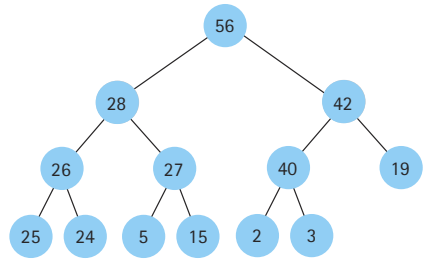
After pq.Enqueue (28)



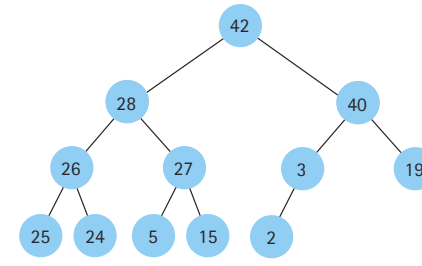
After pq.Enqueue (2)



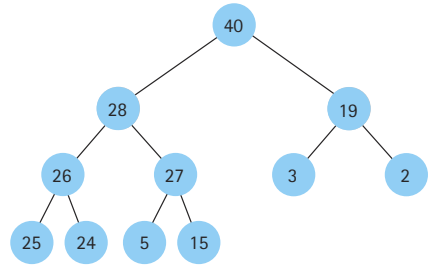
After pq.Enqueue (40)



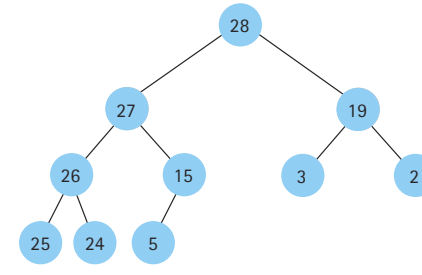
After pq.Dequeue (x)



After pq.Dequeue (y)

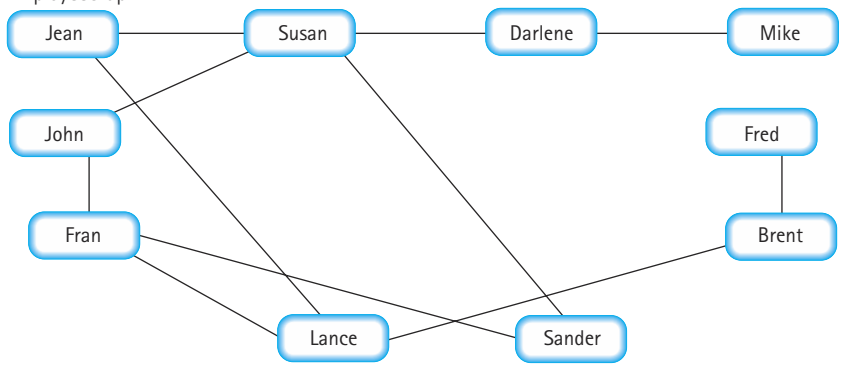


After pq.Dequeue (z)



b. $x = 56$, $y = 42$, $z = 40$

14. EmployeeGraph



15. EmployeeGraph

.numVertices

10

.vertices

[0]	Brent
[1]	Darlene
[2]	Fran
[3]	Fred
[4]	Jean
[5]	John
[6]	Lance
[7]	Mike
[8]	Sander
[9]	Susan

.edges

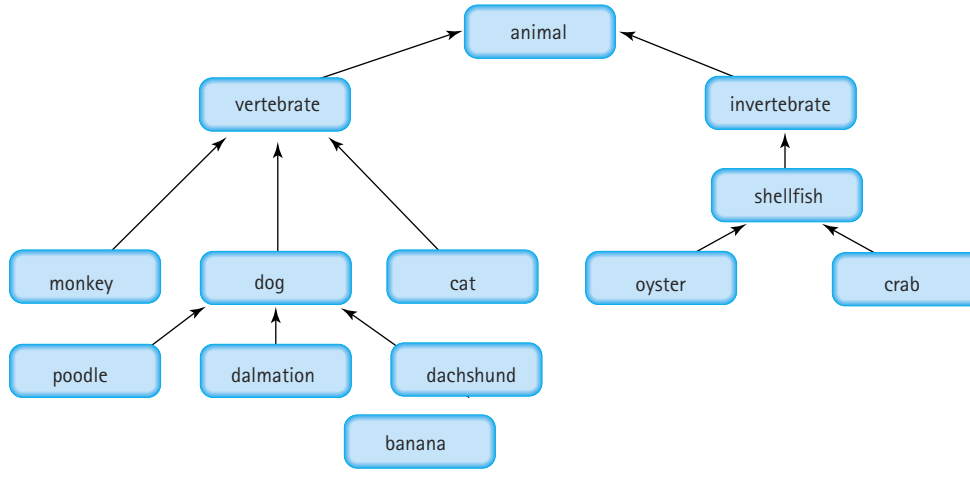
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
[0]	F	F	F	T	F	F	T	F	F	F
[1]	F	F	F	F	F	F	F	T	F	T
[2]	F	F	F	F	F	T	T	F	T	F
[3]	T	F	F	F	F	F	F	F	F	F
[4]	F	F	F	F	F	F	T	F	F	T
[5]	F	F	T	F	F	F	F	F	F	T
[6]	T	F	T	F	T	F	F	F	F	F
[7]	F	T	F	F	F	F	F	F	F	F
[8]	F	F	T	F	F	F	F	F	F	T
[9]	F	T	F	F	T	T	F	F	T	F

16. a. Susan, Jean, Lance

    b. Susan, Sander, Fran, Lance

17. "works with" is the best description of the relationship represented by the edges between vertices in EmployeeGraph, because it is an undirected graph. The other relationships listed have an order implicit in them.

18.



21. The correct answer is (b). For example, a dalmatian is an example of a dog.

22. V(StateGraph) = {Oregon, Alaska, Texas, Hawaii, Vermont, New York, California}
    E(StateGraph) = {(Alaska, Oregon), (Hawaii, Alaska), (Hawaii, Texas), (Texas, Hawaii), (Hawaii, California), (Hawaii, New York), (Texas, Vermont), (Vermont, California), (Vermont, Alaska)}

23. a. No

    b. Yes

    c. Hawaii and Texas

26. a.

```
public boolean edgeExists(Object vertex1, Object vertex2)
// Effect:          Return value = (vertex1, vertex2) is in the set of
//                  edges
// Preconditions:   vertex1 and vertex2 are in the set of vertices
```

    b.

```
{
  return (edges[indexIs(vertex1)][indexIs(vertex2)] != NULL_EDGE);
}
```

30. No. An inorder traversal of a binary search tree produces an ordered list of the tree elements. If such a list is fed to the `insert` method the resulting binary search tree will be completely skewed.

Chapter 10

1. a. Declare a static variable `numSwaps` in the `Sorts` class. Initialize it to 0. Also set it to 0 in the `initValues` method. Print out the `numSwaps` value in the `printValues` method. Finally, increment `numSwaps` by 1 in the `swap` method.
2. The correct answer is (c)
3. a. O(N)
   b. O(N)
4. None.
7. a. 4950
   b. 99
9. a. O(N^2)
   b. O(N)
   c. O(N^2)
   d. O(N)
11. The correct answer is (d).
15. Quick Sort is O(N^2) if the split algorithm repeatedly causes the array to be split into one element and the rest of the unfinished array. For example, if the split algorithm simply chooses the split value as the first value in the subarray and the list is already sorted, the result will be O(N^2).
16. Only (b) is true.
20. a. O($N\log_2 N$)
    b. O(N^2)
    c. O($N\log_2 N$)
21. Quick Sort and Selection Sort would take the longest; Insertion Sort and Short Bubble would take the shortest.
23. a. True. Using the array-based implementation of a binary tree, Heap Sort can be accomplished with constant extra space. However, Merge Sort requires at least enough space to hold information about all the outstanding merge jobs.
    b. False. Depending on the way the split value is selected, nearly sorted data is often a degenerate case for Quick Sort.
    c. True. Heap Sort takes essentially the same steps no matter what the order of the elements.

25. a. We could use the quadratic equation to solve the formula $n^2 - 4n + 1 > 0$ but it is just as easy to use a table to calculate the answer.

n	$f(n) = 4n$	$g(n) = n^2 + 1$
1	4	2
2	8	5
3	12	10
4	16	17
5	20	26

Therefore, when $n > 3$, $g(n) > f(n)$.

28. Programmer time refers to the amount of time it takes a programmer to generate a piece of software, including the time to design, code, and test it. If a programmer needs to finish a software project quickly, sometimes the programmer's time is a more critical efficiency consideration than how fast the resulting program runs on a computer. In this chapter, recursive sorting algorithms are cited as time-savers for the programmer, possibly at the expense of computing time.

29. a.
```
public int compareTo(Circle otherCircle)
{
  Circle other = (Circle)otherCircle;
  return (int)(this.radius - other.radius);
}
```

34. a. Quick Sort or Heap Sort

b. Quick Sort or Merge Sort

36. The correct answer is (a), O(1), since you already know the index.

38. a. 100

b. 100

c. 100

39. a. 50

b. 50

c. 50

40. The correct answer is (a).

43.

[0]	140
[1]	620
[2]	
[3]	
[4]	
[5]	145
[6]	66
[7]	47
[8]	87
[9]	126
[10]	90
[11]	285
[12]	467
[13]	153
[14]	393
[15]	395
[16]	566
[17]	177
[18]	735
[19]	

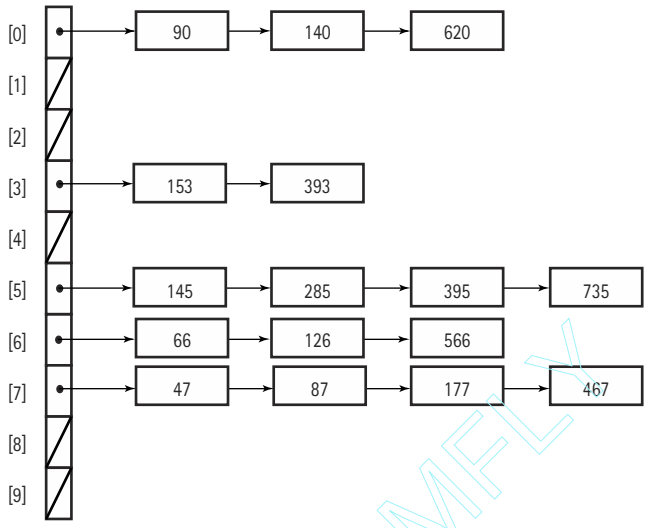
44.

[0]	140
[1]	
[2]	467
[3]	620
[4]	
[5]	145
[6]	66
[7]	47
[8]	285
[9]	126
[10]	87
[11]	
[12]	566
[13]	90
[14]	
[15]	395
[16]	153
[17]	177
[18]	735
[19]	393

45. HashTable

[0]	90	140	620
[1]			
[2]			
[3]	153	393	
[4]			
[5]	145	285	395
[6]	66	126	566
[7]	47	87	177
[8]	467	735	
[9]			

46.



Index

Note: Italicized page locators refer to tables/figures.

E

M