



Data
Abstraction
& Problem
Solving
with

JAVATM

3rd Edition Walls & Mirrors

JANET J. PRICHARD | FRANK M. CARRANO

Summary of Java Statements

assignment

```
variable = expression;
```

compound

```
{ statement1  
statement2  
:  
statementn  
}
```

if

```
if (expression)  
statement
```

if-else

```
if (expression)  
statement1  
else  
statement2
```

switch

```
switch (expression)  
{ case constant1:  
    statement1  
    break;  
:  
case constantn:  
    statementn  
    break;  
  
default:  
    statement  
}
```

for

```
for (initialize; test; update)  
statement
```

```
for (variable:collection)  
statement
```

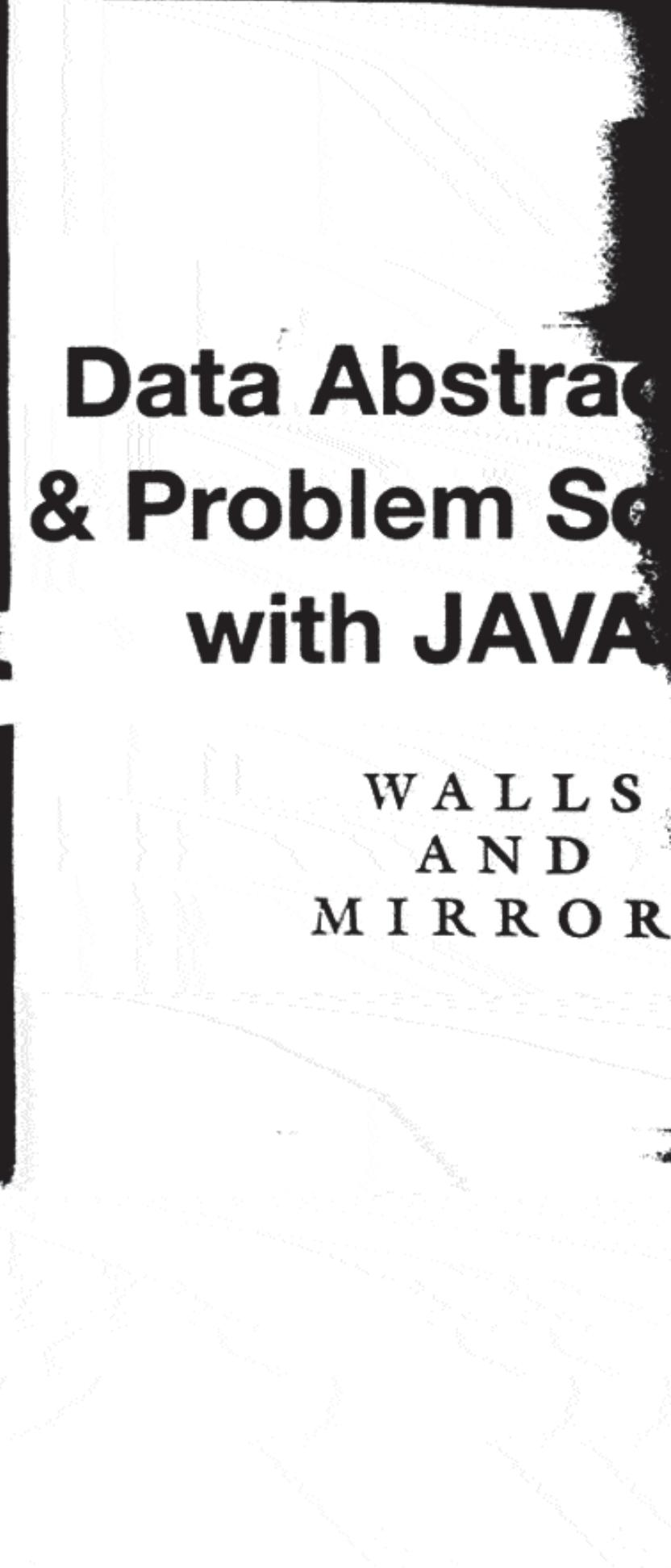
```
for (variable:array)  
statement
```

while

```
while (expression)  
statement
```

do

```
do  
statement  
while (expression);
```



Data Abstraction & Problem Solving with JAVA

**WALLS
AND
MIRRORS**

Data Abstraction & Problem Solving with JAVA

WALLS
AND
MIRRORS

3rd Edition

Janet J. Prichard
Bryant University

Frank Carrano
University of Rhode Island

Addison-Wesley

Boston Columbus Indianapolis New York San Francisco Upper Saddle River
Amsterdam Cape Town Dubai London Madrid Milan Munich Paris Montreal Toronto
Delhi Mexico City Sao Paulo Sydney Hong Kong Seoul Singapore Taipei Tokyo

Editorial Director: Marcia Horton
Editor-in-Chief: Michael Hirsch
Editorial Assistant: Stephanie Sellinger
Marketing Manager: Yezan Alayan
Marketing Assistant: Kathryn Ferranti
Vice President, Production: Vince O'Brien
Managing Editor: Jeff Holcomb
Senior Production Project Manager: Marilyn Lloyd

Operations Specialist: Lisa McDowell
Text Designer: Sandra Rigney
Creative Director: Jayne Conte
Cover Designer: Suzanne Duda
Cover Image: Getty Images/Steve Wall
Full-Service Vendor: GEX Publishing Services
Printer/Cover Printer/Binder: STP Courier
Stoughton

Credits and acknowledgments borrowed from other sources and reproduced, with permission, in this textbook appear on appropriate page within text.

Copyright © 2011, 2006, 2004 Pearson Education, Inc., publishing as Addison-Wesley. All rights reserved.
Manufactured in the United States of America. This publication is protected by Copyright, and permission should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission(s) to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, 501 Boylston Street, Suite 900, Boston, Massachusetts 02116.

Many of the designations by manufacturers and seller to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed in initial caps or all caps.

Library of Congress Cataloging-in-Publication Data is available upon request

10 9 8 7 6 5 4 3 2 1—CRS—14 13 12 11 10

Addison-Wesley
is an imprint of

PEARSON

ISBN 10: 0-13-212230-8
ISBN 13: 978-0-13-212230-6

Brief Contents

PART ONE

Problem-Solving Techniques 1

- 1 Review of Java Fundamentals 3
- 2 Principles of Programming and Software Engineering 81
- 3 Recursion: The Mirrors 137
- 4 Data Abstraction: The Walls 197
- 5 Linked Lists 241

PART TWO

Problem Solving with Abstract Data Types 313

- 6 Recursion as a Problem-Solving Technique 315
- 7 Stacks 351
- 8 Queues 409
- 9 Advanced Java Topics 455
- 10 Algorithm Efficiency and Sorting 505
- 11 Trees 561
- 12 Tables and Priority Queues 643
- 13 Advanced Implementations of Tables 699
- 14 Graphs 777
- 15 External Methods 823

APPENDICES

- A A Comparison of Java to C++ 863
- B Unicode Character Codes (ASCII Subset) 867
- C Java Resources 868
- D Mathematical Induction 870
- Glossary 877
- Self-Test Answers 897
- Index 921

Contents

Preface xv
Chapter Dependency Chart xviii

PART ONE	
Problem-Solving Techniques	1
1 Review of Java Fundamentals	3
1.1 Language Basics	4
Comments	4
Identifiers and Keywords	4
Variables	4
Primitive Data Types	5
References	6
Literal Constants	6
Named Constants	7
Assignments and Expressions	8
Arrays	11
1.2 Selection Statements	14
The <i>if</i> Statement	15
The <i>switch</i> Statement	16
1.3 Iteration Statements	17
The <i>while</i> Statement	17
The <i>for</i> Statement	18
The <i>do</i> Statement	21
1.4 Program Structure	21
Packages	22
Classes	23
Data Fields	24
Methods	26
How to Access Members of an Object	30
Class Inheritance	30

1.5	Useful Java Classes	32
	The Object Class	32
	The Array Class	34
	String Classes	35
1.6	Java Exceptions	40
	Catching Exceptions	40
	Throwing Exceptions	47
1.7	Text Input and Output	49
	Input	49
	Output	51
	The Console Class	54
1.8	File Input and Output	56
	Text Files	58
	Object Serialization	66
	Summary	69
	Cautions	72
	Self-Test Exercises	72
	Exercises	73
	Programming Problems	78

2 Principles of Programming and Software Engineering

81

2.1	Problem Solving and Software Engineering	82
	What Is Problem Solving?	82
	The Life Cycle of Software	83
	What Is a Good Solution?	93
2.2	Achieving an Object-Oriented Design	95
	Abstraction and Information Hiding	96
	Object-Oriented Design	98
	Functional Decomposition	100
	General Design Guidelines	101
	Modeling Object-Oriented Designs Using UML	102
	Advantages of an Object-Oriented Approach	106
2.3	A Summary of Key Issues in Programming	107
	Modularity	107
	Modifiability	109
	Ease of Use	111
	Fail-Safe Programming	112
	Style	118
	Debugging	122
	Summary	125
	Cautions	126
	Self-Test Exercises	126
	Exercises	127
	Programming Problems	132

3 Recursion: The Mirrors

137

3.1	Recursive Solutions	138
	A Recursive Valued Method: The Factorial of n	141
	A Recursive void Method: Writing a String Backward	148

3.2	Counting Things	159
	Multiplying Rabbits (The Fibonacci Sequence)	159
	Organizing a Parade	161
	Mr. Spock's Dilemma (Choosing k out of n Things)	164
3.3	Searching an Array	166
	Finding the Largest Item in an Array	167
	Binary Search	168
	Finding the k^{th} Smallest Item in an Array	172
3.4	Organizing Data	176
	The Towers of Hanoi	176
3.5	Recursion and Efficiency	180
	Summary	187
	Cautions	187
	Self-Test Exercises	188
	Exercises	189
	Programming Problems	195

4 Data Abstraction: The Walls 197

4.1	Abstract Data Types	198
4.2	Specifying ADTs	203
	The ADT List	204
	The ADT Sorted List	209
	Designing an ADT	211
	Axioms (Optional)	215
4.3	Implementing ADTs	218
	Java Classes Revisited	219
	Java Interfaces	221
	Java Packages	224
	An Array-Based Implementation of the ADT List	226
	Summary	233
	Cautions	233
	Self-Test Exercises	234
	Exercises	235
	Programming Problems	238

5 Linked Lists 241

5.1	Preliminaries	242
	Object References	242
	Resizeable Arrays	248
	Reference-Based Linked Lists	249
5.2	Programming with Linked Lists	253
	Displaying the Contents of a Linked List	253
	Deleting a Specified Node from a Linked List	255
	Inserting a Node into a Specified Position of a Linked List	258
	A Reference-Based Implementation of the ADT List	264
	Comparing Array-Based and Reference-Based Implementations	268
	Passing a Linked List to a Method	271
	Processing Linked Lists Recursively	271

Contents

5.3	Variations of the Linked List	277
	Tail References	277
	Circular Linked Lists	278
	Dummy Head Nodes	280
	Doubly Linked Lists	280
5.4	Application: Maintaining an Inventory	284
5.5	The Java Collections Framework	290
	Generics	291
	Iterators	292
	The Java Collection's Framework <i>List</i> Interface	295
	Summary	298
	Cautions	300
	Self-Test Exercises	301
	Exercises	303
	Programming Problems	307

PART TWO **Problem Solving with Abstract** **Data Types** **313**

6 Recursion as a Problem-Solving Technique **315**

6.1	Backtracking	316
	The Eight Queens Problem	316
6.2	Defining Languages	321
	The Basics of Grammars	322
	Two Simple Languages	323
	Algebraic Expressions	326
6.3	The Relationship Between Recursion and Mathematical Induction	336
	The Correctness of the Recursive Factorial Method	336
	The Cost of Towers of Hanoi	337
	Summary	339
	Cautions	339
	Self-Test Exercises	340
	Exercises	340
	Programming Problems	344

7 Stacks **351**

7.1	The Abstract Data Type Stack	352
	Developing an ADT During the Design of a Solution	352
7.2	Simple Applications of the ADT Stack	358
	Checking for Balanced Braces	358
	Recognizing Strings in a Language	362
7.3	Implementations of the ADT Stack	363
	An Array-Based Implementation of the ADT Stack	365
	A Reference-Based Implementation of the ADT Stack	367
	An Implementation That Uses the ADT List	369
	Comparing Implementations	371
	The Java Collections Framework Class <i>Stack</i>	371
7.4	Application: Algebraic Expressions	373
	Evaluating Postfix Expressions	373
	Converting Infix Expressions to Equivalent Postfix Expressions	375

7.5 Application: A Search Problem	378
A Nonrecursive Solution That Uses a Stack	380
A Recursive Solution	388
7.6 The Relationship Between Stacks and Recursion	391
Summary	393
Cautions	393
Self-Test Exercises	394
Exercises	395
Programming Problems	400
8 Queues	409
8.1 The Abstract Data Type Queue	410
8.2 Simple Applications of the ADT Queue	412
Reading a String of Characters	412
Recognizing Palindromes	413
8.3 Implementations of the ADT Queue	414
A Reference-Based Implementation	416
An Array-Based Implementation	419
An Implementation That Uses the ADT List	425
The JCF Interfaces <i>Queue</i> and <i>Deque</i>	426
Comparing Implementations	432
8.4 A Summary of Position-Oriented ADTs	433
8.5 Application: Simulation	434
Summary	444
Cautions	445
Self-Test Exercises	445
Exercises	446
Programming Problems	450
9 Advanced Java Topics	455
9.1 Inheritance Revisited	456
Java Access Modifiers	462
Is-a and Has-a Relationships	464
9.2 Dynamic Binding and Abstract Classes	466
Abstract Classes	469
Java Interfaces Revisited	474
9.3 Java Generics	475
Generic Classes	475
Generic Wildcards	477
Generic Classes and Inheritance	478
Generic Implementation of the Class List	481
Generic Methods	483
9.4 The ADTs List and Sorted List Revisited	484
Implementations of the ADT Sorted List That Use the ADT List	485
9.5 Iterators	489
Summary	493
Cautions	494
Self-Test Exercises	494
Exercises	495
Programming Problems	500

10 Algorithm Efficiency and Sorting	505
10.1 Measuring the Efficiency of Algorithms 506	
The Execution Time of Algorithms 507	
Algorithm Growth Rates 509	
Order-of-Magnitude Analysis and Big O Notation 509	
Keeping Your Perspective 515	
The Efficiency of Searching Algorithms 517	
10.2 Sorting Algorithms and Their Efficiency 518	
Selection Sort 519	
Bubble Sort 523	
Insertion Sort 525	
Mergesort 527	
Quicksort 533	
Radix Sort 545	
A Comparison of Sorting Algorithms 547	
The Java Collections Framework Sort Algorithm 548	
Summary 552 Cautions 553 Self-Test Exercises 553	
Exercises 554 Programming Problems 558	
11 Trees	561
11.1 Terminology 562	
11.2 The ADT Binary Tree 570	
Basic Operations of the ADT Binary Tree 570	
General Operations of the ADT Binary Tree 571	
Traversals of a Binary Tree 574	
Possible Representations of a Binary Tree 577	
A Reference-Based Implementation of the ADT Binary Tree 581	
Tree Traversals Using an Iterator 586	
11.3 The ADT Binary Search Tree 594	
Algorithms for the Operations of the ADT Binary Search Tree 600	
A Reference-Based Implementation	
of the ADT Binary Search Tree 615	
The Efficiency of Binary Search Tree Operations 619	
Treesort 624	
Saving a Binary Search Tree in a File 625	
The JCF Binary Search Algorithm 628	
11.4 General Trees 629	
Summary 631 Cautions 632 Self-Test Exercises 632	
Exercises 634 Programming Problems 640	
12 Tables and Priority Queues	643
12.1 The ADT Table 644	
Selecting an Implementation 651	
A Sorted Array-Based Implementation of the ADT Table 658	
A Binary Search Tree Implementation of the ADT Table 661	

12.2	The ADT Priority Queue: A Variation of the ADT Table	663
Heaps	667	
A Heap Implementation of the ADT Priority Queue	676	
Heapsort	678	
12.3	Tables and Priority Queues in the JCF	681
The JCF <i>Map</i> Interface	681	
The JCF <i>Set</i> Interface	685	
The JCF <i>PriorityQueue</i> Class	689	
Summary	691	Cautions 692 Self-Test Exercises 692
Exercises	693	Programming Problems 696
13	Advanced Implementations of Tables	699
13.1	Balanced Search Trees	700
2-3 Trees	701	
2-3-4 Trees	721	
Red-Black Trees	728	
AVL Trees	731	
13.2	Hashing	737
Hash Functions	741	
Resolving Collisions	743	
The Efficiency of Hashing	752	
What Constitutes a Good Hash Function?	755	
Table Traversal: An Inefficient Operation under Hashing	757	
The JCF <i>Hashtable</i> and <i>TreeMap</i> Classes	758	
The <i>Hashtable</i> Class	758	
The <i>TreeMap</i> Class	761	
13.3	Data with Multiple Organizations	764
Summary	769	Cautions 770 Self-Test Exercises 771
Exercises	771	Programming Problems 774
14	Graphs	777
14.1	Terminology	778
14.2	Graphs as ADTs	781
Implementing Graphs	782	
Implementing a Graph Class Using the JCF	785	
14.3	Graph Traversals	788
Depth-First Search	790	
Breadth-First Search	791	
Implementing a BFS Iterator Class Using the JCF	793	
14.4	Applications of Graphs	796
Topological Sorting	796	
Spanning Trees	799	
Minimum Spanning Trees	804	

Contents

Shortest Paths	807		
Circuits	811		
Some Difficult Problems	814		
Summary	815	Cautions	816
Exercises	817	Self-Test Exercises	816
		Programming Problems	820
15 External Methods			823
15.1 A Look at External Storage	824		
15.2 Sorting Data in an External File	827		
15.3 External Tables	835		
Indexing an External File	837		
External Hashing	841		
B-Trees	845		
Traversals	855		
Multiple Indexing	857		
Summary	858	Cautions	859
Exercises	859	Self-Test Exercises	859
		Programming Problems	862
A A Comparison of Java to C++	863		
B Unicode Character Codes (ASCII Subset)	867		
C Java Resources	868		
Java Web Sites	868		
Using Java SE 6	868		
Integrated Development Environments (IDEs)	869		
D Mathematical Induction	870		
Example 1	870		
Example 2	871		
Example 3	872		
Example 4	873		
Example 5	873		
Self-Test Exercises	874	Exercises	874
Glossary	877		
Self-Test Answers	897		
Index	921		

Preface

Welcome to the third edition of *Data Abstraction and Problem Solving with Java: Walls and Mirrors*. Java is a popular language for beginning computer science courses. It is particularly suitable to teaching data abstraction in an object-oriented way.

This book is based on the original *Intermediate Problem Solving and Data Structures: Walls and Mirrors* by Paul Helman and Robert Veroff (© 1986 by Benjamin Cummings Publishing Company, Inc.). This work builds on their organizational framework and overall perspective and includes technical and textual content, examples, figures, and exercises derived from the original work. Professors Helman and Veroff introduced two powerful analogies, walls and mirrors, that have made it easier for us to teach—and to learn—computer science.

With its focus on data abstraction and other problem-solving tools, this book is designed for a second course in computer science. In recognition of the dynamic nature of the discipline and the great diversity in undergraduate computer science curricula, this book includes comprehensive coverage of enough topics to make it appropriate for other courses as well. For example, you can use this book in courses such as introductory data structures or advanced programming and problem solving. The goal remains to give students a superior foundation in data abstraction, object-oriented programming, and other modern problem-solving techniques.

New in this edition

Uses Java 6: This edition has been thoroughly revised to be compatible with the latest release of Java, known as Java 6. All code has been completely revised to be Java 6 compliant. Generics are also an important part of Java 6, and this material is discussed in depth in Chapter 9, and then used throughout the remainder of the collections in the text.

Enhanced Early Review of Java: We have increased the amount of coverage of the Java language in the first chapter of the book to help students make the transition from their introduction to Java course to this course. Chapter 1 provides a

concise review of important Java material, including brief discussions of constructors, object equality, inheritance, and the `ArrayList` class. A discussion of the `Console` class from Java 6 was also added to Chapter 1. Chapter 9 focuses on advanced Java techniques, and includes an enhanced discussion of how to create an iterator class.

Linked List: The node class for linked lists has been simplified. The implementation now assumes the node class is package access only, and the other classes in the same package have direct access to the data within a node. Students are asked to explore the implications of making the data private in a node as an exercise.

Updates the Use of the Java Collections Framework: The Java Collections Framework is discussed throughout the text, with a section added to show the JFC classes that parallel those presented in the text. The `Deque` class, added in Java 6, is presented in Chapter 8.

Other enhancements: Additional changes aimed at improving the overall usability of the text include new exercises and a new cleaner design that enhances the book's readability.

TO THE STUDENT

Thousands of students before you have read and learned from *Walls and Mirrors*. The walls and mirrors in the title represent two fundamental problem-solving techniques that appear throughout the book. Data abstraction isolates and hides the implementation details of a module from the rest of the program, much as a wall can isolate and hide you from your neighbor. Recursion is a repetitive technique that solves a problem by solving smaller problems of exactly the same type, much as mirror images that grow smaller with each reflection.

This book was written with you in mind. As former college students, and as educators who are constantly learning, we appreciate the importance of a clear presentation. Our goal is to make this book as understandable as possible. To help you learn and to review for exams, we have included such learning aids as margin notes, chapter summaries, self-test exercises with answers, and a glossary. As a help during programming, you will find Java reference materials in Chapter 1, and inside the covers. You should review the list of this book's features given later in this preface under the section "Pedagogical Features."

The presentation makes some basic assumptions about your knowledge of Java as reviewed in Chapter 1. Some of you may need to review this language or learn it for the first time by consulting this chapter. Others will find that they already know most of the constructs presented in Chapter 1. You will need to know about the selection statements `if` and `switch`; the iteration statements `for`, `while`, and `do`; classes, methods, and arguments; arrays; strings; and files. In addition to the material in Chapter 1, this book discusses advanced Java topics such as generics and iterators in Chapter 9. We assume no experience with recursive methods, which are included in Chapters 3 and 6.

All of the Java source code that appears in this book is available for your use. Later in this preface, the description of supplementary materials explains how to obtain these files. See page xxi—Supplemental Materials—for instructions on how to access these files.

TO THE INSTRUCTOR

This edition of *Walls and Mirrors* uses Java 6 to enhance its emphasis on data abstraction and data structures. The book carefully accounts for the strengths and weaknesses of the Java language and remains committed to a pedagogical approach that makes the material accessible to students at the introductory level.

Prerequisites

We assume that readers either know the fundamentals of Java or know another language and have an instructor who will help them make the transition to Java. By using Chapter 1, students without a strong Java background can quickly pick up what they need to know to be successful in the course. In addition, the book formally discusses Java classes. Included are the basic concepts of a class, inheritance, polymorphism, interfaces, and packages. Although the book provides an introduction to these topics in connection with the implementations of abstract data types (ADTs) as classes, the emphasis of the book remains on the ADTs, not on Java. The material is presented in the context of object-based programming, but it assumes that future courses will cover object-oriented design and software engineering in detail, so that the focus can remain on data abstraction. We do, however, introduce the Unified Modeling Language (UML) as a design tool.

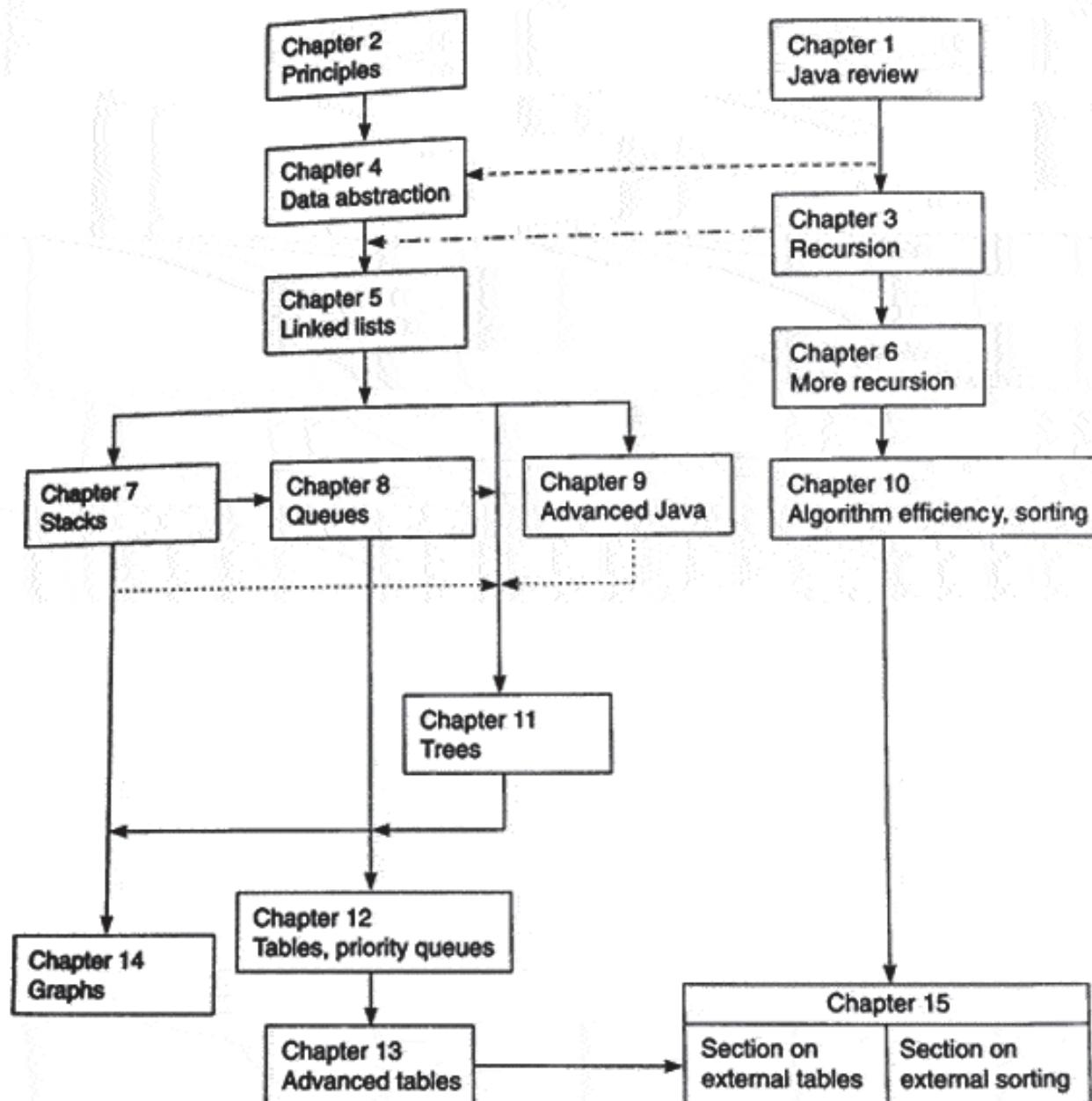
Organization

The chapters in this book are organized into two parts. In most cases, Chapters 1 through 11 will form the core of a one-semester course. Chapters 1 or 2 might be review material for your students. The coverage given to Chapters 11 through 15 will depend on the role the course plays in your curriculum.

Flexibility

The extensive coverage of this book should provide you with the material that you want for your course. You can select the topics you desire and present them in an order that fits your schedule. A chapter dependency chart follows, and shows which chapters should be covered before a given chapter can be taught.

Part I: Problem-Solving Techniques. The first two chapters in Part I resemble an extension of an introductory course in that their emphasis is on major issues in programming and software engineering. Chapter 3 introduces recursion for those students who have had little exposure to this important topic. The ability to think recursively is one of the most useful skills that a



— — — — — Dependency by one section of chapter

----- Dependency that you can ignore

---- ----- Knowledge of Java helpful to begin these chapters

computer scientist can possess and is often of great value in helping one to understand better the nature of a problem. Recursion is discussed extensively in this chapter and again in Chapter 6 and is used throughout the book. Included examples range from simple recursive definitions to recursive algorithms for language recognition, searching, and sorting.

Chapter 4 discusses data abstraction and abstract data types (ADTs) in detail. After a discussion of the specification and use of an ADT, the chapter discusses Java classes, interfaces, and packages, and uses them to implement ADTs. Chapter 5 presents additional implementation tools in its discussion of Java reference variables and linked lists.

You can choose among the topics in Part 1 according to the background of your students and cover these topics in several orders.

Part 2: Problem Solving with Abstract Data Types. Part 2 continues the use of data abstraction as a problem-solving technique. Basic abstract data types such as the stack, queue, binary tree, binary search tree, table, heap, and priority queue are first specified and then implemented as classes. The ADTs are used in examples and their implementations are compared.

Chapter 9 extends the treatment of Java classes by covering inheritance, the relationships among classes, generics, and iterators. Chapter 10 formalizes the earlier discussions of an algorithm's efficiency by introducing order-of-magnitude analysis and Big O notation. The chapter examines the efficiency of several searching and sorting algorithms, including the recursive mergesort and quicksort.

Part 2 also includes advanced topics—such as balanced search trees (2-3, 2-3-4, red-black, and AVL trees) and hashing—that are examined as table implementations. These implementations are analyzed to determine the table operations that each supports best.

Finally, data storage in external direct access files is considered. Mergesort is modified to sort such data, and external hashing and B-tree indexes are used to search it. These searching algorithms are generalizations of the internal hashing schemes and 2-3 trees already developed.

In Part 1, you can choose among topics according to your students' background. Three of the chapters in this part provide an extensive introduction to data abstraction and recursion. Both topics are important, and there are various opinions about which should be taught first. Although in this book a chapter on recursion both precedes and follows the chapter on data abstraction, you can simply rearrange this order.

Part 2 treats topics that you can also cover in a flexible order. For example, you can cover all or parts of Chapter 9 on advanced Java topics either before or after you cover stacks (Chapter 7). You can cover algorithm efficiency and sorting (Chapter 10) any time after Chapter 6. You can introduce trees before queues or graphs before tables, or cover hashing, balanced search trees, or priority queues any time after tables and in any order. You also can cover external methods (Chapter 15) earlier in the course. For example, you can cover external sorting after you cover mergesort in Chapter 10.

Data Abstraction

The design and use of abstract data types (ADTs) permeate this book's problem-solving approach. Several examples demonstrate how to design an ADT as part of the overall design of a solution. All ADTs are first specified—in both English and pseudocode—and then used in simple applications before implementation issues are considered. The distinction between an ADT and the data structure that implements it remains in the forefront throughout the discussion. The book explains both encapsulation and Java classes early. Students see how Java classes hide an implementation's data structure from the client of the ADT. Abstract data types such as lists, stacks, queues, trees, tables, heaps, and priority queues form the basis of our discussions.

Problem Solving

This book helps students learn to integrate problem-solving and programming abilities by emphasizing both the thought processes and the techniques that computer scientists use. Learning how a computer scientist develops, analyzes, and implements a solution is just as important as learning the mechanics of the algorithm; a cookbook approach to the material is insufficient.

The presentation includes analytical techniques for the development of solutions within the context of example problems. Abstraction, the successive refinement of both algorithms and data structures, and recursion are used to design solutions to problems throughout the book.

Java references and linked list processing are introduced early and used in building data structures. The book also introduces at an elementary level the order-of-magnitude analysis of algorithms. This approach allows the consideration—first at an informal level, and then more quantitatively—of the advantages and disadvantages of array-based and reference-based data structures. An emphasis on the trade-offs among potential solutions and implementations is a central problem-solving theme.

Finally, programming style, documentation including preconditions and postconditions, debugging aids, and loop invariants are important parts of the problem-solving methodology used to implement and verify solutions. These topics are covered throughout the book.

Applications

Classic application areas arise in the context of the major topics of this book. For example, the binary search, quicksort, and mergesort algorithms provide important applications of recursion and introduce order-of-magnitude analysis. Such topics as balanced search trees, hashing, and file indexing continue the discussion of searching. Searching and sorting are considered again in the context of external files.

Algorithms for recognizing and evaluating algebraic expressions are first introduced in the context of recursion and are considered again later as an

application of stacks. Other applications include, for example, the Eight Queens problem as an example of backtracking, event-driven simulation as an application of queues, and graph searching and traversals as other important applications of stacks and queues.

Pedagogical Features

The pedagogical features and organization of this book were carefully designed to facilitate learning and to allow instructors to tailor the material easily to a particular course. This book contains the following features that help students not only during their first reading of the material, but also during subsequent review:

- Chapter outlines and previews
- Key Concepts boxes
- Margin notes
- Chapter summaries
- Cautionary warnings about common errors and misconceptions
- Self-test exercises with answers
- Chapter exercises and programming problems. The most challenging exercises are labeled with asterisks. Answers to the exercises appear in the *Instructor's Resource Manual*.
- Specifications for all major ADTs in both English and pseudocode
- Java class definitions for all major ADTs
- Examples that illustrate the role of ADTs in the problem-solving process
- Appendixes, including a review of Java
- Glossary of terms

SUPPLEMENTAL MATERIALS

The following supplementary materials are available online to all readers of this book at www.pearsonhighered.com/cssupport.

- *Source code* of all the Java classes, methods, and programs that appear in the book
- *Errata*: We have tried not to make mistakes, but mistakes are inevitable. A list of detected errors is available and updated as necessary. You are invited to contribute your finds.

The following instructor supplements are only available to qualified instructors. Please visit Addison-Wesley's Instructor Resource Center (www.pearsonhighered.com/irc) or contact your local Addison-Wesley Sales Representative to access them.

- *Instructor's Guide with Solutions:* This manual contains teaching hints, sample syllabi, and solutions to all the end-of-chapter exercises in the book.
- *Test Bank:* A collection of multiple choice, true/false, and short-answer questions
- *PowerPoint Lectures:* Lecture notes with figures from the book

TALK TO US

This book continues to evolve. Your comments, suggestions, and corrections will be greatly appreciated. You can contact us through the publisher at computing@aw.com, or:

Computer Science Editorial Office
Addison-Wesley
501 Boylston Street, Suite 900
Boston, MA 02116

ACKNOWLEDGMENTS

The suggestions from outstanding reviewers have, through the past few editions, contributed greatly to this book's present form. In alphabetical order, they are:

- Ronald Alferez—*University of California at Santa Barbara*
Claude W. Anderson—*Rose-Hulman Institute of Technology*
Don Bailey—*Carleton University*
N. Dwight Barnette—*Virginia Tech*
Jack Beidler—*University of Scranton*
Elizabeth Sugar Boese—*Colorado State University*
Debra Burhans—*Canisius College*
Tom Capaul—*Eastern Washington University*
Eleanor Boyle Chlan—*Johns Hopkins University*
Chakib Chraibi—*Barry University*
Jack N. Donato—*Jefferson Community College*
Susan Gauch—*University of Kansas*
Mark Holliday—*Western Carolina University*
Lily Hou—*SUN Microsystems, Inc.*
Helen H. Hu—*Westminster College*
Lester I. McCann—*The University of Arizona*
Rameen Mohammadi—*SUNY, Oswego*
Narayan Murthy—*Pace University*
Thaddeus F. Pawlicki—*University of Rochester*

Timothy Rolfc—*Eastern Washington University*

Hongjun Song—*University of Memphis*

We especially thank the people who produced this book. Our editors at Addison-Wesley, Michael Hirsch and Stephanie Sellinger, provided invaluable guidance and assistance. Also, Marilyn Lloyd, Linda Knowles, Yez Alayan and Kathryn Ferranti contributed their expertise and care during the final production and in the marketing of the book.

Many other wonderful people have contributed in various ways. They are Doug McCreadie, Michael Hayden, Sarah Hayden, Andrew Hayden, Albert Prichard, Frances Prichard, Sarah Mason, Karen Mellor, Maybeth Conway, Ted Emmott, Lorraine Berube, Marge White, James Kowalski, Ed Lamagna, Gerard Baudet, Joan Peckham, Victor Fay-Wolfe, Bala Ravikumar, Karl Abrahamson, Ronnie Smith, James Wirth, Randy Hale, John Cardin, Gail Armstrong, Tom Manning, Jim Abreu, Bill Harding, Hal Records, Laurie MacDonald, Ken Fougere, Ken Sousa, Chen Zhang, Suhong Li, Richard Glass, and Aby Chaudhury. In special memory of Wallace Wood.

Numerous other people provided input for the previous editions of *Walls and Mirrors* at various stages of its development. All of their comments were useful and greatly appreciated. In alphabetical order, they are: Stephen Alberg, Vicki Allan, Jihad Almahayni, James Ames, Andrew Azzinaro, Tony Baiching, Don Bailey, Wolfgang W. Bein, Sto Bell, David Berard, John Black, Richard Botting, Wolfin Brumley, Philip Carrigan, Stephen Clamage, Michael Clancy, David Clayton, Michael Cleron, Chris Constantino, Shaun Cooper, Charles Denault, Vincent J. DiPippo, Suzanne Dorney, Colleen Dunn, Carl Eckberg, Karla Steinbrugge Fant, Jean Foltz, Marguerite Hafen, George Hamer, Judy Hankins, Lisa Hellerstein, Mary Lou Hines, Jack Hodges, Stephanie Horoschak, John Hubbard, Kris Jensen, Thomas Judson, Laura Kenney, Roger King, Ladislav Kohout, Jim LaBonte, Jean Lake, Janusz Laski, Cathie LeBlanc, Urban LeJeune, John M. Linebarger, Ken Lord, Paul Luker, Manisha Mande, Pierre-Arnoul de Marneffe, John Marsaglia, Jane Wallace Mayo, Mark McCormick, Dan McCracken, Vivian McDougal, Shirley McGuire, Sue Medeiros, Jim Miller, Guy Mills, Cleve Moler, Paul Nagin, Rayno Niemi, Paul Nagin, John O'Donnell, Andrew Oldroyd, Larry Olsen, Raymond L. Paden, Roy Pargas, Brenda C. Parker, Keith Pierce, Lucasz Pruski, George B. Purdy, David Radford, Steve Ratner, Stuart Regis, J. D. Robertson, John Rowe, Michael E. Rupp, Sharon Salveter, Charles Saxon, Chandra Sekharan, Linda Shapiro, Yujian Sheng, Mary Shields, Carl Spicola, Richard Snodgrass, Neil Snyder, Chris Spannabel, Paul Spirakis, Clinton Staley, Matt Stallman, Mark Stehlick, Harriet Taylor, David Teague, David Tetreault, John Turner, Susan Wallace, James E. Warren, Jerry Weltman, Nancy Wiegand, Howard Williams, Brad Wilson, Salih Yurttas, and Alan Zaring.

Thank you all.

F. M. C.
J. J. P.

PART ONE

Problem-Solving Techniques

The primary concern of the six chapters in Part One of this book is to develop a repertoire of problem-solving techniques that form the basis of the rest of the book. Chapter 1 begins by providing a brief overview of Java fundamentals. Chapter 2 describes the characteristics of a good solution and the ways to achieve one. These techniques emphasize abstraction, modularity, and information hiding. The remainder of Part One discusses data abstraction for solution design, more Java for use in implementations, and recursion as a problem-solving strategy.

CHAPTER 1

Review of Java Fundamentals

This book assumes that you already know how to write programs in a modern programming language. If that language is Java, you can probably skip this chapter, returning to it for reference as necessary. If instead you know a language such as C++, this chapter will introduce you to Java.

It isn't possible to cover all of Java in these pages. Instead this chapter focuses on the parts of the language used in this book. First we discuss basic language constructs such as variables, data types, expressions, operators, arrays, decision constructs, and looping constructs. Then we look at the basics of program structure, including packages, classes, and methods, with a brief introduction to inheritance. We continue with useful Java classes, exceptions, text input and output, and files.

1.1 Language Basics

- Comments
- Identifiers and Keywords
- Variables
- Primitive Data Types
- References
- Literal Constants
- Named Constants
- Assignments and Expressions
- Arrays

1.2 Selection Statements

- The *if* Statement
- The *switch* Statement

1.3 Iteration Statements

- The *while* Statement
- The *for* Statement
- The *do* Statement

1.4 Program Structure

- Packages
- Classes
- Data Fields
- Methods
- How to Access Members of an Object

1.5 Useful Java Classes

- The *Object* Class
- String Classes

1.6 Java Exceptions

- Catching Exceptions
- Throwing Exceptions

1.7 Text Input and Output

- Input
- Output

1.8 File Input and Output

- Text Files
- Object Serialization

Summary

Cautions

1.1 Language Basics

Let's begin with the elements of the language that allow you to perform simple actions within a program. The following sections provide a brief overview of the basic language constructs of Java.

Comments

A variety of commenting styles are available in Java

Each comment line in Java begins with two slashes (//) and continues until the end of the line. You can also begin a multiple-line comment with the character /* and end it with */. Although the programs in this book do not use /* and */, it is a good idea to use this notation during debugging. That is, to isolate an error, you can temporarily ignore a portion of a program by enclosing it within /* and */. However, a comment that begins with /* and ends with */ cannot contain another comment that begins with /* and ends with */. Java also has a third kind of comment that is used to generate documentation automatically using *javadoc*, a documentation utility available in the Software Development Kit (SDK). This comment uses a /** to start and a */ to end.

Identifiers and Keywords

Java is case sensitive

A Java **identifier** is a sequence of letters, digits, underscores, and dollar signs that must begin with either a letter or an underscore. Java distinguishes between uppercase and lowercase letters, so be careful when typing identifiers.

You use identifiers to name various parts of the program. Certain identifiers, however, are reserved by Java as **keywords**, and you should not use them for other purposes. A list of all Java keywords appears inside the front cover of this book. The keywords that occur within Java statements in this book are in boldface.

Variables

A variable contains either the value of a primitive data type or a reference to an object

A variable, whose name is a Java identifier, represents a memory location that contains a value of a primitive data type or a reference. You declare a variable's data type by preceding the variable name with the data type, as in

```
double radius; // radius of a sphere  
String name; // reference to a String object
```

Note that the second declaration does not create a *String* object, only a variable that stores the location of a *String* object. You must use the *new* operator to create a new object.

Primitive Data Types

The primitive data types in Java are organized into four categories: boolean, character, integer, and floating point. For example, the following two lines declare variables of the primitive type *double*.

```
double radius;  
double radiusCubed;
```

Some of the data types are available in two forms and sizes. Figure 1-1 lists the available primitive data types.

A boolean value can be either *true* or *false*. You represent characters by enclosing them in single quotes or by providing their Unicode integer value (see Appendix B). Integer values are signed and allow numbers such as -5 and +98. The floating-point types provide for real numbers that have both an integer portion and a fractional portion. Character and integer types are called **integral types**. Integral and floating-point types are called **arithmetic types**.

A value of a primitive type is not considered to be an object and thus cannot be used in situations where an object type is expected. For this reason, the package *java.lang* provides corresponding **wrapper classes** for each of the primitive types. Figure 1-1 also lists the wrapper class corresponding to each of the primitive types.

Each of these classes provides a constructor to convert a value of a primitive type to an object when necessary. Once such an object has been created, the value contained within the object cannot be modified. Here is a simple example involving integers:

```
int x = 9;  
Integer intObject = new Integer(x);  
System.out.println("The value stored in intObject = "  
    + intObject.intValue());
```

A wrapper class is available for each primitive data type

You can represent the value of a primitive data type by using a wrapper class

Category	Data Type	Wrapper Class
Boolean	boolean	Boolean
Character	char	Character
Integer	byte short int long	Byte Short Integer Long
Floating point	float double	Float Double

FIGURE 1-1

Primitive data types and corresponding wrapper classes

The class *Integer* has a method *intValue* that retrieves the value stored in an *Integer* object. Classes corresponding to the other primitive types provide methods with similar functionality.

Java has a feature called **autoboxing** that makes it easier to convert from a primitive type to their equivalent wrapper class counterparts. In the previous example, we explicitly created a new *Integer* object to store the value 9. With autoboxing, we can simply write

```
Integer intObject = 9;
```

The compiler automatically adds the code to convert the integer value into the proper class (*Integer* in this example).

The reverse process of converting an object of one of the wrapper classes into a value of the corresponding primitive type is called **auto-unboxing**. In the example

```
int x = intObject + 1;
```

the compiler again automatically generates the code to convert the *Integer* object *intObject* to a primitive type (*int* in this example) so that the expression can be evaluated.

References

A reference variable contains an object's location in memory

Java has one other type, called a **reference**, that is used to locate an object. Unlike other languages, such as C++, Java does not allow the programmer to perform any operations on the reference value. When an object is created using the *new* operator, the location of the object in memory is returned and can be assigned to a reference variable. For example, the following line shows the reference variable *name* being assigned the location of a new *String* object:

```
String name = new String("Sarah");
```

A special reference value of *null* is provided to indicate that a reference variable has no object to reference.

Literal Constants

Literal constants indicate particular values within a program

You use literal constants to indicate particular values within a program. In the following expression, the 4 and 3 are examples of literal constants that are used within a computation.

```
4 * Math.PI * radiusCubed / 3
```

You can also use a literal constant to initialize the value of a variable. For example, you use `true` and `false` as the values of a boolean variable, as we mentioned previously.

You write decimal integer constants without commas, decimal points, or leading zeros.¹ The default data type of such a constant is either `int`, if small enough, or `long`.

You write floating constants, which have a default type of `double`, with a decimal point. You can specify an optional power-of-10 multiplier by writing `e` or `E` followed by the power of 10. For example, `1.2e-3` means 1.2×10^{-3} .

Character constants are enclosed in single quotes—for example, '`A`' and '`2`'—and have a default type of `char`. You write a literal character string as a sequence of characters enclosed in double quotes.

Several characters have names that use a backslash notation, as given in Figure 1-2. This notation is useful when you want to embed one of these characters within a literal character string. For example, the statement

```
System.out.println("Hello\n Let's get started!");
```

uses the new-line character `\n` to place a new-line character after the string `Hello`. You will learn about this use of `\n` in the discussion of output later in this chapter. You also use the backslash notation to specify either a single quote as a character constant (`\'`) or a double quote (`\"`) within a character string.

Do not begin a decimal integer constant with zero

Named Constants

Unlike variables, whose values can change during program execution, named constants have values that do not change. The declaration of a named constant is like that of a variable, but the keyword `final` precedes the data type. For example,

```
final float DEFAULT_RADIUS = 1.0;
```

The value of a named constant does not change

Constant	Name
<code>\n</code>	New line
<code>\t</code>	Tab
<code>\'</code>	Single quote
<code>\"</code>	Double quote
<code>\0</code>	Zero

FIGURE 1-2

Some special character constants

1. Octal and hexadecimal constants are also available, but they are not used in this book. An octal constant begins with `0`, a hex constant with `0x` or `0X`.

Named constants make a program easier to read and modify

declares `DEFAULT_RADIUS` as a named floating-point constant. Once a constant such as `DEFAULT_RADIUS` is declared, you can use it in your code and assign it another value. By using named constants, your code becomes both easier to read and easier to modify.

Assignments and Expressions

You form an expression by combining variables, constants, operators, and parentheses. The assignment statement

`radius = initialRadius;`

assigns to a previously declared variable `radius` the value of the expression on the right-hand side of the assignment operator `=`, assuming that `initialRadius` has a value. The assignment statement

`double radiusCubed = radius * radius * radius;`

also declares `radiusCubed`'s data type, and assigns it a value.

Arithmetic expressions. You can combine variables and constants using arithmetic operators and parentheses to form arithmetic expressions. The arithmetic operators are

- | | |
|---|---|
| • <code>*</code> Multiply | • <code>+</code> Binary add or unary plus |
| <code>/</code> Divide | <code>-</code> Binary subtract or unary minus |
| <code>%</code> Remainder after division | |

Operators have a set precedence.

The operators `*`, `/`, and `%` have the same precedence,² which is higher than that of `+` and `-`; unary operators³ have a higher precedence than binary operators.

The following examples demonstrate operator precedence:

- | | | | |
|------------------------|-------|--------------------------|---|
| <code>a - b / c</code> | means | <code>a - (b / c)</code> | (precedence of <code>/</code> over <code>-</code>) |
| <code>-5 / a</code> | means | <code>(-5) / a</code> | (precedence of unary <code>-</code>) |
| <code>a / -5</code> | means | <code>a / (-5)</code> | (precedence of unary <code>-</code>) |

Arithmetic operators and most other operators are left-associative, which means that operators of the same precedence execute from left to right in an expression. Thus,

2. A list of all Java operators and their precedences appears inside the back cover of this book.
3. A unary operator requires only one operand, for example, the `-` in `-5`. A binary operator requires two operands, for example, the `+` in `2 + 3`.

$a / b * c$

means

$(a / b) * c$

Operators are either left- or right-associative

The assignment operator and all unary operators are **right-associative**, as you will see later. You can use parentheses to override operator precedence and associativity.

Relational and logical expressions. You can combine variables and constants with parentheses; with the **relational, or comparison, operators** $<$, \leq , \geq , and $>$; and with the **equality operators** == (equal to) and != (not equal to) to form a **relational expression**. Such an expression evaluates to **false** if the specified relation is false and to **true** if it is true. For example, the expression $5 \text{ != } 4$ has a value of **true** because 5 is not equal to 4. Note that equality operators have a lower precedence than relational operators. Also note that the equality operators work correctly only with the primitive types and references. The == operator determines only whether two reference variables are referencing the same object, but not whether two objects are equal.

You can combine variables and constants of the arithmetic types, relational expressions, and the **logical operators** $\&\&$ (and) and $\| \|$ (or) to form **logical expressions**, which evaluate to **false** if false and to **true** if true. Java evaluates logical expressions from left to right and stops as soon as the value of the entire expression is apparent; that is, Java uses **short-circuit evaluation**. For example, Java determines the value of each of the following expressions without evaluating $(a < b)$:

```
(5 == 4) && (a < b) // false since (5 == 4) is false
(5 == 5) || (a < b) // true since (5 == 5) is true
```

Equality operators work correctly only with primitive types and references

Logical expressions are evaluated from left to right

Sometimes the value of a logical expression is apparent before it is completely examined

Conversions from one data type to another occur during both assignment and expression evaluation

Implicit type conversions for the primitive numeric types. Automatic conversions from one numeric data type to another can occur during assignment and during expression evaluation. For assignments, the data type of the expression on the right-hand side of the assignment operator is converted to the data type of the item on the left-hand side just before the assignment occurs. Floating-point values are truncated—not rounded—when they are converted to integral values.

During the evaluation of an expression, any values of type *byte*, *char*, or *short* are converted to *int*. These conversions are called **integral promotions**. After these conversions, if the operands of an operator differ in data type, the data type that is lower in the following hierarchy is converted to one that is higher (*int* is lowest):

int → *long* → *float* → *double*

For example, if *A* is *long* and *B* is *float*, *A* + *B* is *float*. A copy of *A*'s *long* value is converted to *float* prior to the addition; the value stored at *A* is unchanged.

Explicit type conversions for primitive numeric types. Numeric conversions from one type to another are possible by means of a **cast**. The cast operator is a unary operator formed by enclosing the desired data type within parentheses. Thus, the sequence

```
double volume = 14.9;  
System.out.print((int)volume);
```

displays 14.

You convert from one numeric type to another by using a cast

Multiple assignment. If you omit the semicolon from an assignment statement, you get an **assignment expression**. You can embed assignment expressions within assignment expressions, as in *a* = 5 + (*b* = 4).

This expression first assigns 4 to *b* and then 9 to *a*. This notation contributes to the terseness of Java and is sometimes convenient, but it can be confusing. The assignment operator is right-associative. Thus, *a* = *b* = *c* means *a* = (*b* = *c*).

Other assignment operators. In addition to the assignment operator =, Java provides several two-character assignment operators that perform another operation before assignment. For example,

a += *b* means *a* = *a* + *b*

Other operators, such as -=, *=, /=, and %=, have analogous meanings.

Two more operators, ++ and --, provide convenient incrementing and decrementing operations:

++*a* means *a* += 1, which means *a* = *a* + 1

Similarly,

--*a* means *a* -= 1, which means *a* = *a* - 1

The operators ++ and -- are useful for incrementing and decrementing a variable

The operators ++ and -- can either precede their operands, as you just saw, or follow them. Although *a*++, for instance, has the same effect as ++*a*, the results differ when the operations are combined with assignment. For example,

b = ++*a* means *a* = *a* + 1; *b* = *a*

Here, the ++ operator acts on *a* *before* the assignment to *b* of *a*'s new value. In contrast,

b = *a*++ means *b* = *a*; *a* = *a* + 1

The assignment operator assigns *a*'s old value to *b* *before* the ++ operator acts on *a*. That is, the ++ operator acts on *a* *after* the assignment. The operators ++

and `--` are often used within loops and with array indexes, as you will see later in this chapter.

In addition to the operators described here, Java provides several other operators. A summary of all Java operators and their precedences appears inside the back cover of this book.

Arrays

An array is a collection of elements, items, or values that have the same data type. Array elements have an order: An array has a first element, a second element, and so on, as well as a last element. That is, an array contains a finite, limited number of elements. Like objects, an array does not come into existence until it is allocated using the `new` statement. At that time, you specify the desired size of the array. Because you can access the array elements directly and in any order, an array is a direct access, or random access, data structure.

An array is a collection of data that has the same type

One-dimensional arrays. When you decide to use an array in your program, you must declare it and, in doing so, indicate the data type of its elements. The following statements declare a **one-dimensional array**, `maxTemps`, which contains the daily maximum temperatures for a given week:

```
final int DAYS_PER_WEEK = 7;  
double [] maxTemps = new double[DAYS_PER_WEEK];
```

You can access array elements directly and in any order

The bracket notation `[]` declares `maxTemps` as an array. The array is then allocated memory for seven floating-point elements.

The declared length of an array is accessible using the data field `length` associated with the array. For example, `maxTemps.length` is 7. You can refer to any of the floating-point elements in `maxTemps` directly by using an expression, which is called the **index**, or **subscript**, enclosed in square brackets. In Java, array indexes must have integer values in the range 0 to `length - 1`, where `length` is the data field just described. The indexes for `maxTemps` range from 0 to `DAYS_PER_WEEK - 1`. For example, `maxTemps[4]` is the fifth element in the array. If `k` is an integer variable whose value is 4, `maxTemps[k]` is the fifth element in the array, and `maxTemps[k+1]` is the sixth element. Also, `maxTemps[++k]` adds 1 to `k` and then uses the new value of `k` to index `maxTemps`, whereas `maxTemps[k++]` accesses `maxTemps[k]` before adding 1 to `k`. Note that you use one index to refer to an element in a one-dimensional array.

Use an index to specify a particular element in an array

Figure 1-3 illustrates the array `maxTemps`, which at present contains only five temperatures. The last value in the array is `maxTemps[4]`; the values of `maxTemps[5]` and `maxTemps[6]` are 0.0, the default initial value for floating-point numbers.

An array index has an integer value greater than or equal to 0

You can initialize the elements of an array when you declare it by specifying an **initializer list**. The initializer list is a list of values separated by commas and enclosed in braces. For example,

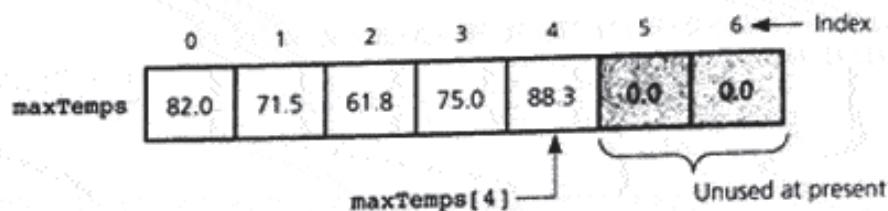


FIGURE 1-3

A one-dimensional array of at most seven elements

You can initialize an array when you declare it.

```
double [] weekDayTemps = {82.0, 71.5, 61.8, 75.0, 88.3};
```

initializes the array *weekDayTemps* to have five elements with the values listed. Thus, *weekDayTemps[0]* is 82.0, *weekDayTemps[1]* is 71.5, and so on.

You can also declare an array of object references. The declaration is similar to that of an array of primitive types. Here is a declaration of an array for ten *String* references:

```
String[] stuNames = new String[10];
```

Note that all of the references will have the initial value *null* until actual *String* objects are created for them to reference. The following statement creates a *String* object for the first element of the array:

```
stuName[0] = new String("Andrew");
```

Multidimensional arrays. You can use a one-dimensional array, which has one index, for a simple collection of data. For example, you can organize 52 temperatures linearly, one after another. A one-dimensional array of these temperatures can represent this organization.

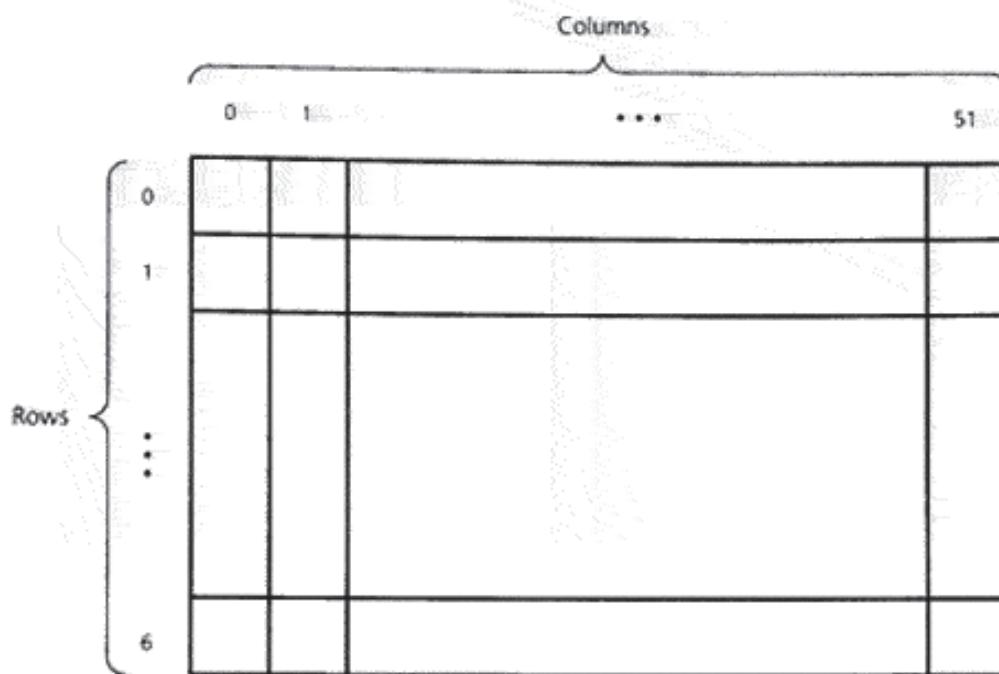
You can also declare multidimensional arrays. You use more than one index to designate an element in a multidimensional array. Suppose that you wanted to represent the minimum temperature for each day during 52 weeks. The following statements declare a two-dimensional array, *minTemps*:

```
final int DAYS_PER_WEEK = 7;
final int WEEKS_PER_YEAR = 52;

double[][] minTemps = new
    double[DAYS_PER_WEEK][WEEKS_PER_YEAR];
```

These statements specify the ranges for two indexes: The first index can range from 0 to 6, while the second index can range from 0 to 51. Most people picture a two-dimensional array as a rectangular arrangement, or matrix, of elements

An array can have more than one dimension

**FIGURE 1-4**

A two-dimensional array

that form rows and columns, as Figure 1-4 indicates. The first dimension given in the definition of `minTemps` is the number of rows. Thus, `minTemps` has 7 rows and 52 columns. Each column in this matrix represents the seven daily minimum temperatures for a particular week.

To reference an element in a two-dimensional array, you must indicate both the row and the column that contain the element. You make these indications of row and column by writing two indexes, each enclosed in brackets. For example, `minTemps[1][51]` is the element in the 2nd row and the 52nd column. In the context of the temperature example, this element is the minimum temperature recorded for the 2nd day (Monday) of the 52nd week. The rules for the indexes of a one-dimensional array also apply to the indexes of multidimensional arrays.

As an example of how to use a two-dimensional array in a program, consider the following program segment, which determines the smallest value in the previously described array `minTemps`:

```
// minTemps is a two-dimensional array of daily minimum  
// temperatures for 52 weeks, where each column of the  
// array contains temperatures for one week.
```

```
// initially, assume the lowest temperature is  
// first in the array  
double lowestTemp = minTemps[0][0];  
int dayOfWeek = 0;  
int weekOfYear = 0;
```

In a two-dimensional array, the first index represents the row, the second index represents the column

An example of using a two-dimensional array

```

// search array for lowest temperature
for (int weekIndex = 0; weekIndex < WEEKS_PER_YEAR;
     ++weekIndex) {
    for (int dayIndex = 0; dayIndex < DAYS_PER_WEEK;
         ++dayIndex) {
        if (lowestTemp > minTemps[dayIndex][weekIndex]) {
            lowestTemp = minTemps[dayIndex][weekIndex];
            dayOfWeek = dayIndex;
            weekOfYear = weekIndex;
        } // end if
    } // end for
} // end for
// Assertion: lowestTemp is the smallest value in
// minTemps and occurs on the day and week given by
// dayOfWeek and weekOfYear; that is, lowestTemp ==
// minTemps[dayOfWeek][weekOfYear].

```

It is entirely possible to declare *minTemps* as a one-dimensional array of 364 ($7 * 52$) elements, in which case you might use *minTemps[81]* instead of *minTemps[4][11]* to access the minimum temperature on the 4th day of the 11th week. However, doing so will make your program harder to understand!

Although you can declare arrays with more than two dimensions, it is unusual to have an array with more than three dimensions. The techniques for working with such arrays, however, are analogous to those for two-dimensional arrays.

You can initialize the elements of a two-dimensional array just as you initialize a one-dimensional array. You list the initial values row by row. For example, the statement

```
int[][] x = {{1,2,3},{4,5,6}};
```

initializes a 2-by-3 array *x* so that it appears as

1	2	3
4	5	6

That is, the statement initializes the elements *x[0][0]*, *x[0][1]*, *x[0][2]*, *x[1][0]*, *x[1][1]*, and *x[1][2]* in that order. In general, when you assign initial values to a multidimensional array, it is the last, or rightmost, index that increases the fastest.

1.2 Selection Statements

Selection statements allow you to choose among several courses of action according to the value of an expression. In this category of statements, Java provides the *if* statement and the *switch* statement.

The *if* Statement

You can write an *if* statement in one of two ways:

```
if (expression)
    statement1
```

An *if* statement has
two basic forms

or

```
if (expression)
    statement1
else
    statement2
```

where *statement₁* and *statement₂* represent any Java statement. Such statements can be **compound**; a compound statement, or **block**, is a sequence of statements enclosed in braces. Though not a requirement of Java, this text will always use a compound statement in language constructs, even if only a single statement is required.

If the value of *expression* is **true**, *statement₁* is executed. Otherwise, the first form of the *if* statement does nothing, whereas the second form executes *statement₂*. Note that the parentheses around *expression* are required.

For example, the following *if* statements each compare the values of two integer variables *a* and *b*:

```
if (a > b) {
    System.out.println(a + " is larger than " + b + ".");
} // end if
System.out.println("This statement is always executed.");

if (a > b) {
    larger = a;
    System.out.println(a + " is larger than " + b + ".");
}
else {
    larger = b;
    System.out.println(b + " is larger than " + a + ".");
} // end if

System.out.println(larger + " is the larger value.");
```

Parentheses around
the expression in
an *if* statement
are required

You can nest *if* statements in several ways, since either *statement₁* or *statement₂* can itself be an *if* statement. The following example, which determines the largest of three integer variables *a*, *b*, and *c*, shows a common way to nest *if* statements:

You can nest *if* statements

```
if ((a >= b) && (a >= c)) {
    largest = a;
}
else if (b >= c) {      // a is not largest at this point
    largest = b;
}
else {
    largest = c;
} // end if
```

A *switch* statement provides a choice of several actions according to the value of an integral expression

Without a *break* statement, execution of a *case* will continue into the next *case*

The *switch* Statement

When you must choose among more than two courses of action, the *if* statement can become unwieldy. If your choice is to be made according to the value of an integral expression, you can use a *switch* statement.

For example, the following statement determines the number of days in a month. The *int* variable *month* designates the month as an integer from 1 to 12.

```
switch (month) {
    // 30 days hath Sept., Apr., June, and Nov.
    case 9: case 4: case 6: case 11:
        daysInMonth = 30;
        break;
    // all the rest have 31
    case 1: case 3: case 5: case 7: case 8: case 10: case 12:
        daysInMonth = 31;
        break;

    // except February
    case 2: // assume leapYear is true if leap
             // year, else is false
        if (leapYear) {
            daysInMonth = 29;
        }
        else {
            daysInMonth = 28;
        } // end if
        break;

    default:
        System.out.println("Incorrect value for Month.");
} // end switch
```

Parentheses must enclose the integral *switch* expression—*month*, in this example. The *case* labels have the form

case *expression*:

where *expression* is a constant integral expression. After the *switch* expression is evaluated, execution continues at the *case* label whose expression has the same value as the *switch* expression. Subsequent statements execute until either a *break* or a *return* is encountered or the *switch* statement ends.

It bears repeating that unless you terminate a *case* with either a *break* or a *return*, execution of the *switch* statement continues. Although this action can be useful, omitting the *break* statements in the previous example would be incorrect.

If no *case* label matches the current value of the *switch* expression, the statements that follow the *default* label, if one exists, are executed. If no *default* exists, the *switch* statement exits.

1.3 Iteration Statements

Java has three statements—the *while*, *for*, and *do* statements—that provide for repetition by iteration—that is, loops. Each statement controls the number of times that another Java statement—the body—is executed. The body can be a single statement, though this text will always use a compound statement.

The **while** Statement

The general form of the *while* statement is

```
while (expression)
    statement
```

As long as the value of *expression* is *true*, *statement* is executed. Because *expression* is evaluated before *statement* is executed, it is possible that *statement* will not execute at all. Note that the parentheses around *expression* are required.

Suppose that you wanted to compute the sum of a list of integers stored in an array called *myArray*. The following *while* loop accomplishes this task:

```
int sum = 0;
int index = 0;
while (index <= myArray.length) {
    sum += myArray[index];
} // end while
```

A **while** statement executes as long as the expression is *true*

The *break* and *continue* statements. You can use the *break* statement—which you saw earlier within a *switch* statement—within any of the iteration statements. A *break* statement within the body of a loop causes the loop to exit immediately. Execution continues with the statement that follows the loop. This use of *break* within a *while*, *for*, or *do* statement is generally considered poor style.

Use of a **break** statement within a loop is generally poor style

The `continue` statement stops only the current iteration of the loop and begins the next iteration at the top of the loop. The `continue` statement is valid only within `while`, `for`, or `do` statements.

The `for` Statement

The `for` statement provides for counted loops and has the general form

```
for (initialize; test; update)
    statement
```

A `for` statement lists the initialization, testing, and updating steps in one location

where *initialize*, *test*, and *update* are expressions. Typically, *initialize* is an assignment expression that initializes a counter to control the loop. This initialization occurs only once. Then, if *test*, which is usually a logical expression, is `true`, *statement* executes. The expression *update* executes next, usually incrementing or decrementing the counter. This sequence of events repeats, beginning with the evaluation of *test*, until the value of *test* is `false`. As with the previous constructs, *statement* is usually a compound statement.

For example, the following `for` statement sums the integers from 1 to *n*:

```
int sum = 0;
for (int counter = 1; counter <= n; ++counter) {
    sum += counter;
} // end for

// this statement is always executed
int x = 0;
```

If *n* is less than 1, the `for` statement does not execute at all. Thus, the previous statements are equivalent to the following `while` loop:

```
int sum = 0;
int counter = 1;
while (counter <= n) {
    sum += counter;
    ++counter;
} // end while
// this statement is always executed
int x = 0;
```

In general, the logic of a `for` statement is equivalent to

A `for` statement is equivalent to a `while` statement

```
initialize;
while (test) {
    statement;
    update;
} // end while
```

with the understanding that if *statement* contains a *continue*, *update* will execute before *test* is evaluated again.

The following two examples demonstrate the flexibility of the *for* statement:

```
for (byte ch = 'z'; ch >= 'a'; --ch) {  
    // ch ranges from 'z' to 'a'  
    // statements to process ch  
}  
  
for (double x = 1.5; x < 10; x += 0.25) {  
    // x ranges from 1.5 to 9.75 at steps of 0.25  
    // statements to process x  
}
```

The *initialize* and *update* portions of a *for* statement each can contain several expressions separated by commas, thus performing more than one action. For example, the following loop raises a floating-point value to an integer power by using multiplication:

```
// floating-point power equals floating-point x  
// raised to int n; assumes integer expon  
for (power = 1.0, expon = 1; expon <= n; ++expon){  
    power *= x;  
}  
// end for
```

Both *power* and *expon* are assigned values before the body of the loop executes for the first time.

Because the *for* statement consolidates the initialization, testing, and updating steps of a loop into one statement, Java programmers tend to favor it over the *while* statement. For example, notice how the following *while* loop sums the values in an array *x*:

```
sum = 0;  
int i = 0;  
while (i < x.length) {  
    sum += x[i];  
    i++;  
}  
// end while
```

A **for** statement is usually favored over the **while** statement

This loop is equivalent to the following *for* statement:

```
for (int i = 0, sum = 0; i < x.length; sum += x[i++]) {  
}
```

In fact, this *for* statement has an empty body!

You can omit any of the initialization, testing, and updating steps in a `for` statement, but you cannot omit the semicolons

You can omit any of the expressions *initialize*, *test*, or *update* from a `for` statement, but you cannot omit the semicolons. For example, you can move the *update* step from the previous `for` statement to the body of the loop:

```
for (int i = 0, sum = 0; i < x.length; ) {  
    sum += x[i++];  
} // end for
```

You also could omit both the initialization and the update steps, as in the following loop:

```
for ( ; x > 0; ) {  
    statements to process nextValue in inputLine  
} // end for
```

This `for` statement offers no advantage over the equivalent `while` statement:

```
while (x > 0)
```

Although you can omit the *test* expression from `for`, you probably will not want to do so, because then the loop would be infinite.

The `for` loop and arrays. Java provides a loop construct that simplifies iteration through the elements of an array. A logical name for this loop construct would be the “foreach” loop, but the language developers wanted to avoid adding a new keyword to the language. So the new form of the `for` loop is often referred to as the “enhanced for loop.”

The syntax for the enhanced for loop when used with arrays is as follows:

```
for (ArrayElementType variableName : arrayName)  
    statement
```

where *ArrayElementType* is the type of each element in the array, and *arrayName* is the name of the array you wish to process element by element. The loop begins with the *variableName* assigned the first element in the array. With each iteration of the loop, *variableName* is associated with the next element in the array. This continues until all of the elements in the array have been processed. For example:

```
String[] nameList = { "Janet", "Frank", "Mike", "Doug" };  
for (string name: nameList) { // for each name in nameList  
    System.out.println(name);  
} // end for
```

is equivalent to the following:

```
String[] nameList = { "Janet", "Frank", "Mike", "Doug"};
for (int index=0; index < nameList.length; index++) {
    System.out.println(nameList[index]);
} // end for
```

The do Statement

Use the **do** statement when you want to execute a loop at least once. Its general form is

```
do {
    statement
} while (expression);
```

A **do** statement loops at least once

Here, *statement* executes until the value of *expression* is *false*.

For example, suppose that you execute a sequence of statements and then ask the user whether to execute them again. The **do** statement is appropriate, because you execute the statements before you decide whether to repeat them:

```
char response;
do {
    . . . (a sequence of statements)
    ask the user if they want to do it again
    store user's response in response
} while ( (response == 'Y') || (response == 'y') );
```

1.4 Program Structure

Let's begin our discussion of program structure with the simple Java application in Figure 1-5 that computes the volume of a sphere. It consists of two classes, **SimpleSphere** and **TestClass**. Each of these classes is contained in a separate file that has the same name as the class, with **.java** appended to the end. A typical Java program consists of several classes, some of which you write and some of which you use from the Java Application Programming Interface (API). A Java application has one class that contains a method **main**, the starting point for program execution. Running the program in Figure 1-5 produces the following output:

```
The volume of a sphere of radius 19.1 inches is 29186.95
```

Each Java application must contain at least one class that has a method **main**

This application includes all of the basic elements of Java program structure (packages, classes, data fields, and methods). The sections that follow discuss each of these elements.

File SimpleSphere.java

```

package MyPackage;
import java.lang.Math;
public class SimpleSphere {
    private double radius;
    public static final double DEFAULT_RADIUS = 1.0;
    public SimpleSphere() {
        radius = DEFAULT_RADIUS;
    } // end default constructor
    public SimpleSphere(double initialRadius) {
        radius = initialRadius;
    } // end constructor
    public double getRadius() {
        return radius;
    } // end getRadius
    public double getVolume() {
        // Computes the volume of the sphere.
        double radiusCubed = radius * radius * radius;
        return 4 * Math.PI * radiusCubed / 3;
    } // end getVolume
} // end SimpleSphere

```

1. Indicates SimpleSphere is part of a package ----->

2. Indicates class Math is used by SimpleSphere ----->

3. Begins class SimpleSphere ----->

4. Declares a private data field radius ----->

5. Declares a constant ----->

6. A default constructor ----->

7. Assignment statement ----->

8. A second constructor ----->

9. Assignment statement ----->

10. Begins method getRadius ----->

11. Returns data field radius ----->

12. Begins method getVolume ----->

13. A comment ----->

14. Declares and assigns a local variable ----->

15. Returns result of computation ----->

16. Ends class SimpleSphere ----->

File TestClass.java

```

package MyPackage;
public class TestClass {
    static public void main(String[] args) {
        SimpleSphere ball;
        ball = new SimpleSphere(19.1);
        System.out.println("The volume of a sphere of radius "
            + ball.getRadius() + " inches is "
            + (float)ball.getVolume()
            + "cubic inches\n");
    } //end main
} // end TestClass

```

17. Indicates TestClass is part of a package ----->

18. Begins class TestClass ----->

19. Begins method main ----->

20. Declares reference ball ----->

21. Creates a SimpleSphere object ----->

22. Outputs results ----->

23. Continuation of output string ----->

24. Continuation of output string ----->

25. Ends class TestClass ----->

FIGURE 1-5

A simple Java application

Packages

Java packages provide a mechanism for grouping related classes. To indicate that a class is part of a package, you include a *package* statement as the first program line of your code. For example, lines 1 and 17 in Figure 1-5 indicate

that both of these classes, `SimpleSphere` and `TestClass`, are in the package `MyPackage`. The format of the `package` statement is

```
package package-name;
```

Java assumes that all of the classes in a particular package are contained in the same directory. Furthermore, this directory must have the same name as the package.

The Java API actually consists of many predefined packages. Some of the more common of these packages are `java.lang`, `java.util`, and `java.io`. The dot notation in these package names directly relates to the directory structure containing these packages. In this case, all of the directories corresponding to these packages are contained in a parent directory called `java`.

import statement. The `import` statement allows you to use classes contained in other packages. The format of the `import` statement is as follows:

```
import package-name.class-name;
```

For example, line 2 in Figure 1-5 imports the class `Math` from the package `java.lang`. The following line also could have been used:

```
import java.lang.*;
```

In this case, the `*` indicates that all of the items from the package `java.lang` should be imported. Actually, this particular line can be omitted from the program, since `java.lang` is implicitly imported to all Java code. Explicitly importing `java.lang.Math` makes it clear to others who read your code that you are using the class `Math` in this code.

Classes

An object in Java is an instance of a class. You can think of a class as a data type that specifies the data and methods that are available for instances of the class. A class definition includes an optional subclassing modifier, an optional access modifier, the keyword `class`, an optional `extends` clause, an optional `implements` clause, and a class body. Figure 1-6 describes each of the components of a class.

When a new class is created in Java, it is either specifically made a subclass of another class through the use of the `extends` clause or it is implicitly a subclass of the Java class `Object`. Creating a subclass is known as inheritance and is discussed briefly in Chapter 4 and in depth in Chapter 9 of this text.

To create an object or instance of a class, you use the `new` operator. For example, the expression

```
new SimpleSphere()
```

creates an instance of the type `SimpleSphere`.

To include a class in a package, begin the class's source file with a **package** statement

Place the files that contain a package's classes in the same directory

The **import** statement provides access to classes within a package

An object is an instance of a class

A Java class defines a new data type

Component	Syntax	Description
Subclassing modifier (use only one)	<code>abstract</code>	Class must be extended to be useful.
	<code>final</code>	Class cannot be extended.
Access modifiers	<code>public</code>	Class is available outside of package.
	no access modifier	Class is available only within package.
Keyword <code>class</code>	<code>class class-name</code>	Class should be contained in a file called <code>class-name.java</code> .
<code>extends</code> clause	<code>extends superclass-name</code>	Indicates that this class is a subclass of the class <code>superclass-name</code> in the <code>extends</code> clause.
<code>implements</code> clause	<code>implements interface-list</code>	Indicates the interfaces that this class implements. The <code>interface-list</code> is a comma-separated list of interface names.
Class body	Enclosed in braces	Contains data fields and methods for the class.

FIGURE 1-6

Components of a class

Now let's briefly examine the contents of the class body: data fields and methods.

Data Fields

Data fields are class members that are either variables or constants. Data field declarations can contain modifiers that control the availability of the data field (access modifiers) or that modify the way the data field can be used (use modifiers). The access modifiers are effective only if the class is declared `public`. Although this text uses only a subset of the modifiers, Figure 1-7 shows them all for completeness.

Type of modifier	Keyword	Description
Access modifier (use only one)	public	Data field is available everywhere (when the class is also declared public).
	private	Data field is available only within the class.
	protected	Data field is available within the class, available in subclasses, and available to classes within the same package.
	No access modifier	Data field is available within the class and within the package.
Use modifiers (all can be used at once)	static	Indicates that only one such data field is available for all instances of this class. Without this modifier, each instance has its own copy of a data field.
	final	The value provided for the data field cannot be modified (a constant).
	transient	The data field is not part of the persistent state of the object.
	volatile	The value provided for the data field can be accessed by multiple threads of control. Java ensures that the freshest copy of the data field is always used.

FIGURE 1-7

Modifiers used in data field declarations

Data fields are typically declared **private** or **protected** within a class, with access provided by methods in the class. Hence, a method within a class has access to all of the data fields declared in the class. This allows the developer of the class to maintain control over how the data stored within the class is used.

A class's data fields should be **private** or **protected**

Methods

Methods are used to implement operations. The syntax of a method declaration is as follows:

```
access-modifier use-modifiers return-type
                    method-name (formal-parameter-list) {
    method-body
}
```

Usually, each method should perform one well-defined task. For example, the following method returns the larger of two integers:

A method definition implements a method's task

```
public static int max(int x, int y) {
    if (x > y) {
        return x;
    }
    else {
        return y;
    } // end if
} // end max
```

Method modifiers can be categorized as access modifiers and use modifiers, with the access modifier typically appearing first. In the example just given, the access modifier *public* appears first, followed by the use modifier *static*. Again, although this text uses only a subset of modifiers, Figure 1-8 shows them all for completeness.

The **return type** of a **valued method**—one that returns a value—is the data type of the value that the method will return. The body of a valued method must contain a statement of the form

A valued method must use **return** to return a value

```
return expression;
```

where *expression* has the value to be returned. A method can also return a reference to an object. For the method *max*, the return type is *int*. The type of the value must be specified immediately before the method name. If the method does not have a value to return, the return type is specified as *void*.

After the method name, the formal parameter list appears in parentheses. You declare a formal parameter by writing a data type and a parameter name, separating it from other formal parameter declarations with a comma, as in

```
int x, int y
```

Type of modifier	Keyword	Description
Access modifier (use only one)	public	Method is available everywhere (when the class is also declared as <i>public</i>).
	private	Method is available only within the class (cannot be declared <i>abstract</i>).
	protected	Method is available within the class, available in subclasses, and available to classes within the same package.
	No access modifier	Method is available within the class and to classes within the package.
Use modifiers (all can be used at once)	static	Indicates that only one such method is available for all instances of this class. Since a <i>static</i> method is shared by all instances, the method can refer only to data fields that are also declared <i>static</i> and shared by all instances.
	final	The method cannot be overridden in a subclass.
	abstract	The method must be overridden in a subclass.
	native	The body of the method is not written in Java but in some other programming language.
	synchronized	The method can be run by only one thread of control at a time.

FIGURE 1-8

Modifiers used in a method declaration

When you call, or invoke, the method *max*, you pass it actual arguments that correspond to the formal parameters with respect to number, order, and data type. For example, the following method contains two calls to *max*:

```
public void printLargest(int a, int b, int c) {
    int largerAB = max(a, b);
    System.out.println("The largest of " + a + ", " + b + ", "
        + " and " + c + " is " + max(largerAB, c));
} // end printLargest
```

When you call a method, you pass it actual arguments that correspond to the formal parameters in number, order, and data type

An actual argument passed by value is copied within the method

Arguments passed to Java methods are passed by value. That is, the method makes local copies of the values of the actual arguments—*a* and *b*, for example—and uses these copies wherever *x* and *y* appear in the method definition. Thus, the method cannot alter the actual arguments that you pass to it.

Passing an array to a method. If you want a method to compute the average of the first *n* elements of a one-dimensional array, you could declare the method as follows:

```
public static double averageTemp(double[] temps, int n)
```

You can invoke the method by writing

```
double avg = averageTemp(maxTemps, 6);
```

where *maxTemps* is declared an integer array of any length, and *maxTemps* is the previously defined array.

Arrays are always passed by reference to a method

The location of the array is passed to the method. You cannot return a new array through this value, but the method can modify the contents of the array. This restriction avoids the copying of perhaps many array elements. Thus, the method *averageTemp* could modify the elements of *maxTemps*.

An argument that is a reference can be used to directly access the object or array

So note that when the formal parameter is an object or an array, the actual argument is a reference value that is copied. This means that you can change the contents of the array or object, but not the value of the reference itself. For example, you cannot have a method that creates a new object for a reference in the parameter list. If it does, the new reference value will simply be discarded when the method terminates, and the original reference to the object will be left intact.

Java has a feature that allows a method to have a variable number of arguments of the same type. When defining the method, the rightmost parameter of the method uses the ellipses (three consecutive dots) to indicate that any number of arguments of that type can be specified. For example:

```
public static int max(int... numbers) {  
  
    int maximum = Integer.MIN_VALUE;  
    for (int num : numbers) {  
        if (maximum < num){  
            maximum = num;  
        } // end if  
    } // end for  
    return maximum;  
} // end max
```

Note that the variable arguments can be accessed as an array, where the formal parameter name is used as the name of the array within the method. This also means that you can use the same techniques you use to process arrays, such as using the enhanced for loop as demonstrated here.

Constructors. There is one special kind of method called a **constructor**. Constructor methods have the same name as the class and no return type. The constructor is executed only when a new instance of the class is created. A class can contain multiple constructors, differentiated by the number and types of the parameters. The actual arguments you provide when creating a new instance determine which constructor is executed.

A constructor allocates memory for an object and can initialize the object's data to particular values. A class can have more than one constructor, as is the case for the class *SimpleSphere*.

The first constructor in *SimpleSphere* is the **default constructor**. A default constructor by definition has no parameters. Typically, a default constructor initializes data fields to values that the class implementation chooses. For example, the implementation

```
public SimpleSphere() {  
    radius = DEFAULT_VALUE; // DEFAULT_VALUE = 1.0  
} // end default constructor
```

sets *radius* to 1.0. The following statement invokes the default constructor, which creates the object *unitSphere* and sets its radius to 1.0:

```
SimpleSphere unitSphere = new SimpleSphere();
```

The next constructor in *SimpleSphere* is

```
public SimpleSphere(double initialRadius) {  
    setRadius(initialRadius);  
} // end constructor
```

It creates a sphere object of radius *initialRadius*. You invoke this constructor by writing a declaration such as

```
SimpleSphere mySphere = new SimpleSphere(5.1);
```

In this case, the object *mySphere* has a radius of 5.1.

If you omit all constructors from your class, the compiler will generate a default constructor—that is, one with no parameters—for you. A compiler generated **default constructor**, however, might not initialize data fields to values that you will find suitable.

If you define a constructor that has parameters, *but you omit the default constructor*, the compiler will not generate one for you. Thus, you will not be able to write statements such as

```
SimpleSphere defaultSphere = new SimpleSphere();
```

How to Access Members of an Object

You can access data fields and methods that are declared *public* by naming the object, followed by a period, followed by the member name:

```
static public void main(String[] args) {  
    SimpleSphere ball = new SimpleSphere(19.1);  
    System.out.println("The volume of a sphere of radius "  
        + ball.getRadius() + " inches is "  
        + (float)ball.getVolume()  
        + "cubic inches\n");  
} //end main
```

An object such as *ball* can, upon request, return its radius and compute its volume. These requests to an object are called **messages** and are simply calls to methods. Thus, an object responds to a message by acting on its data. To invoke an object's method, you qualify the method's name—such as *getRadius*—with the object variable—such as *ball*.

The previous program is an example of a **client** of a class. A client of a particular class is simply a program or module that uses the class. We will reserve the term **user** for the person who uses a program. You can also access members of a class that are declared *static* (data fields or methods that are shared by all instances of the class) by using the class name followed by the name of the static member. For example, the *static* field *DEFAULT_RADIUS* declared in line 5 of Figure 1-5 can be accessed outside of the class as follows:

```
SimpleSphere.DEFAULT_RADIUS;
```

Class Inheritance

A brief discussion of inheritance is provided here, since it is a common way to create new classes in Java. A more complete discussion of inheritance appears in Chapter 9.

Suppose that we want to create a class for colored spheres, knowing that we have already developed the class *SimpleSphere*. We could write an entirely new class for the colored spheres, but if the colored spheres are actually like the spheres in the class *SimpleSphere*, we can reuse the *SimpleSphere* implementation and add color operations and characteristics by using

A reference to the private data field *radius* would be illegal within this program

inheritance. Here is an implementation of the class *ColoredSphere* that uses inheritance:

```
import java.awt.Color;
public class ColoredSphere extends SimpleSphere {
    private Color color;

    public ColoredSphere(Color c) {
        super();
        color = c;
    } // end constructor

    public ColoredSphere(Color c, double initialRadius) {
        super(initialRadius);
        color = c;
    } // end constructor

    public void setColor(Color c) {
        color = c;
    } // end setColor

    public Color getColor() {
        return color;
    } // end getColor
} // end ColoredSphere
```

A class derived
from the class
SimpleSphere

SimpleSphere is called the base class or superclass, and *ColoredSphere* is called the derived class or subclass of the class *SimpleSphere*. The definition of the subclass includes an *extends* clause that indicates the superclass to be used. When you declare a class without an *extends* clause, you are implicitly extending the class *Object*, so *Object* is its superclass.

The subclass inherits the contents of the superclass, details of which are discussed in Chapter 9. For the moment, suffice it to say that the subclass will have all of the public members of the superclass available. Any instance of the subclass is also considered to be an instance of the superclass and can be used in a program anywhere that an instance of the superclass can be used. Also, any of the publicly defined methods or variables that can be used with instances of the superclass can be used with instances of the subclass. The subclass instances also have the methods and variables that are publicly defined in the subclass definition.

In the constructor for the *ColoredSphere* class, notice the use of the keyword *super*. You use this keyword to call the constructor of the superclass, so *super()* calls the constructor *SimpleSphere()*, and *super(initialRadius)* calls the constructor *SimpleSphere(double initialRadius)*. If the subclass constructor explicitly calls the superclass constructor, the call to *super* must precede all other statements in the subclass constructor. Note that if a subclass

Public members of
the superclass are
available in the
subclass

A constructor in a
subclass should
invoke **super** to call
the constructor of
the superclass

constructor contains no call to the superclass constructor, the default superclass constructor is implicitly called.

If a subclass needs to call a method defined in the superclass, the call is preceded by the keyword *super*. For example, to make a call to the method *getVolume* from within the class *ColoredSphere*, you would write the following:

```
super.getVolume()
```

Here is an example of a method that uses the *ColoredSphere* class:

```
public void useColoredSphere() {
    ColoredSphere redBall =
        new ColoredSphere(java.awt.Color.red);
    System.out.println("The ball volume is " +
        redBall.getVolume());
    System.out.println("The ball color is " +
        redBall.getColor());
    // other code here...
} // end useColorSphere
```

This method uses the constructor and the method *getColor* from the subclass *ColoredSphere*. It also uses the method *getVolume* that is defined in the superclass *SimpleSphere*.

1.5 Useful Java Classes

The Java Application Programming Interface (API) provides a number of useful classes. The classes mentioned here are ones that are used within this text.

The *Object* Class

Every Java class inherits the methods of the class *Object*

Java supports a single class inheritance hierarchy, with the class *Object* as the root. Thus, the class *Object* provides a number of useful methods that are inherited by every Java class. In some cases, it is common for a class to redefine, or override, the version of the method inherited from *Object*. The paragraphs that follow summarize some of the more useful methods from the class *Object*.

```
public boolean equals(Object obj)
```

Indicates whether some other object is “equal to” this one. As defined in the class *Object*, equality is based upon references—that is, upon whether both of the references are referencing the same object. This is referred to as shallow equality.

Default *equals* as defined in the class *Object* compares two references

Let's examine the `equals` method for objects a bit further. Suppose we have the following code:

```
SimpleSphere s1 = new SimpleSphere();
SimpleSphere s2 = s1;
if (s1.equals(s2)) {
    System.out.println("s1 and s2 are the same object");
} // end if
```

This will produce the following output:

```
s1 and s2 are the same object
```

It is common for a class to redefine this method for deep equality—in other words, to check the equality of the contents of the objects.

Suppose that you want to determine whether two spheres have the same radius. For example,

```
SimpleSphere s1 = new SimpleSphere(2.0);
SimpleSphere s3 = new SimpleSphere(2.0);
if (s1.equals(s3)) {
    System.out.println("s1 and s3 have the same radius");
}
else {
    System.out.println("s1 and s3 have different radii");
} // end if
```

will produce the output

```
s1 and s3 have different radii
```

which is not true! Both `s1` and `s3` have a radius of 2.0. Remember that the default `equals` compares two references; they differ here because they reference two distinct objects. If you want to have `equals` check the values contained in the object for equality, you must redefine `equals` in the class. Here is an example of such an `equals` for the class `SimpleSphere`:

```
public boolean equals(Object rhs) {
    return ((rhs instanceof SimpleSphere) &&
            (radius == ((SimpleSphere)rhs).radius));
} // end equals
```

Notice that the parameter of `equals` is of type `Object`. Remember, we are overriding this method as inherited from the class `Object`, and the parameter list and return value must match. Also notice that we are explicitly checking to make sure that the object parameter `rhs` is an instance of the

Customizing `equals` for a class

An `equals` method that determines whether two spheres have the same radius

class *SimpleSphere* by using the *instanceof* operator. If the incoming object *rhs* is an instance of the class *SimpleSphere* (or one of its subclasses), *instanceof* will return *true*; otherwise, the operator returns *false*. Thus, the *equals* method will return *false* when *rhs* is of a class other than *Sphere*. If the *instanceof* operator returns *true*, the boolean expression proceeds to check whether the data fields are equal. In this example, the data field of the class *SimpleSphere* is a primitive type. If an object is used as a data field, *equals* may have to be defined for that object's class as well. It is up to the designer to decide how "deep" the equality checks must be for a particular class.

Other useful *Object* methods include the following:

protected void finalize()

Java has a garbage collection mechanism to destroy objects that a program no longer needs. When a program no longer references an object, the Java runtime environment marks it for garbage collection. Periodically, the Java runtime environment executes a method that returns the memory used by these marked objects to the system for future use. The garbage collector calls the *finalize* method on an object when it determines that there are no more references to the object.

public int hashCode()

Associated with each object is a unique identifying value called a hash code. This method returns the hash code for the object as an integer.

public String toString()

Returns a string that "textually represents" this object. As defined in the class *Object*, this method returns a string that contains the name of the class of which the object is an instance, followed by the at-sign character (@), and ending with the unsigned hexadecimal representation of the hash code of the object. For example, given the statement

`Sphere mySphere = new Sphere();`

the method call `mySphere.toString()` will return a string similar to `Sphere@733f42ab`.

The Array Class

This class contains various static methods for manipulating arrays. Many of the methods have unique specifications for each of the primitive types (*boolean*, *byte*, *char*, *short*, *int*, *long*, *float*, *double*). To simplify the presentation of these methods, *ptype* will be used as a placeholder for a primitive type. Though only the methods for the primitive types are specifically discussed, many of the methods also support an array of elements of type *Object* and generic types.

```
public static ptype[] copyOf(ptype[] original, int newLength)
```

Copies the specified array of primitive types, truncating or padding (if needed) so the copy has the specified length. If padding is necessary, the numeric types will pad with *zero*, *char* will pad with *null*, and *boolean* will pad with *false*.

```
public static ptype[] copyOfRange(ptype[] original,
                                  int beginIndex, int endIndex)
```

Copies the range *beginIndex* to *endIndex-1* of the specified array into a new array. The index *beginIndex* must lie between zero and *original.length*, inclusive. As long as there are values to copy, the value at *original[beginIndex]* is placed into the first element of the new array, with subsequent elements in the original array placed into subsequent elements in the new array. Note that *beginIndex* must be less than or equal to *endIndex*. The length of the returned array will be *endIndex- beginIndex*.

```
public static String toString(ptype[] a)
```

Returns a string representation of the contents of the specified array. The resulting string consists of a list of the array's elements, separated by a comma and a space, enclosed in square brackets ("[]"). It returns *null* if the array is null.

```
public static int binarySearch(ptype[] a, ptype key)
```

Searches the array for the *key* value using the binary search algorithm. The array must be sorted before making this call. If it is not sorted, the results are undefined. If the array contains duplicate elements with the *key* value, there is no guarantee which one will be found. For floating point types, this method considers all *NaN* values to be equivalent and equal. The method is not defined for *boolean* or *short*.

```
public static void sort(ptype[] a)
```

Sorts the array into ascending order. For floating point values, the method uses the total order imposed by the appropriate *compareTo* method and all *NaN* values are considered equivalent and equal. This method is not defined for *boolean* or *short*.

String Classes

Java provides three classes that are useful when working with strings: *String*, *StringBuffer*, and *StringTokenizer*. The class *String* is a nonmutable

string type; once the value of the string has been set, it cannot be modified. The class *StringBuffer* implements a mutable sequence of characters; it provides many of the same operations as the *String* class plus others for changing the characters stored in the string. Although at first glance it would seem reasonable for us to simply study *StringBuffer*, using *String* is more efficient. In fact, many methods within the Java API use the class *String*. The last class, *StringTokenizer*, provides methods for breaking strings into pieces.

The class *String*. Earlier, you saw that Java provides literal character strings such as

"This is a string."

This section describes how you can create and use variables that contain such strings. Java provides a class *String* in the package *java.lang* to support non-mutable strings. A nonmutable string is one that cannot be changed once it has been created. Instances of the *String* class can be combined to form new strings, and numerous methods are provided for examining *String* objects. Our presentation includes only some of the possible operations on strings.

You can declare a string reference *title* by writing

```
String title;
```

When you initialize a string variable with a string literal, Java actually creates a *String* object to store the string literal and assigns the reference to the variable. Thus, you can assign a *String* reference by writing

```
String title = "Walls and Mirrors";
```

You can subsequently assign another string to *title* by using an assignment statement such as

```
title = "J Perfect's Diary";
```

Note that this actually creates a new *String* instance for *title* to reference.

In each of the previous examples, *title* has a length of 17. You use the method *length* to determine the current length of a string. Thus, *title.length()* is equal to 17.

You can reference the individual characters in a string by using the method *charAt* with the same index that you would use for an array. Thus, in the previous example, *title.charAt(0)* contains the character *J*, and *title.charAt(16)* contains the character *y*.

You should not use the == operator to test whether two strings are equal. Using the == operator determines only whether the references to the strings are the same; it does not compare the contents of the *String* instances.

Use the method *length* to determine a string's length

Use *charAt* to reference any character within a string

You can compare strings by using the `compareTo` method. Not only can you determine whether two strings are equal, but you can also determine which of two strings comes before the other according to the Unicode table. The `compareTo` method is used as follows:

```
string1.compareTo(string2)
```

Use `compareTo` to compare two strings

The character sequence represented by the `String` object `string1` is compared to the character sequence represented by the argument `string2`. The result is a negative integer if `string1` precedes `string2`. The result is a positive integer if `string1` follows `string2`. The result is zero if the strings are equal. The ordering of two strings is analogous to alphabetic ordering, but you use the Unicode table instead of the alphabet. The following expressions demonstrate the behavior of `compareTo`:

```
"dig".compareTo("dog")      //returns negative
"Star".compareTo("star")    //returns negative
"abc".compareTo("abc")     //returns zero
"start".compareTo("star")   //returns positive
"d".compareTo("abc")       //returns positive
```

You can concatenate two strings to form another string by using the `+` operator. That is, you place one string after another to form another string. For example, if

```
String s = "Com";
```

the statements

```
String t = s + "puter";
s += "puter";
```

Use the `+` operator to concatenate two strings

assign the string "Computer" to each of `t` and `s`. Similarly, you can append a single character to a string, as in

```
s += 's';
```

Besides adding two strings together, you can also concatenate a string and a value of a primitive type together by using the `+` operator. For example,

```
String monthName = "December";
int day = 31;
int year = 02;
String date = monthName + " " + day + ", 20" + year;
```

assigns the string "December 31, 2002" to `date`.

As we mentioned earlier, the class `Object` has a method called `toString` that returns a string that “textually represents” an object. The result of the `toString` method is often combined with other strings by means of the `+` operator.

You can examine a portion of a string by using the method

Use `substring` to access part of a string

```
public String substring(int beginIndex, int endIndex)
```

The first parameter, `beginIndex`, specifies the position of the beginning of the substring. (Remember that 0 is the position of the first character in the string.) The end of the substring is at position `endIndex - 1`. For example, in

```
title = "J Perfect's Diary";
```

```
title.substring(2, 9) is the string "Perfect".
```

Other useful `String` methods include the following:

Other useful `String` methods

```
public int indexOf(String str, int fromIndex)
```

Returns the index of the first substring equal to `str`, starting from the index `fromIndex`.

```
public String replace(char oldChar, char newChar)
```

Returns a string that is obtained by replacing all characters `oldChar` in the string with `newChar`.

```
public String trim()
```

Returns a string that has all leading and trailing spaces in the original string removed.

Instances of the class `StringBuffer` are strings that you can alter

The class `StringBuffer`. In some situations, it is useful to be able to alter the sequence of characters stored in a string. But class `String` supports only nonmutable strings. To create mutable strings (strings that can be modified) use the class `StringBuffer` from the package `java.lang`. This class provides the same functionality as the class `String`, plus the following methods that actually modify the value stored in the `StringBuffer` object:

```
public StringBuffer append(String str)
```

Appends the string `str` to this string buffer.

```
public StringBuffer insert(int offset, String str)
```

The string `str` is inserted into this string buffer at the index indicated by `offset`. Any characters originally above that position are moved up and the length of this string buffer increased by the length of `str`. If `str` is `null`, the string “`null`” is inserted into this string buffer.

```
public StringBuffer delete(int start, int end)
```

Removes the characters in a substring of this string buffer starting at index *start* and extending to the character at index *end* - 1 or to the end of the string buffer if no such character exists. If *start* is equal to *end*, no changes are made. This method may throw *StringIndexOutOfBoundsException* if the value of *start* is negative, greater than the length of the string buffer, or greater than *end*.

```
public void setCharAt(int index, char ch)
```

The character at index *index* of this string buffer is set to *ch*. This method may throw *IndexOutOfBoundsException* if the value of *index* is negative or is greater than or equal to the length of the string buffer.

```
public StringBuffer replace(int start, int end,  
                           String str)
```

Replaces the characters in a substring of this string buffer with characters in the specified string *str*. The substring to be replaced begins at index *start* and extends to the character at index *end* - 1 or to the end of the string buffer if no such character exists. The substring is removed from the string buffer, and then the string *str* is inserted at index *start*. If necessary, the string buffer is lengthened to accommodate the string *str*. This method may throw *StringIndexOutOfBoundsException* if the value of *start* is negative, greater than the length of the string buffer, or greater than *end*.

The class StringTokenizer. Another useful class when working with strings is *StringTokenizer* in the package *java.util*. This class allows a program to break a string into pieces or **tokens**. The tokens are separated by characters known as delimiters. When you create a *StringTokenizer* instance, you must specify the string to be tokenized. Other constructors within *StringTokenizer* allow you to specify the delimiting characters and whether the delimiting characters themselves should be returned as tokens. Here is brief description of the three constructors for *StringTokenizer*:

```
public StringTokenizer(String str)
```

This constructor creates a string tokenizer for the specified string *str*. The tokenizer uses the default delimiter set, which is the space character, the tab character, the newline character, the carriage-return character, and the form-feed character. Delimiter characters themselves are not treated as tokens.

```
public StringTokenizer(String str, String delim)
```

This constructor creates a string tokenizer for the specified string *str*. All characters in the *delim* string are the delimiters for separating tokens. Delimiter characters themselves are not treated as tokens.

Instances of the class *StringTokenizer* are strings that you can break into pieces called tokens

```
public StringTokenizer(String str, String delim,
                      boolean returnTokens)
```

This constructor creates a string tokenizer for the specified string *str*. All characters in the *delim* string are the delimiters for separating tokens. If the *returnTokens* flag is true, the delimiter characters are also returned as tokens. Each delimiter is returned as a string of length 1. If the flag is false, the delimiter characters are skipped and serve only as separators between tokens.

StringTokenizer also provides the following methods for retrieving tokens from the string:

```
public String nextToken()
```

Returns the next token in the string. If there are no more tokens in the string, it throws the exception *NoSuchElementException*. Exceptions are discussed in the next section.

```
public boolean hasMoreTokens()
```

Returns *true* if the string contains more tokens.

1.6 Java Exceptions

An exception is a mechanism for handling an error during execution

Many programming languages, including Java, support a mechanism known as an **exception**, which handles an error during execution. A method indicates that an error has occurred by **throwing** an exception. The exception returns to the point at which you invoked the method, where you **catch** the exception and deal with the error condition.

Catching Exceptions

To handle an exception, Java provides **try-catch** blocks. You place the statement that might throw an exception within a **try** block. The **try** block must be followed by one or more **catch** blocks. Each **catch** block indicates the type of exception you want to handle. A **try** block can have many **catch** blocks associated with it, since even a single statement may be capable of throwing more than one type of exception. Also, the **try** block can contain many statements, any of which might throw an exception. Here is the general syntax for a **try** block:

```
try {
    statement(s);
}
```

Use a **try** block for statements that can throw an exception

The syntax for a **catch** block is as follows:

```
catch (exceptionClass identifier) {
    statement(s);
}
```

Use a **catch** block
for each type of
exception that you
handle

When a statement in the **try** block actually throws an exception, the remainder of the **try** block is abandoned, and control is passed to the **catch** block that corresponds to the type of exception thrown. The statements in the **catch** block then execute, and upon completion of the **catch** block, execution resumes at the point following the last **catch** block.

The system decides which **catch** block to execute by considering the **catch** blocks in the order in which they appear, using the first one that produces a legal assignment of the thrown exception and the argument specified in the **catch** block. Thus, you must order the **catch** blocks so that the most specific exception classes appear before the more general exception classes; otherwise, the code will not compile. For example,

```
try {
    int result = 99 / 0;
    // other statements appear here
} // end try
catch (Exception e) {
    System.out.println("Something else was caught");
} // end catch
catch (ArithmaticException e) {
    System.out.println("ArithmaticException caught");
} // end catch
```

The order of these
two **catch** blocks is
incorrect

compiles with an error message similar to the following:

```
TestExceptionExample.java:43: exception
    java.lang.ArithmaticException has already been caught
        catch (ArithmaticException e) {
        ^
1 error
```

To get the code to compile successfully, you must switch the order of the **catch** blocks.

The following program demonstrates what happens when an exception is thrown and not caught. Figure 1-9 illustrates these events.

```
class ExceptionExample {
    private int [] myArray;

    public ExceptionExample() {
        myArray = new int[10];
    } // end default constructor
```

This program does
not handle the
exception that is
thrown and, there-
fore, execution
terminates

```
public void addValue(int n, int value) {
    // add value to element n by calling addOne n times
    for (int i = 1; i <= value; i++) {
        addOne(n);
    } // end for
} // end addValue

public void addOne(int n) {
    // add 1 to the element n
    myArray[n] += 1;
} // end addOne
} end ExceptionExample

public class TestExceptionExample {
    public static void main(String[] args) {
        ExceptionExample e1 = new ExceptionExample();
        e1.addValue(99, 3); // add 3 to element 99
    } // end main
} // end TestExceptionExample
```

The method `addOne` causes `ArrayIndexOutOfBoundsException` from `java.lang` to be thrown when an attempt is made to access `myArray[99]`. Since `addOne` does not provide a handler for the exception (Point 1 in Figure 1-9), the method terminates and the exception is propagated back to `addValue` to the point where `addOne` was called. The method `addValue` also does not provide an exception handler, so it also terminates (Point 2 in Figure 1-9) and the exception is propagated back to `main`. Since `main` is the main method of the program, and the exception is not handled in `main` (Point 3 in Figure 1-9), the program terminates, and an error message similar to the following is displayed on the screen:

```
java.lang.ArrayIndexOutOfBoundsException: 99
at ExceptionExample.addOne(ExceptionExample.java)
at ExceptionExample.addValue(Compiled Code)
at TestExceptionExample.main(TestExceptionExample.java)
```

Notice that the error message for the exception includes a stack trace, the sequence of method calls that led to the exception being thrown. This is the default behavior when no exception handler is provided. The message may also contain information specific to the exception at hand; in this case, it contains the index value 99 that caused the exception to be thrown.

This code does not indicate that the method `addOne` might throw the exception `ArrayIndexOutOfBoundsException`. The method's documentation should indicate the exceptions it might throw.

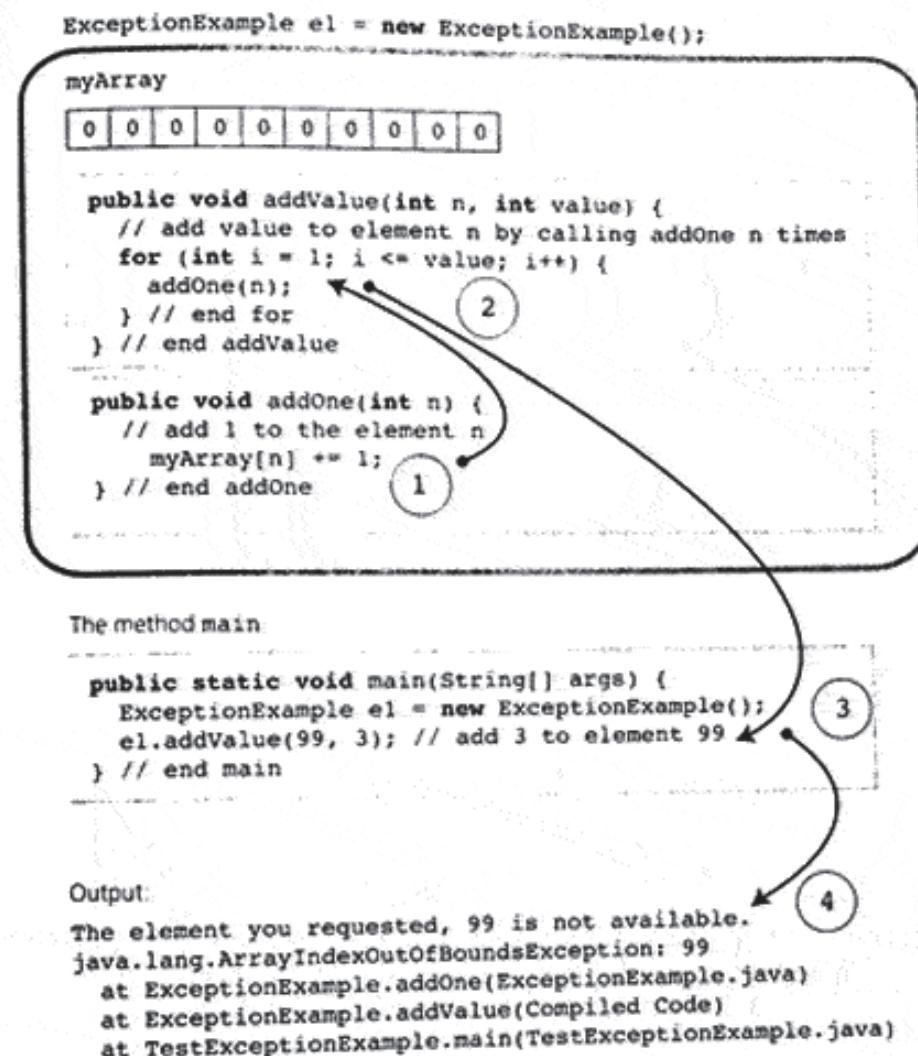


FIGURE 1-9

Flow of control in a simple Java application

The exception `ArrayIndexOutOfBoundsException` could be caught at any point in the sequence of method calls. For example, `addOne` in the class `ExceptionExample` could be rewritten as follows to catch the exception:

```

public void addOne(int n) {
    try {
        myArray[n] += 1;
    } // end try
    catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("The element you requested, " +
                           n + ", is not available.");
    } // end catch
} // end addOne

```

An example of handling an exception

This version of `addOne` produces the following output:

```
The element you requested, 99, is not available.  
The element you requested, 99, is not available.  
The element you requested, 99, is not available.
```

The method `addOne` is called three times by `addValue` when `e1.addValue(99, 3)` executes, and hence the exception is thrown three times. When the exception was not handled, the program terminated the first time the exception occurred. By adding a `catch` block to handle the exception, we allow the code to continue execution.

Although the `addOne` method is where `ArrayIndexOutOfBoundsException` is thrown, it is not necessarily the best place to handle the exception. For example, if the call `e1.addValue(99, 10000)` executed, the message printed by `addOne` would have appeared 10,000 times! In this case it makes more sense for the handler to appear in the `addValue` method, and not in the `addOne` method. This assumes that `addOne` no longer handles the exception but propagates it back to `addValue`. Here is the code for `addValue` with the exception handler:

An improved way to handle an exception

```
public void addValue(int n, int value) {  
    try {  
        for (int i = 1; i <= value; i++) {  
            addOne(n);  
        } // end for  
    } // end try  
    catch (ArrayIndexOutOfBoundsException e) {  
        System.out.println("The element you requested, " +  
                           n + " is not available.");  
        e.printStackTrace();  
    } // end catch  
} // end addValue
```

This method produces the following output:

```
The element you requested, 99 is not available.  
java.lang.ArrayIndexOutOfBoundsException: 99  
at ExceptionExample.addOne(ExceptionExample.java)  
at ExceptionExample.addValue(Compiled Code)  
at TestExceptionExample.main(TestExceptionExample.java)
```

When `addOne` throws the exception `ArrayIndexOutOfBoundsException`, it is propagated back to `addValue`. The method `addValue` abandons execution of the statements in the `try` block, executes the statement in the `catch` block, and resumes execution after the last `catch` block. The message is printed only once, since the `for` loop is inside the `try` block, which is abandoned when the exception occurs. If the `try` block was placed inside the `for` loop (around the

call to `addOne`), the exception would be thrown and handled at each iteration of the loop, causing the message to be printed multiple times.

The `catch` block also contains the method call `e.printStackTrace()`. Recall that the `catch` block specifies the type of exception handled and an identifier. This identifier provides a name for the caught exception that can be used within the `catch` block. In this case, the method `printStackTrace` is called for the exception object `e`. The `printStackTrace` method is one of many methods available to exception objects. Other uses of the exception name in the `catch` block are discussed in the next section on throwing exceptions.

You may have noticed that some exceptions from the Java API cannot be totally ignored. You must provide a handler for these exceptions. For example, in the class `java.io.FileInputStream`, the constructor will throw `java.io.FileNotFoundException` if the file specified cannot be found. In this case, the compiler will complain if no exception handler is provided. For example, compiling the following code:

```
import java.io.*;
public class TestExceptionExample {
    public static void getInput(String fileName) {
        FileInputStream fis;
        fis = new FileInputStream(fileName);
        // file processing code appears here
    } // end getInput

    static public void main(String[] args) {
        getInput("test.dat");
    } // end main
} // end TestExceptionExample
```

produces a compilation error message similar to the following:

```
TestExceptionExample.java:5: unreported exception
java.io.FileNotFoundException must be caught, or declared
to be thrown
    fis = new FileInputStream(fileName);
               ^
1 error
```

One way to resolve this error message is to provide an exception handler within the `getInput` method as follows:

```
public static void getInput(String fileName) {
    FileInputStream fis;
    try {
        fis = new FileInputStream(fileName);
        // file processing code appears here
    } // end try
```

Some exceptions
must be handled

```
        catch (FileNotFoundException e) {
            System.out.println("The file " + fileName +
                " is not available");
            System.out.println(e);
        } // end catch
        System.out.println("After try-catch blocks");
    } // end getInput
```

Output similar to the following results when the file named `test.dat` does not exist:

```
The file test.dat is not available
java.io.FileNotFoundException: test.dat
    at java.io.FileInputStream.<init>(FileInputStream.java:56)
    at TestExceptionExample.getInput(TestExceptionExample.java)
    at TestExceptionExample.main(TestExceptionExample.java)
After try-catch blocks
```

Two types of exceptions: checked and runtime

Java has two types of exceptions: checked exceptions and runtime exceptions. The exception `java.io.FileNotFoundException` is an example of a checked exception. Checked exceptions are instances of classes that are subclasses of the `java.lang.Exception` class. They must be handled locally or explicitly thrown from the method (as discussed in the next section). They are typically used when the method encounters a serious problem. In some cases, the error may be considered serious enough that the program should be terminated.

Runtime exceptions occur when the error is not considered as serious. These types of exceptions can often be prevented by fail-safe programming. For example, it is fairly easy to avoid allowing an array index to go out of range, a situation that causes the runtime exception `ArrayIndexOutOfBoundsException` to be thrown. Runtime exceptions are instances of classes that are subclasses of the `java.lang.RuntimeException` class. `RuntimeException` is a subclass of `java.lang.Exception` that relaxes the requirement forcing the exception to be either handled locally or explicitly thrown by the method.

The `finally` block. As an option, you can follow the last `catch` block with a `finally` block that has the following form:

```
finally {
    statement(s);
}
```

This block is executed whether or not an exception is thrown within the `try` block. If an exception is thrown, the appropriate `catch` block executes and then the `finally` block executes. If no exception is thrown, the `finally` block executes upon completion of the `try` block. Note that you can have a `finally` block even if no `catch` block is present. Later in this chapter—in the section *File Input and Output*—you will see an example of a `finally` block that deals with a file when the program no longer needs it.

Throwing Exceptions

As we've mentioned, all exceptions in Java are instances of the class `java.lang.Exception` or one of its subclasses. When a method specification contains a `throws` clause, it also specifies the type of exception that the method can throw. If the method can throw more than one type of exception, each is listed after the `throws` clause, separated by commas. For example, here is the method header for one of the constructors for `FileInputStream`:

```
public FileInputStream(String name)
    throws FileNotFoundException
```

A `throws` clause indicates that a method might throw an exception

The `throws` clause indicates that a method may throw an exception if an error occurs during its execution. In this case, the constructor will throw the exception `FileNotFoundException` if the file specified by `name` can't be opened.

An exception is thrown when the `throw` statement is executed. The syntax of this statement is:

```
throw reference
```

where `reference` refers to an instance of a subclass of the class `java.lang.Exception`. When the `throw` statement executes, the remaining code in the `try` block or method is ignored. Typically, a `throw` statement will appear as follows:

```
throw new exceptionClass(stringArgument);
```

Use a `throw` statement to throw an exception

where `exceptionClass` is the type of exception you want to throw, and `stringArgument` is an argument to the `exceptionClass` constructor that specifies the detail message, a more detailed description of what may have caused the exception.

In certain situations, the Java API will have a predefined exception class that will suit the exception needs of your program. For example, the Java API has an exception `java.lang.IndexOutOfBoundsException` that could be used when an array's index is out of range.

You may also want to define your own exception class. Usually, you use `Exception` or `RuntimeException` as the base class for the exception. Base your decision as to which one to use upon how you want other parts of the program to treat the exception. If you don't want the exception to be ignored, extend `Exception`. If you don't care whether the exception is ignored, or if you have indicated in your precondition how the exception could be avoided, you might choose to extend `RuntimeException`. In either case, your class will

You can define your own exception class

Use ***StringTokenizer*** to break the string into tokens, then convert each token to a value of the primitive type

StringTokenizer class seen earlier. Then, you apply the method that converts the string to a primitive type value. The following code demonstrates this technique by extracting two integers *x* and *y* from *nextLine*:

```
BufferedReader stdin = new BufferedReader(
    new InputStreamReader(System.in));

String nextLine = stdin.readLine();

 StringTokenizer input = new StringTokenizer(nextLine);
 x = Integer.parseInt(input.nextToken());
 y = Integer.parseInt(input.nextToken());
```

The *Scanner* Class. The *Scanner* class makes it easier to get strings and primitive types from keyboard input, *String* objects, and files. The *Scanner* class is located in the *java.util* package, so any code that uses the *Scanner* class should include the statement

```
import java.util.Scanner;
```

A *Scanner* object can be used to break its input into tokens using a delimiter pattern. The default pattern matches any white space, including blanks, tabs, and carriage returns. This pattern can be set and changed using various methods in the *Scanner* class in conjunction with the *Pattern* class. The *Scanner* class also provides various *next* methods to retrieve tokens from the input and convert them to primitive type values and strings. Here is a brief summary of the more useful *next* methods as described in the Java API:

Method	Description
String next()	Finds and returns the next complete token from this scanner.
boolean nextBoolean()	Scans the next token of the input into a boolean value and returns that value.
double nextDouble()	Scans the next token of the input as a double.
float nextFloat()	Scans the next token of the input as a float.
int nextInt()	Scans the next token of the input as an int.

<code>String nextLine()</code>	Advances this scanner past the current line and returns the input that was skipped.
<code>long nextLong()</code>	Scans the next token of the input as a long.
<code>short nextShort()</code>	Scans the next token of the input as a short.

Note that these methods scan the next token of the input and convert the value to the specified type. If the next token cannot be properly interpreted as the specified type (for example `float` in the case of `nextFloat()`), then an `InputMismatchException` is thrown. These methods will also throw `NoSuchElementException` if the input has been exhausted, and `IllegalStateException` if the scanner is closed.

Suppose that you wanted to compute the sum of integers that you enter at the keyboard. Note that the `Scanner` class does not provide any easy way to detect the end of an input line, so we will use a negative value or zero to indicate the end of the list of integers. The following code accomplishes this task:

```
int nextValue;
int sum=0;
Scanner kbInput = new Scanner(System.in);

nextValue = kbInput.nextInt();
while (nextValue > 0) {
    sum += nextValue;
    nextValue = kbInput.nextInt();
} // end while
kbInput.close();
```

Note the use of the `Scanner` class constructor with `System.in` (of type `InputStream`) to specify that the input will be from the keyboard. The `Scanner` class also provides constructors for the `String` and `File` data types. The method `close` simply closes the `Scanner` object.

If you are concerned that the user might enter a non-integer value in the list, you can use exception handling to react to that error. You can also use the method `hasNextInt`. This method returns `true` if the next token is an integer value, `false` otherwise. Similar methods exist for the other primitive types and strings.

Output

Java provides the methods `print` and `println` to write character strings, primitive types, and objects to the standard output stream `System.out`. The method `println` differs from `print` in that it terminates a line of output so that subsequent output will start on the next line. When the argument is a

The methods `print` and `println` write to an output stream

string, it is simply placed in the output stream. For example, the following program segment uses `println` with a `String` argument:

```
int count = 5;
double average = 20.3;
System.out.println("The average of the " + count
    + " distances read is " + average
    + " miles.");
```

produces the following output:

```
The average of the 5 distances read is 20.3 miles.
```

As we mentioned in the section on strings, the operator `+` can be used to concatenate strings with other strings, primitive types, and objects. Thus, the previous statements concatenate the string `"The average of the "` to the string that represents the value of `count`, and so on.

When `println`'s argument is a primitive type or an object, the static method `valueOf` from the `String` class is used to determine the corresponding string value that is placed on the output stream. For primitive types, this is a simple string representation of the value. For objects, this is ultimately the value returned by the object's `toString` method. Thus, for example, using the method `toString` as defined in the class `Object`, the statements

```
SimpleSphere mySphere = new SimpleSphere();
System.out.println(mySphere);
```

will produce output similar to

```
SimpleSphere@733f42ab
```

You usually override `toString` with your own version. Here is an example that could be used in the class `SimpleSphere`:

```
public String toString() {
    return ("SimpleSphere: radius = " + radius);
} // end toString
```

Now if you execute the statements

```
SimpleSphere mySphere = new SimpleSphere();
System.out.println(mySphere);
```

the output appears as follows:

```
SimpleSphere: radius = 1.0
```

The method
`toString` is
implicitly invoked
when an object is
an argument of
`println`

One of the problems with the `print` and `println` methods is the lack of formatting abilities. Java provides a C-style formatted output method called `printf`. This method uses the new variable arguments feature, and has the following format:

```
printf(String format, Object... args)
```

With the new autoboxing feature, the arguments can also be of a primitive type. The format string may contain fixed text and one or more embedded format specifiers. For example:

```
String name = "Jamie";
int x = 5, y = 6;
int sum = x + y;
System.out.printf("%s, %d + %d = %d", name, x, y, sum);
```

produces the output:

```
Jamie, 5 + 6 = 11
```

In this example, each of the format specifiers has a corresponding argument value that is placed into the format string upon output. The format specifiers in this example are of the simplest form—they start with the `%` character and contain only a conversion character. The conversion characters for common data types are:

Conversion Character	Data Type
b	boolean
s	String – this is also used with objects and the <code>toString</code> method
c	character
d	decimal integer
e	decimal number (formatted in computerized scientific notation)
f	decimal number

A more complete form of the format specifier is as follows:

```
%[width][.precision]conversion
```

The `width` specifies the minimum field width that the value should be printed within. When printing decimal numbers, the `precision` specifies the number of digits of `precision` to be printed after the decimal point. When

using precision with strings, it represents a maximum number of characters. Figure 1-10 shows some examples with the corresponding output.

The `Console` Class

Java 6 introduced the `Console` class to access the character-based console device associated with the current Java virtual machine. Java provides a predefined object of type `Console`, as defined in the package `java.io`, that has many of the same capabilities provided by the Standard streams. This `Console` object can be accessed as follows:

```
Console myConsole = System.console();
```

If the JVM running this code has a console available, it returns a reference to it. But if the JVM does not have a console device available, this call will return `null`. So code such as the following often accompanies an attempt to access the console:

```
if (myConsole == null) {
    System.err.println("No console available.");
    System.exit(1);
} // end if
```

Similar to the `BufferedReader` class, the `Console` class also provides a `readLine()` method to retrieve a line of text from the console. It also defines a second `readLine` method of the form

```
String readLine(String fmt, Object... args)
```

This version provides the ability to create a formatted prompt, and then reads a single line of text from the console. The formatting of the prompt string works much like the `printf` method described earlier in this section.

Output											
Column number											
S	a	r	a								
1	.	1	0	e	+0	4	5				
1	0	1	2	3	.	3	5				
1	0	1	2	3	.	3	4	5	6	9	
1	2	3	4	5	6	7	8	9	10	11	

```
String name = "Sarah";
double y = 10123.34568;
int n = 145;
System.out.printf("%4s\n", name);
System.out.printf("%10.2s\n", name);
System.out.printf("%10d\n", n);
System.out.printf("%10.2e\n", y);
System.out.printf("%10.2f\n", y);
System.out.printf("%5.5f\n", y);
```

FIGURE 1-10

Formatting examples with `printf`

The `Console` class also provides two methods for the input of a user password:

```
char[] readPassword()  
char[] readPassword(String fmt, Object... args)
```

These methods behave similarly to the `readLine` method, but have two important differences. First, when the user enters data at a point in the program where `readPassword` is being executed, it is not echoed back to the console, so it can't be seen by the user (this is what we expect when we type in a password). This is an important feature for helping users maintain password security. Second, the characters typed for the password are entered into a character array, not a `String`. Once the password has been verified, it is strongly suggested that the array holding the password be overwritten with blanks or some other character to minimize the lifetime of the password in memory.

Output to the console is accomplished using the following method:

```
Console printf(String format, Object... args)
```

Alternatively, you can use this method

```
PrintWriter writer()
```

to retrieve a `Printwriter` object that has methods `print` and `println` similar to those used with `System.out`.

The following example shows how the `Console` class could be used to prompt for a username and password.

```
import java.io.Console;  
import java.util.Arrays;  
import java.io.PrintWriter;  
  
public class ConsoleExample {  
  
    static boolean validateLogin(String username,  
                                char[] password) {  
        // Would put code in here to validate the user login  
        return true;  
    } // end validateLogin  
  
    public static void main (String args[]){  
        Console cons = System.console();  
        if (cons == null) {  
            System.err.println("No console available.");  
            System.exit(1);  
        } // end if  
        PrintWriter consOutput = cons.writer();
```

```
String username = cons.readLine("Username: ");
char [] password = cons.readPassword("Password: ");

if (validateLogin(username, password)) {
    // At this point you have validated the password
    consOutput.println("User " + username +
        " successfully logged in");
    // Now wipe out the password from memory
    Arrays.fill(password, ' ');
    // And let the user start his or her session...
} // end if

} // end main
} // end ConsoleExample
```

One interesting thing to note here is that if you try to use the `Console` class from some of the Integrated Development Environments (IDEs) such as Eclipse or NetBeans, the IDE may run the JVM in the background, and hence when you execute the above code you will most likely get the message "No console available." But if you execute the program from the command prompt using the `java` command, it will work as expected.

1.8 File Input and Output

You have used files ever since you wrote your first program. In fact, your Java source program is in a file that you probably created by using a text editor. You can create and access such files outside of and independently of any particular program. Files can also contain data that is either read or written by your program. It is this type of file that concerns us here.

A file is a sequence of components of the same data type that resides in auxiliary storage, often a disk. Files are useful because they can be large and can exist after program execution terminates. In contrast, variables of primitive data types and objects, for example, represent memory that is accessible only within the program that creates them. When program execution terminates, the operating system reuses this memory and changes its contents.

Since files can exist after program execution, they not only provide a permanent record for human users, they also allow communication between programs. Program *A* can write its output into a file that program *B* can use later for input. However, files that you discard after program execution are also not unusual. You use such a file as a scratch pad during program execution when you have too much data to retain conveniently in memory all at once.

It is useful to contrast files with their closest Java relatives, arrays. Files and arrays are similar in that they are both collections of components of the

A file is a sequence
of components of
the same data type

same type. For example, just as you can have an array of elements whose type is `char`, so also can you have a file of elements whose type is `char`. In both cases, the components are characters. However, in addition to the previous distinction between files and all other data types—files can exist after program execution and arrays cannot—files and arrays have two other differences:

- **Files grow in size as needed; arrays have a fixed size.** When you declare an array, you specify its maximum size. Thus, a fixed amount of memory represents the array. A well-written program always checks that an array can accommodate a new piece of data before attempting to insert it. If the array cannot accommodate the data, the program might have to terminate with a message of explanation. You can increase the array size—hopefully by changing the value of a named constant—and compile and run the program again.⁴ On the other hand, if you declare the array's maximum size to be larger than you need, you waste memory. In contrast, the size of a file is not fixed. When the system first creates a file, the file requires almost no storage space. As a program adds data to the file, the file's size increases as necessary, up to the limit of the storage device. Thus, at any given time, the file occupies only as much space as it actually requires. This dynamic nature is a great advantage.
- **Files provide both sequential and random access; arrays provide random access.** If you want the 100th element in the one-dimensional array `x`, you can access it directly by writing `x[99]`; you do not need to look at the elements `x[0]` through `x[98]` first. You could choose, of course, to process an array's elements sequentially, but you would do so by accessing each successive element directly and independently of any other element.

However, you can access elements in a file either directly or sequentially. If you want the 100th element in a file, you can access it directly by position without first reading past the 99 elements that precede it. On the other hand, you could also read all of the first 100 elements one at a time, in sequential order, without specifying any element's position.

Files are classified as follows. A **text file** is a file of characters that are organized into lines. The files that you create—by using an editor—to contain your Java programs are text files. Because text files consist of characters, and accessing characters by position number is usually not convenient, you typically process a text file sequentially. A file that is not a text file is called a **binary file** or sometimes a **general file** or a **nontext file**.

4. Chapter 4 describes resizable arrays. If you reach the end of such an array, you can increase its size during execution. However, this process requires copying the old array into the new array.

A text file contains lines of characters

Files end with a special end-of-file symbol

Text Files

Text files are designed for easy communication with people. As such, they are flexible and easy to use, but they are not as efficient with respect to computer time and storage as binary files.

One special aspect of text files is that they *appear* to be divided into lines. This illusion is often the source of much confusion. In reality, a text file—like any other file—is a sequence of components of the same type. That is, a text file is a sequence of characters. A special end-of-line symbol creates the illusion that a text file contains lines by making the file *behave* as if it were divided into lines. On some systems this end-of-line symbol is simply a carriage return, while on others it consists of a carriage return and line feed character. You need not worry about how your system actually views the end-of-line symbol; this is taken care of by the Java runtime system.

When you create a text file by typing data at your keyboard, each time you press the Enter or Return key, you insert one end-of-line symbol into the file. When an output device, such as a printer or monitor, encounters an end-of-line symbol in a text file, the device moves to the beginning of the next line. In Java, you can specify this end-of-line symbol by using the character `\n`.

In addition, you can think of a special end-of-file symbol that follows the last component in a file. Such a symbol may or may not actually exist in the file, but Java behaves as if one did. Figure 1-11 depicts a text file with these special symbols.

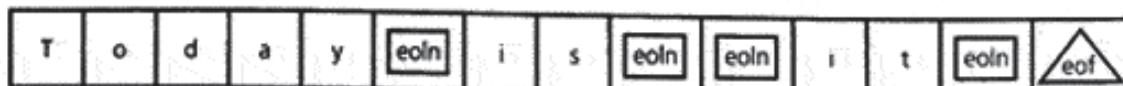
Note that the `Scanner` class presented in the previous section can be used to process text files in a manner very similar to the way we handled input from the keyboard. The `Scanner` class has two constructors that can be used to create `Scanner` objects for input files:

`Scanner(InputStream source)`

Constructs a new `Scanner` that produces values scanned from the specified input stream.

`Scanner(File source)`

Constructs a new `Scanner` that produces values scanned from the specified file.



`eoln` is the end-of-line symbol

`eof` is the end-of-file symbol

FIGURE 1-11

A text file with end-of-line and end-of-file symbols

The first constructor allows for any type of *InputStream*, including *System.in* and objects of the subclass *FileInputStream*. The second constructor is based upon the class *File*. This class is part of the *java.io* package. It provides an abstraction for the file within a program. Instances of the class *File* are not used directly for input and output, but for getting characteristics of a file, such as its access mode. The first constructor creates a scanner for the specified file using the *File* class. Here is a simple example that uses the *Scanner* class to read a first name, last name, and age from each line of a file called *Ages.dat* and prints it to standard output:

```
String fname, lname;
int age;
Scanner fileInput;
File inFile = new File("Ages.dat");

try {
    fileInput = new Scanner(inFile);

    while (fileInput.hasNext()) {
        fname = fileInput.next();
        lname = fileInput.next();
        age = fileInput.nextInt();
        age = fileInput.nextInt();
        System.out.printf("%s %s is %d years old.\n",
                           fname, lname, age);
    } // end while

    fileInput.close();

} // end try
catch (FileNotFoundException e) {
    System.out.println(e);
} // end catch
```

Note that here the *hasNext* method is used to determine if the end-of-file symbol has been reached—when the end-of-file is reached, there are no more tokens in the file to be processed. Also, the code above would need to import the classes *File* and *FileNotFoundException* from the package *java.io*, and the *Scanner* class from the package *java.util*.

Alternatively, text input files can be processed using the classes *FileInputStream* and *FileReader*. Output files are also supported by two main classes: *FileOutputStream* and *PrintWriter*. When using these classes, actual read or write access to the file is done through streams. A variety of tasks related to processing files using streams are now presented.

Opening a stream to a file. Before you can read from or write to a file, you need to open a stream to the file. That is, you need to create a stream instance.

Use streams to access a file

You must initialize, or open, a stream before you can use it

One way to open a stream to a file for reading is to use the class *FileReader* and provide the file's name when you declare the file stream. For example,

```
FileReader inStream = new FileReader("Ages.dat");
```

declares an input stream variable *inStream* and associates it with the file named *Ages.dat*. The file name can be either a literal constant, as it is here, or a string variable.

Alternatively, you can use an instance of *File* by writing

```
File inFile = new File("Ages.dat");
FileReader inStream = new FileReader(inFile);
```

Unfortunately, the methods available from the class *FileReader* do not lend themselves very well to text processing. Because of this, the stream instance is usually embedded within an instance of the class *BufferedReader*. (This is the same class that we used to read input from the keyboard.) *BufferedReader* provides the method *readLine* for obtaining a line of text as a *String* object. A line is considered to be terminated by an end-of-line character, as we mentioned earlier. Here is an example of opening a stream to a file and adding the functionality of the class *BufferedReader*:

```
FileReader fr = new FileReader("Ages.dat");
BufferedReader input = new BufferedReader(fr);
```

Often, this is combined into a single statement:

```
BufferedReader input = new BufferedReader(
    new FileReader("Ages.dat"));
```

The method **readLine** reads a line of text as a string

If the file is not found, an exception is thrown

Note that the *FileReader* constructor will throw the exception *FileNotFoundException* if the file is not found. Since this is a checked exception, the statement must be enclosed in a *try* block. Therefore, the actual code used to open a stream to a text file would be similar to the following:

```
BufferedReader input;
try {
    input = new BufferedReader(new FileReader("Ages.dat"));
    // read data from file
} // end try
catch (FileNotFoundException e) {
    e.printStackTrace();
    System.exit(1); // File not found so exit
} // end catch
```

Now, using the instance of *BufferedReader*, data can be read from the file. As with keyboard input, you can use a *StringTokenizer* to break up the

string returned by `readLine` into tokens for easier processing. But how do you know when you have read all of the data in the file? `BufferedReader` provides a method `ready` that determines whether the underlying character stream is ready. This method returns a boolean value that can be used in a `while` loop to determine whether more data is available in the file, as follows:

```
 StringTokenizer line;
while (input.ready()) {
    line = new StringTokenizer(input.readLine());
    // process line of data
    ...
}

} // end while
```

You can also detect when the end of the file is reached by checking whether the method `readLine` returns `null`. For example, the following loop will process all of the lines in the file:

```
 StringTokenizer line;
String inputLine;
while ((inputLine = input.readLine()) != null) {
    line = new StringTokenizer(inputLine);
    // process line of data
    ...
}

} // end while
```

The method `readLine` can throw the exception `IOException`, another checked exception. Also, `readLine` must appear within the same `try` block that creates the `BufferedReader` instance; otherwise, the compiler won't be able to verify that the instance has been initialized properly. One way you can handle this is simply to add another `catch` block for the `IOException` to the `try` statement. Or, since `FileNotFoundException` is a subclass of `IOException`, you can use a single `catch` block as follows:

```
 BufferedReader input;
 StringTokenizer line;
 String inputLine;
 try {
    input = new BufferedReader(new FileReader("Ages.dat"));
    while ((inputLine = input.readLine()) != null) {
        line = new StringTokenizer(inputLine);
        // process line of data
        ...
    }
 } // end try
```

The method `ready` can be used to determine whether the file contains more data

```

    catch (IOException e) {
        System.out.println(e);
        System.exit(1); // I/O error, exit the program
    } // end catch
}

```

File output. To write text to a file, you need to open an output stream to the file. One way to open a file for writing is to use the class *FileWriter* and provide the file's name when you declare the file stream. For example,

```
FileWriter outStream = new FileWriter("Results.dat");
```

declares an output stream variable *outStream* and associates it with the file name *Results.dat*. The file name can be a literal constant, as it is here, or a string variable. If the file *Results.dat* does not exist, a new empty file with this name is created. If the file *Results.dat* already exists, opening it erases⁵ the data in the file.

Like *FileReader*, the *FileWriter* class itself does not provide useful methods for writing data to the file. Another class, *PrintWriter*, provides two methods, *print* and *println*. These methods are already familiar to you; they are the same methods used by *System.out*. Here is a simple example of writing data to a file:

```

try {
    PrintWriter output = new PrintWriter(
        new FileWriter("Results.dat"));
    output.println("Results of the survey");
    output.println("Number of males: " + numMales);
    output.println("Number of females: " + numFemales);

    // other code and output appears here...
} // end try

catch (IOException e) {
    System.out.println(e);
    System.exit(1); // I/O error, exit the program
} // end catch

```

Closing a file. When you have finished using a file, you should close the stream associated with that file. To close a stream (input or output), you use the method *close* as follows:

```
myStream.close();
```

The file associated with this stream is no longer available for input or output until you open it again.

When you are finished using a file, call **close** to close the stream

5. The data might not actually be erased, but the file will behave as if it were empty.

Adding to a text file. When you open a stream to a file for writing, you can specify a second argument in addition to the file's name to indicate whether the file should be replaced or appended. If this second argument to the `FileOutputStream` constructor is `true`, the file is appended rather than replaced. For example,

```
PrintWriter ofStream = new PrintWriter(
    new FileOutputStream("Results.dat", true));
```

You can append
data to a file

This retains the old contents of the file `Results.dat`, and you can write additional components.

Copying a text file. Suppose that you wanted to make a copy of the text file associated with the stream variable `original`. Copying a text file requires some work and provides a good example of the statements you have just studied. The approach taken by the following method copies the file one line at a time.

```
public static void copyTextFile(String originalFileName,
                                String copyFileName) {
    // -----
    // Makes a duplicate copy of a text file.
    // Precondition: originalFileName is the name of an existing
    // external text file, and copyFileName is the name of the
    // text file to be created.
    // Postcondition: The text file named copyFileName is a
    // duplicate of the file named originalFileName.
    // -----
    BufferedReader ifStream = null;
    PrintWriter ofStream = null;

    try {
        ifStream = new BufferedReader(
            new FileReader(originalFileName));
        ofStream = new PrintWriter(new FileWriter(copyFileName));
        String line;

        // copy lines one at a time from given file
        // to new file
        while ((line = ifStream.readLine()) != null) {
            ofStream.println(line);
        } // end while
    } // end try
    catch (IOException e) {
        System.out.println("Error copying file");
    } // end catch
```

```

        finally {
            try {
                ifStream.close(); // close the files
                ofStream.close();
            } // end try
            catch (IOException e) {
                e.printStackTrace();
            } // end catch
        } // end finally
    } // end copyTextFile
}

```

The `finally` block allows the files to be closed regardless of whether an exception is thrown. It is executed after correct execution of the `try` block or after an exception is handled in a `catch` block.

Searching a text file sequentially. Suppose that you have a text file of data about a company's employees. For simplicity, assume that this file contains two consecutive lines for each employee. The first line contains the employee's name, and the next line contains data such as salary.

Given the name of an employee, you can search the file for that name and then determine other information about this person. A sequential search examines the names in the order in which they appear in the file until the desired name is located. The following method performs such a sequential search, given a class to represent a person:

```

public class Person {
    private String name;
    private double salary;

    public Person(String n, double s) {
        name = n;
        salary = s;
    } // end constructor
    // other methods appear here
} // end Person

public static Person searchFileSequentially(
    String fileName, String desiredName) {
    // -----
    // Searches a text file sequentially for a desired person.
    // Precondition: fileName is the name of a text file of
    // names and data about people. Each person is represented
    // by two lines in the file: The first line contains the
    // person's name, and the second line contains the person's
    // salary. desiredName is the name of the person sought.
    // Postcondition: If desiredName was found in the file,
    // a Person object that contains the person's
}

```

name and data is returned. Otherwise, the value null is returned to indicate that the desiredName was not found. The file is unchanged and closed.

```
-----  
BufferedReader ifStream = null;  
String nextName = null;  
String nextSalary = null;  
boolean found = false;  
  
try {  
    ifStream = new BufferedReader(new FileReader(fileName));  
    while (!found &&  
        (nextName = ifStream.readLine()) != null) {  
        nextSalary = ifStream.readLine();  
        if (nextName.compareTo(desiredName) == 0) {  
            found = true;  
        } // end if  
    } // end while  
} // end try  
catch (IOException e) {  
    System.out.println("Error processing file");  
    return null;  
} // end catch  
finally {  
    if(ifStream != null) {  
        try {  
            ifStream.close(); // close the file  
        } // end try  
        catch (IOException e) {  
            System.out.println("Error closing file");  
        } // end catch  
    } // end finally  
    if (found) {  
        return new Person(nextName,  
                           Double.parseDouble(nextSalary));  
    }  
    else {  
        return null;  
    } // end if  
} // end searchFileSequentially
```

This method needs to look at all the names in the file before determining that a particular name does not occur. If the names were in alphabetical order, you could determine when the search had passed the place in the file that should have contained the desired name, if it existed. In this way, you could terminate the search before you needlessly searched the rest of the file.

Object Serialization

In the method `searchFileSequentially`, we assumed that all of the information about a person had been placed in a text file in a very specific format. In that example, the data fields were strings and primitive values, so it was simply a matter of writing the name data field on one line and the salary on the next line. But what about situations in which the data is more complex? For example, suppose that the `Person` class also kept track of the person's dependents by using an ADT list.⁶ To save this information to a text file involves a more complicated scheme, since we may not know beforehand how many dependents an employee has.

When data is stored to a file for later use by the same program or another program, it is called **data persistence**. Normally, any information stored in the various variables and data structures in a program is lost when the program terminates execution. In many cases, however, it is desirable to save the data to a file for later retrieval before terminating the program. Java provides a mechanism for creating persistent objects, called **object serialization**. Serialization is the process of transforming an object into a sequence of bytes that represents the object. Deserialization is the process of transforming a sequence of bytes back into an object. Once an object is serialized, it can be stored in a file and read back at a later time using deserialization.

Any object that is to be saved using object serialization must implement the interface `java.io.Serializable`. This interface is somewhat unique in that it contains no methods. It is used to signal the compiler that the instances of this class may need to have their state serialized or deserialized.

One interesting aspect of object serialization is that when an object is serialized, all objects that it references are also serialized, as long as the referenced objects are instances of a class that implements the `Serializable` interface. For example, suppose that the following is the `Person` class described earlier:

```
import java.io.Serializable;

public class Person implements Serializable {
    private String name;
    private double salary;
    private Person[] dependents;
    private int numDepend = 0;

    public Person(String n, double s) {
        name = n;
        salary = s;
        // assume that ListArrayBased also implements the
        // Serializable interface
        dependents = new Person[25];
    } // end constructor
```

Object serialization
transforms an object
into a sequence of
bytes

6. Chapter 4 introduces the ADT list.

```
public void addDependent(Person p) {
    numDepend++;
    dependents[numDepend] = p;
} // end addDependent

public String getName() {
    return name;
} // end getName

// other methods for class appear here
} // end Person
```

When an instance of the *Person* class is serialized, all of the referenced objects are also serialized—the *String* object *name* and the list *dependents*. You accomplish the actual serialization of an object by using the *writeObject* method of the stream class *ObjectOutputStream*. Much like *PrintWriter*, *ObjectOutputStream* adds functionality to *FileOutputStream*. The following statements save a *Person* object *p* to a file *EmployeeDB.dat*:

```
ObjectOutputStream ooStream = new ObjectOutputStream(
    new FileOutputStream("EmployeeDB.dat"));
ooStream.writeObject(p);
```

When the object is deserialized, both it and the objects it originally referenced will be restored to their original state. To do this, use the *readObject* method of the stream class *ObjectInputStream*. Like the *BufferedReader*, *ObjectInputStream* adds functionality to *FileInputStream*. The following statements retrieve a *Person* object *p* from a file *EmployeeDB.dat*:

```
ioStream = new ObjectInputStream(
    new FileInputStream("EmployeeDB.dat"));
nextPerson = (Person)ioStream.readObject();
```

The following method demonstrates how the file could be searched sequentially for a particular person (this method parallels the method given for text files):

```
public static Person searchFileSequentially(
    String fileName, String desiredName) {
    // -----
    // Searches a text file sequentially for a desired person.
    // Precondition: fileName is the name of a binary file
    // of Person objects. desiredName is the name of the person
    // sought.
    // Postcondition: If desiredName was found in the file,
    // a Person object that contains the person's
```

```
// name and data is returned. Otherwise, the value null
// is returned to indicate that desiredName was not
// found. The file is unchanged and closed.
// -----
ObjectInputStream ioStream = null;
Person nextPerson = null;
boolean found = false;

try {
    ioStream = new ObjectInputStream(
        new FileInputStream(fileName));
    while (!found && (nextPerson =
        (Person)ioStream.readObject()) != null) {
        if (nextPerson.getName().compareTo(desiredName) == 0) {
            found = true;
        } // end if
    } // end while
} // end try
catch (IOException e) {
    System.out.println("Error processing file");
    return null;
} // end catch
catch (ClassNotFoundException e) {
    System.out.println("Unexpected object type in file");
    return null;
} // end catch
finally {
    //Close the ObjectInputStream
    try {
        if (ioStream != null) {
            ioStream.close();
        }
    } catch (IOException ex) {
        ex.printStackTrace();
    } // end catch
} // end finally
if (found) {
    return nextPerson;
}
else {
    return null;
} // end if

} // end searchFileSequentially
```

1. Each comment line in Java begins with two slashes (//) and continues until the end of the line.
2. A Java identifier is a sequence of letters, digits, underscores, and dollar signs that must begin with either a letter or an underscore.
3. The primitive data types in Java are organized into four types: integer, character, floating point, and boolean.
4. A Java reference is used to locate an object. When an object is created using the new operator, the location of the object in memory is returned and can be assigned to a reference variable.
5. You define named constants by using a statement of the form

```
final type-identifier = value;
```

6. Java uses short-circuit evaluation for expressions that contain the logical operators `&&` (and) and `||` (or). That is, evaluation proceeds from left to right and stops as soon as the value of the entire expression is apparent.
7. An array is a collection of references that have the same data type. You can refer to the elements of an array by using an index that begins with zero. First the array must be instantiated with the number of elements desired. Then you can assign the references of the array an object.
8. The general form of the if statement is

```
if (expression)
    statement1
else
    statement2
```

If *expression* is true, *statement₁* executes; otherwise, *statement₂* executes.

9. The general form of the switch statement is

```
switch (expression) {
    case constant1:
        statement1
        break;
    .
    .
    .
    case constantn: case constantn+1:
        statementn
        break;
    .
    .
    .
}
```

10. The general form of the `while` statement is

```
while (expression)
    statement
```

As long as *expression* is true, *statement* executes. Thus, it is possible that *statement* will never execute.

11. The general form of the `for` statement is

```
for (initialize; test; update)
    statement
```

where *initialize*, *test*, and *update* are expressions. Typically, *initialize* is an assignment expression that occurs only once. Then if *test*, which is usually a logical expression, is true, *statement* executes. The expression *update* executes next, usually incrementing or decrementing a counter. This sequence of events repeats, beginning with the evaluation of *test*, until *test* is false.

12. The enhanced `for` loop makes it easier to process arrays. The general form of this loop is:

```
for (ArrayType variableName: arrayName)
    statement
```

13. The general form of the `do` statement is

```
do
    statement
while (expression);
```

Here, *statement* executes until the value of *expression* is false. Note that *statement* always executes at least once.

14. The filename for a Java source code file has the same name as the class it contains, with `.java` appended to the end.
15. Java packages provide a mechanism for grouping related classes. To indicate that a class is part of a package, you include a `package` statement as the first program line of your code.
16. To use classes contained in other packages, you must include an `import` statement before the class definition. The format of the `import` statement is

```
import package-name.class-name;
```

17. An object in Java is an instance of a class. A class can be thought of as a data type that specifies the data and methods that are available for instances of the class. A class definition includes an optional subclassing modifier, an optional access modifier, the keyword `class`, an optional `extends` clause, an optional `implements` clause, and a class body.
18. Data fields are class members that are either variables or constants. Data field declarations can contain modifiers that control the availability of the data field (access modifiers) or that modify the way the data field can be used (use modifiers).

19. Methods are used to implement object behaviors. The general form of a method definition is:

```
access-modifier use-modifier type name(formal-parameter-list) {  
    body  
}
```

A valued method returns a value by using the `return` statement. A `void` method can use `return` to exit.

20. When you invoke a method, the actual arguments must correspond to the formal parameters in number, order, and type.
21. A method makes local copies of the values of any arguments that are passed. Thus, the arguments remain unchanged by the method. When the argument is a reference, a method can modify the object it references, but not the value of the reference variable itself.
22. Members of a class should be declared as `public` or `private`. The client of the class—that is, the program that uses the class—cannot use members that are `private`. However, the implementations of methods within the class implementation can use them. Typically, you should make the data fields of a class `private` and provide `public` methods to access some or all of the data fields.
23. You can access data fields and methods that are declared `public` by naming the object, followed by a period, followed by the member name.
24. A Java class contains at least one constructor, which is an initialization method.
25. If you do not define any constructors for a class, the compiler will generate a default constructor—that is, one without parameters—for you.
26. Inheritance allows a new class to be defined based on the data fields and methods of an existing class while adding its own functionality. This enhances our ability to reuse code.
27. A class that is derived from another class is called the derived class or subclass. The class from which the subclass is derived is called the base class or superclass.
28. When defining a subclass, the class name is followed by an `extends` clause that names the superclass. If there is no `extends` clause, the class is implicitly a subclass of `Object`.
29. The `equals` method defined in the class `Object` is based on reference equality; it simply checks to see if two references refer to the same object. This is known as shallow equality.
30. It is common for a class to redefine the `equals` method for deep equality—in other words, to check the equality of the contents of the objects.
31. The `Array` class contains various static methods for manipulating arrays.
32. A string is a sequence of characters. The `String` class supports nonmutable strings, while the `StringBuffer` class supports mutable strings. In the `String` class, you can access the entire string, a substring, or the individual characters. In the `StringBuffer` class, you can access and actually manipulate the entire string, a substring, or the individual characters.

33. Exceptions are used to handle errors during execution. A method indicates that an error has occurred by throwing an exception. When an exception occurs, the statements within the `catch` block that correspond to the exception are executed.
34. The method `System.out.println` places a value into an output stream. Reading a value from an input stream is easier when the `Scanner` class is used.
35. In Java, files are accessed using the `Scanner` class or streams.
36. The `Console` class provides an alternative way to get input and output from the console of the current program execution environment.
37. Data persistence is supported in Java through object serialization. You serialize an object by using the method `writeObject` from the stream class `ObjectOutputStream`, and you deserialize an object by using the method `readObject` from the stream class `ObjectInputStream`.

Cautions

1. Remember that `=` is the assignment operator; `==` is the equality operator.
2. Do not begin a decimal integer constant with zero. A constant that begins with zero is either an octal constant or a hexadecimal constant.
3. Without a `break` statement, execution of a `case` within a switch statement will continue into the next `case`.
4. You must be careful that an array index does not exceed the size of the array. Java will throw the exception `ArrayIndexOutOfBoundsException` if an index value is less than zero or greater than or equal to the length of the array.
5. If you define a constructor for a class but do not also define a default constructor, the compiler will not generate one for you. In this case, a statement such as

```
MyNewClass test = new MyNewClass();
```

is illegal.

6. When using an IDE, the `Console` object is often not accessible since the JVM executes as a background process.
7. Opening an existing file for output erases the data in the file, unless you specify append mode.

Self-Test Exercises

1. To use each of the following Java classes in your program, indicate an `import` statement that would allow the program to use each of the following methods. If one is not needed, then state so. You may need to do a little research to determine the appropriate package.
 - a. `static int round(float a)` in the class `Math`
 - b. `void println(String x)` in the class `PrintWriter`
 - c. `boolean isEmpty()` in the class `Vector`
 - d. `int getErrorCode()` in the class `SQLException`

2. What are the differences between the three types of comments in Java?
3. The syntax of a method declaration is as follows:

```
access-modifier use-modifiers return-type  
    method-name (formal-parameter-list) {  
        // method-body  
    }
```

What are the possible values for *access-modifier* and *use-modifier*?

4. Using the *SimpleSphere* class shown in Figure 1-5, and the following declarations, are the statements below correct or will they generate a compiler error? If they will generate a compiler error, explain why.

```
SimpleSphere myBall = new SimpleSphere(4.695);
```

- a. `myBall.radius = 5.0;`
- b. `int rad = myBall.getRadius();`
- c. `float d = myBall.getDiameter();`
- d. `myBall.DEFAULT_RADIUS = 5.0;`

5. What is meant by "short circuit operator" in a boolean expression? Give an example.
6. What is the difference between checked exceptions and unchecked exceptions?

Exercises

1. Suppose we have the following declarations:

```
int i, j;  
float x, y;  
double u, v;
```

Determine if the following assignments are *valid* or *invalid*. If they are invalid, explain why.

- a. `i = x;`
- b. `j = (int)u / i;`
- c. `u = j / i * v;`
- d. `x = u * i + x;`

2. Evaluate the following expressions:

- a. $4 + 3 * 11 / 2.0 = (-2)$
- b. $4.6 - 2.0 + 3.2 - 1.1 * 2$
- c. $23 \% 4 - 23 / 4$
- d. $12 / 3 * 2 + (\text{int})(2.5 * 10)$

3. Rewrite the following *if* statement using a *switch* statement. If it cannot be done, explain why. Assume the variables *sumA*, *sumB*, and *sumC* have been declared and initialized.

```
int n = result;
if (n > 0 && n <= 3)
    sumA++;
else if (n == 7 || n == 11)
    sumB++;
else
    sumC++;
```

4. Rewrite the following *switch* statement using an *if* statement(s).

```
switch (ans) {
    case 'Y': case 'y': x = x + 1;
                break;
    case 'N': case 'n': x = x - 1;
                break;
    default: x = 0;
}
```

5. What is the problem with the following code?

```
if (amount = 0) {
    System.out.println("Sorry, there are none left");
}
```

6. Given the following *if* statement:

```
if (x <= 0) {
    if (x <= 100)
        System.out.println("Statement A");
    else
        System.out.println("Statement B");
}
else {
    if (x > 10)
        System.out.println("Statement C");
    else
        System.out.println("Statement D");
}
```

Using relational operators, give the range of values for *x* that produce the following output:

- a. Statement A
- b. Statement B
- c. Statement C
- d. Statement D

7. Write array declarations for the following:

- Create an array called `firstname` capable of holding 15 first names.
- Create an array called `temp` capable of holding the high temperature for every day of a given month.
- Create an array called `finalAve` capable of holding the final averages of 50 students.
- Create an array called `finalLetter` capable of holding the final letter grade for `num` students.

8. What is the output of the following statement?

```
System.out.println("John said \"It should be located in" +
    "C:\\myfiles\\\" \\n in a worried tone.\"");
```

9. For each set of following statements, indicate the number of times the statement `System.out.print("x");` is executed. If it is an infinite loop, indicate so.

a. <pre>x = 12; while (x > 0) { System.out.print("x"); x = x - 2; } // end while</pre>	b. <pre>x = 3; do { System.out.print("x"); x--; } while (x < 0);</pre>
c. <pre>x = 5; while (x > 0) { System.out.print("x"); } // end while</pre>	d. <pre>x = 3; do { System.out.print("x"); x = x + 2; } while (x <= 9);</pre>
e. <pre>for (i = 0; i <= 99; i++) System.out.print("x");</pre>	f. <pre>for (i = 84; i <= 96; i++) for (j = 7; j < 10; j++) System.out.println("x");</pre>

10. Consider the following method that is intended to swap the values of two integer numbers:

```
public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
} // end swap

// In other code assume x=5 and y=10, and we have the
// following call
swap(x, y);
```

Why doesn't the method actually swap the contents of `x` and `y`? Provide a detailed explanation.

11. Given a class *Pet* as started in the following example, add two constructors—one to create pets with a name, the other to create pets with a name and an age.

```
class Pet {
    private String name;
    private int age;
    // add constructors here
```

12. Suppose you have the following class:

```
class Second {
    private int x;
    public int z;

    public int sum() {
        return x + y + z;
    } // end sum

    private void reset(int a, int b, int c) {
        x = a; y = b; c = z;
    } // end reset

    public boolean check(float x) {
        return x < 0;
    } // end check
} // end Second
```

Given the following declaration,

```
Second myClass = new Second();
```

indicate for each statement (which might appear in testing code) if it is *legal* or *illegal* (will cause an error).

- `myClass.x = 5;`
- `myClass.z = 5;`
- `myClass.sum(x);`
- `int ans = myClass.sum();`
- `myClass.reset(1, 2, 3);`
- `boolean x = myClass.check(11.2);`

13. Given the following class *Complex*, complete the following questions:

```
class Complex {
    private int real;
    private int imaginary;
    public Complex(int r, int i) {
        real = r;
        imaginary = i;
    } // end constructor
```

```
public String toString() {
    return real + " + " + imaginary + "i";
} // end toString
} //end class Complex
```

- a. Write a statement that creates a complex number $3 + 2i$ called c1.
 - b. Write a statement that creates a complex number $4 - 5i$ called c2.
 - c. Write a statement that prints a complex number called c1.
 - d. For the class Complex, modify the `toString` method so that if the real or imaginary part is zero, it is not placed in the string. If both are zero, then just print zero. Finally, if the imaginary part is 1 or -1, simply print + 1 instead of + 1i and - 1 instead of - 1i.
 - e. For the class Complex, add the following methods:

```
public Complex add(Complex val)
// returns a Complex number whose value is (this + val)
public Complex subtract(Complex val)
// returns a Complex number whose value is (this - val)
public Complex multiply(Complex val)
// returns a Complex number whose value is (this * val)
```
 - f. Add a `main` program with test code that demonstrates that the above methods are working properly.
14. Write a class `Address` that contains the street, city, and zip code. Provide one or more methods to initialize these values, and a method called `toString` that returns a `String` representation that contains all of these values.
15. What is the output of the following statements? Assume each group is independent. If the result is an empty string, state so.
- a. `String str = "Hello World!";
str.substring(6, 10);`
 - b. `String str = "Hello World!";
str.substring(0, 1);`
 - c. `String str = "Hello World!";
System.out.println(str.toLowerCase().getCharAt(0));`

16. Given this code segment,

```
try {
    // statements appear here...
}
catch (IOException ex) {
    System.out.println("I/O error!");
}
catch (NumberFormatException ex) {
    System.out.println("Bad input!");
}
```

```
    finally {
        System.out.println("Finally!");
    }
System.out.println("Done!");
```

- what will be printed if an *FileNotFoundException* occurs in the *try* block?
- what will be printed if an *ArrayIndexOutOfBoundsException* occurs in the *try* block?
- what will be printed if no exception occurs in the *try* block?
- what would happen if the following *catch* clause was added as the first *catch* clause in the code?

```
catch (Exception ex) {
    System.out.println("Error!");
}
```

17. Write a small program that allows a user to change his or her password using the *Console* class. As in the example in the text, assume that there is a method *validateLogin* already created that validates the login, but simply returns *true*. Also assume there is a method *resetPassword* that would reset the password in the system, but for now, simply returns *true*.

Your program should require a new password to have at least six characters with at least one character that is not a letter (upper case or lower case). Also, be sure to prompt the user for the password twice and make sure the responses match. If they don't match, the user should start the process of entering a new password again. This reduces the possibility that the user mistypes the new password. Remember, you may need to run this program from a command window.

Programming Problems

1. Create an application called *Registrar* that has the following classes:

A *Student* class that minimally stores the following data fields for a student:

- name
- student id number
- number of credits
- total grade points earned

The following methods should also be provided:

- A constructor that initializes the name and id fields
- A method that returns the student name field
- A method that returns the student ID field
- A method that determines if two student objects are equal if their student id numbers are the same (*override equals* from the class *Object*)
- Methods to set and retrieve the total number of credits
- Methods to set and retrieve the total number of grade points earned
- A method that returns the GPA (grade points divided by credits)

An *Instructor* class that minimally stores the following data fields for an instructor:

- name
- faculty id number
- department

The following methods should also be provided:

- A constructor that initializes the name and id fields
- Methods to set and retrieve the instructor's department

A *Course* class that minimally stores the following data for a course:

- name of the course
- course registration code
- maximum number of 35 students
- instructor
- number of students
- students registered in the course (an array)

The following methods should also be provided:

- A constructor that initializes the name, registration code, and maximum number of students
- Methods to set and retrieve the instructor
- A method to search for a student in the course; the search should be based on an ID number.
- A method to add a student to the course. If the course is full, then an exception with an appropriate message should be raised (try creating your own exception class for this). Also, be sure that the student is not already registered in the course. The list of students should be in the order that they registered.
- A method to remove a student from the course. If the student is not found, then an exception with an appropriate message should be raised (use the same exception class mentioned above).
- A method that will allow Course objects to be output to a file using object serialization
- A method that will allow Course objects to be read in from a file created with Object serialization

You will note that the *Student* and *Instructor* classes described above have some commonality. Create a *Person* class that captures this commonality and uses it as a base class for *Student* and *Instructor*. This class should be responsible for the *name* and *id* fields and also provide a *toString* method that returns a string of the form *name, id*. This will be the inherited *toString* method for the *Student* and *Instructor* classes.

- a. Draw a UML diagram for this application.
- b. Implement the previous classes in Java. Write a main program that can serve as a test class that tests all of the methods created and demonstrates that they are working.

- c. Write a second main program that provides a menu to allow the user to:
 - i. create a course, prompting the user for all of the course information.
 - ii. add students to the course.
 - iii. check to see if a student is registered in the course; and
 - iv. remove a student from the course.
- d. Add to the previous menu the ability to save a course using object serialization. Also add a menu choice to read in a course from a file given the course code. Come up with a system of naming the file so that the user need only be asked the course code to load the course information from a file.

CHAPTER 2

Principles of Programming and Software Engineering

This chapter summarizes several fundamental principles that serve as the basis for dealing with the complexities of large programs. The discussion both reinforces the basic principles of programming and demonstrates that writing well-designed and well-documented programs is cost-effective. The chapter also presents a brief discussion of algorithms and data abstraction and indicates how these topics relate to the book's main theme of developing problem-solving and programming skills. In subsequent chapters, the focus will shift from programming principles to ways of organizing and using data. Even when the focus of discussion is on these new techniques, you should note how all solutions adhere to the basic principles discussed in this chapter.

2.1 Problem Solving and Software Engineering

- What Is Problem Solving?
- The Life Cycle of Software
- What Is a Good Solution?

2.2 Achieving an Object-Oriented Design

- Abstraction and Information Hiding
- Object-Oriented Design
- Functional Decomposition
- General Design Guidelines
- Modeling Object-Oriented Designs
- Using UML
- Advantages of an Object-Oriented Approach

2.3 A Summary of Key Issues in Programming

- Modularity
- Modifiability
- Ease of Use
- Fail-Safe Programming
- Style
- Debugging

- Summary
- Cautions
- Self-Test Exercises
- Exercises
- Programming Problems

2.1 Problem Solving and Software Engineering

Where did you begin when you wrote your last program? After reading the problem specifications and procrastinating for a certain amount of time, most novice programmers simply begin to write code. Obviously, their goal is to get their programs to execute, preferably with correct results. Therefore, they run their programs, examine error messages, insert semicolons, change the logic, delete semicolons, pray, and otherwise torture their programs until they work. Most of their time is probably spent checking both syntax and program logic. Certainly, your programming skills are better now than when you wrote your first program, but will you be able to write a really large program by using the approach just described? Maybe, but there are better ways.

Realize that an extremely large software development project generally requires a team of programmers rather than a single individual. Teamwork requires an overall plan, organization, and communication. A haphazard approach to programming will not serve a team programmer well and will not be cost-effective. Fortunately, an emerging engineering field related to a branch of computer science—**software engineering**—provides techniques to facilitate the development of computer programs.

Whereas a first course in computer science typically emphasizes programming issues, the focus in this book will be on the broader issues of problem solving. This chapter begins with an overview of the problem-solving process and the various ways of approaching a problem.

What Is Problem Solving?

Here the term **problem solving** refers to the entire process of taking the statement of a problem and developing a computer program that solves that problem. This process requires you to pass through many phases, from gaining an understanding of the problem to be solved, through designing a conceptual solution, to implementing the solution with a computer program.

Exactly what is a solution? Typically, a **solution** consists of two components: algorithms and ways to store data. An **algorithm** is a step-by-step specification of a method to solve a problem within a finite amount of time. One action that an algorithm often performs is to operate on a collection of data. For example, an algorithm may have to put new data into a collection, remove data from a collection, or ask questions about a collection of data.

Perhaps this description of a solution leaves the false impression that all the cleverness in problem solving goes into developing the algorithm and that how you store your data plays only a supporting role. This impression is far from the truth. You need to do much more than simply store your data. When constructing a solution, you must organize your data collection so that you can operate on the data easily in the manner that the algorithm requires. In fact, most of this book describes ways of organizing data.

Coding without a solution design increases debugging time

Software engineering facilitates development of programs

A solution specifies algorithms and ways to store data

When you design a solution to a given problem, you can use several techniques that will make your task easier. This chapter introduces those techniques, and subsequent chapters will provide more detail.

The Life Cycle of Software

The development of good software involves a lengthy and continuing process known as the software's life cycle. This process begins with an initial idea, includes the writing and debugging of programs, and continues for years to involve corrections and enhancements to the original software. Figure 2-1 pictures the nine phases of the software life cycle as segments on a water wheel.¹ This arrangement suggests that the phases are part of a cycle and are not simply a linear list. Although you start by specifying a problem, typically you move from any phase to any other phase. For example, testing a program can suggest changes to either the problem specifications or the solution design. Also notice that the nine phases surround a documentation core in the figure. Documentation is not a separate phase, as you might expect. Rather, it is integrated into all phases of the software life cycle.

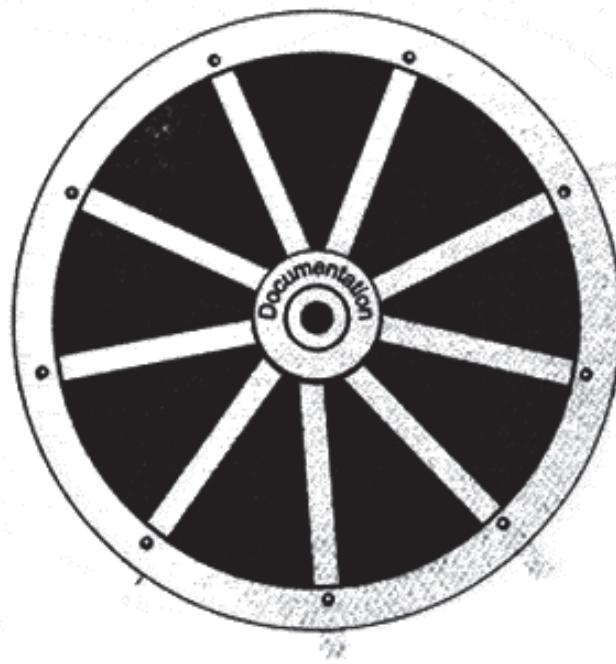


FIGURE 2-1

The life cycle of software as a water wheel that can rotate from one phase to any other phase

1. Thanks to Raymond L. Paden for suggesting that the "wheel" be a "water wheel."

Within the last few years, incremental and iterative development methods have emerged. These methods apply the first seven phases (specification, design, risk analysis, verification, coding, testing, and refinement) incrementally in a circular pattern. The refinement phase is where the next changes (or refinements) to the system are considered, leading the development back to the specification phase. Using this approach, a portion of the overall system is developed initially, and then refinements to the solution are incorporated. Once the system is complete, it then moves to the production and maintenance phases. When using an object-oriented language such as Java, this means that the initial development may involve building a subset of objects, then incrementally enhancing these objects and adding new objects until the system is complete and ready for production.

Here, then, are the phases in the life cycle of typical software. Although all phases are important, only those that are most relevant to this book are discussed in detail.

Phase 1: Specification. Given an initial statement of the software's purpose, you must specify clearly all aspects of the problem. Often the people who describe the problem are not programmers, so the initial problem statement might be imprecise. The specification phase, then, requires that you bring precision and detail to the original problem statement and that you communicate with both programmers and nonprogrammers.

Here are some questions that you must answer as you write the specifications for the software: What is the input data? What data is valid and what data is invalid? Who will use the software and what user interface should be used? What error detection and error messages are desirable? What assumptions are possible? Are there special cases? What is the form of the output? What documentation is necessary? What enhancements to the program are likely in the future?

One way to improve communication among people and to clarify the software specifications is to write a **prototype program** that simulates the behavior of portions of the desired software product. For example, a simple—even inefficient—program could demonstrate the proposed user interface for analysis. It is better to discover any difficulties or to change your mind now than to do so after programming is underway or even complete.

Your previous programming assignments probably stated the program specifications for you. Perhaps aspects of these specifications were unclear and you had to seek clarification, but most likely you have had little practice in writing your own program specifications.

Phase 2: Design. Once you have completed the specification phase, you must design a solution to the problem. Most people who design solutions of moderate size and complexity find it difficult to cope with the entire program at once. The best way to simplify the problem-solving process is to divide a large problem into small, manageable parts. The resulting program will contain **modules**, which are self-contained units of code.

When using an object-oriented language such as Java, these modules take the form of objects. As discussed in Chapter 1, objects are implemented using

Make the problem statement precise and detailed

Prototype programs can clarify the problem

classes. Classes should be designed so that the objects are independent, or **loosely coupled**. Coupling is the degree to which objects in a program are interdependent. If every object in a program is connected to every other object in the program, that is called highly coupled, and it means that the flow of information between objects is potentially high. If the objects are loosely coupled, changes in one object will have minimal effects on other objects in the program.

Classes should also be designed so that objects are highly cohesive. Cohesion is the degree to which the data and methods of an object are related. Ideally, each object should represent one component in the solution. Methods within an object should also be highly cohesive, each should perform one well-defined task.

During the design phase, it is also important that you clearly specify the object interactions. Objects interact by sending messages to each other through method calls, which in turn represents the **data flow** among objects. When designing the methods, you should provide answers to these questions: What data within the object is utilized by the method? What does the method assume? What actions does the method perform, and is the data stored in the object changed after the method executes? Thus you should specify in detail the assumptions, input, and output for each method.

For example, if you as program designer needed to provide a method for a shape object that moves it to a new location on the screen, you might write the following specification:

The method will receive an (x, y) coordinate.

The method will move the shape to the new location on the screen.

You can view these specifications as the terms of a **contract** between your method and the code that calls it.

If you alone write the entire program, this contract helps you systematically decompose the problem into smaller tasks. If the program is a team project, the contract helps delineate responsibilities. Whoever writes the move method must live up to this contract. After the move method has been written and tested, the contract tells the rest of the program how to call the move method properly and lets it know the result of doing so.

It is important to notice, however, that a method's contract does not commit the method to a particular way of performing its task. If another part of the program assumes anything about the method, it does so at its own risk. Thus, for example, if at some later date you rewrite your method to use a different algorithm for moving the shape on the screen, you should not need to change the rest of the program at all. As long as the new method honors the terms of the original contract, the rest of the program should be oblivious to the change.

This discussion should not be news to you. Although you might not have explicitly used the term "contract" before, the concept should be familiar. You write a contract when you write a method's **precondition**, which is a statement of the conditions that must exist at the beginning of a method, as well as when you write its **postcondition**, which is a statement of the conditions at

Loosely coupled
objects are
independent

Highly cohesive
methods each
perform one well-
defined task

Specify each
method's purpose,
assumptions, input,
and output

Specifications as a
contract

A method's specifi-
cation should not
describe a method
of solution

Method specifica-
tions include precise
preconditions and
postconditions

the end of a method. For example, the move method that adheres to the previous contract could appear in pseudocode² as

First-draft specifications

```
move(x, y)
// Moves a shape to a new location on the screen.
// Precondition: The calling code provides an
// (x, y) pair, both integers.
// Postcondition: The shape is moved to the new
// location.
```

These particular pre- and postconditions are actually deficient, as may be the case in a first-draft contract. For example, does “moved” mean that the shape is moved relative to its previous location by (x, y) or that the shape is moved to the new coordinate location (x, y) ? What is the range of values for x and y ? While implementing this method, you might assume that “moved” means the shape is moved to a new coordinate location (x, y) and that the range for x and y is 0 through 100. Imagine the difficulties that can arise when another person tries to use *move* to move a shape relative to its previous location using $(-5, -5)$. This user does not know your assumptions unless you document them by revising the pre- and postconditions, as follows:

Revised specifications

```
move(x, y)
// Moves a shape to coordinate (x, y) on the screen.
// Precondition: The calling code provides an
// (x, y) pair, both integers, where
// 0 <= x <= MAX_XCOOR, 0 <= y <= MAX_YCOOR, where
// MAX_XCOOR and MAX_YCOOR are class constants that
// specify the maximum coordinate values.
// Postcondition: The shape is moved to coordinate
// (x, y).
```

When you write a precondition, begin by describing the method’s formal parameters, mention any class named constants that the method uses, and finally list any assumptions that the method makes. Similarly, when you write a postcondition, begin by describing the method’s effect on its parameters—or in the case of a valued method, the value it returns—and then describe any other action that has occurred. (Although people tend to use the words parameter and argument interchangeably, we will use parameter to mean formal parameter and argument to mean actual argument.)

In an object-oriented system, a method may also change the state of an object. Object state refers to the data that an object holds. In this example, a shape object has two data values that represent its location on the screen. The *move* method actually modifies these values within the object so that the effect

2. Pseudocode in this book appears in italics.

is to move the shape to a different location on the screen. Note that the post-condition in the `move` method reflects this change of object state.

Novice programmers tend to dismiss the importance of precise documentation, particularly when they are simultaneously designer, programmer, and user of a small program. If you design `move` but do not write down the terms of the contract, will you remember them when you later implement the method? Will you remember how to use `move` weeks after you have written it? To refresh your memory, would you rather examine your Java code or read a simple set of pre- and postconditions? As the size of a program increases, good documentation becomes even more important, regardless of whether you are the sole author or part of a team.

You should not ignore the possibility that you or someone else has already implemented some of the required objects and methods. Java facilitates the reuse of software components, which are typically organized into class libraries that group classes into packages containing compiled code. That is, you will not always have access to a method's Java code. The **Java Application Programming Interface (API)** is an example of one such collection of preexisting software. For example, you know how to use the static method `sqrt` contained in the Java API package `java.lang.Math`, yet you do not have access to its source statements, because it is precompiled. You know, however, that if you pass `sqrt` an expression of type `double`, it will return the square root of the value of that expression as a `double`. You can use `java.lang.Math.sqrt` even though you do not know its implementation. Furthermore, it may be that `java.lang.Math.sqrt` was written in a language other than Java! There is so much about `java.lang.Math.sqrt` that you do not know, yet you can use it in your program without concern, *as long as you know its specifications*.

If, in the past, you have spent little or no time in the design phase for your programs, you must change this habit! The end result of the design phase should be a solution that is easy to translate into the constructs of a particular programming language. By spending adequate time in the design phase, you will spend less time when you write and debug your program.

We will resume our discussion of design later.

Precise documentation is essential

Incorporate existing software components into your design

Phase 3: Risk analysis. Building software entails risks. Some risks are the same for all software projects and some are peculiar to a particular project. You can predict some risks, while others are unknown. Risks can affect a project's timetable or cost, the success of a business, or the health and lives of people. You can eliminate or reduce some risks but not others. Techniques exist to identify, assess, and manage the risks of creating a software product. You will learn these techniques if you study software engineering in a subsequent course. The outcome of risk analysis will affect the other phases of the life cycle.

You can predict and manage some, but not all, risks

Phase 4: Verification. Formal, theoretical methods are available for proving that an algorithm is correct. Although research in this area is incomplete, it is useful to mention some aspects of the verification process.

An **assertion** is a statement about a particular condition at a certain point in an algorithm. Preconditions and postconditions are simply assertions about conditions at the beginning and end of methods.

Java supports an assertion statement that allows you to test a condition at a certain point in a program. The Java assertion statement has two forms:

```
assert booleanExpression;  
assert booleanExpression : valueExpression;
```

In the first form, if *booleanExpression* is false, an **AssertionError** is thrown with no further detail information. In the second form, if *booleanExpression* is false, the *valueExpression* is evaluated and sent to the **AssertionError** constructor so as to provide more detailed information about the failed assertion. In many instances, the *valueExpression* is simply a string that describes the problem. Here is a simple example of an **assert** statement in a program:

```
public static void main(String[] args) {  
    Scanner reader = new Scanner(System.in);  
    System.out.print("Enter your score: ");  
    int score = reader.nextInt();  
    assert score >= 0 && score <= 100 :  
        "Score "+score+" is not in range 0-100";  
    // Continue processing score  
    System.out.println("Processing score...");  
}
```

So if a value out of range is entered by the user, a message similar to the following will appear:

```
Exception in thread "main" java.lang.AssertionError:  
    Score -23 is not in range 0-100  
    at AssertionClass.main(AssertionClass.java:9)
```

Note that for the **assert** statement to be executed in a program, you must make sure that the compiler settings enable assertions. In most Integrated Development Environments (IDEs), this feature is usually turned off by default, and so the assertion statements will be ignored.

An **invariant** is a condition that is always true at a particular point in an algorithm. A **loop invariant** is a condition that is true before and after each execution of an algorithm's loop. As you will see, loop invariants can help you to write correct loops. By using invariants, you can detect errors before you begin coding and thereby reduce your debugging and testing time. Overall, invariants can save you time.

Proving that an algorithm is correct is like proving a theorem in geometry. For example, to prove that a method is correct, you would start with its

preconditions—which are analogous to the axioms and assumptions in geometry—and demonstrate that the steps of the algorithm lead to the post-conditions. To do so, you would consider each step in the algorithm and show that an assertion before the step leads to a particular assertion after the step.

By proving the validity of individual statements, you can prove that sequences of statements, and then methods, and finally the program itself are correct. For example, suppose you show that if assertion A_1 is true and statement S_1 executes, assertion A_2 is true. Also, suppose you have shown that assertion A_2 and statement S_2 lead to assertion A_3 . You can then conclude that if assertion A_1 is true, executing the sequence of statements S_1 and S_2 will lead to assertion A_3 . By continuing in this manner, you eventually will be able to show that the program is correct.

Clearly, if you discovered an error during the verification process, you would correct your algorithm and possibly modify the problem specifications. Thus, by using invariants, it is likely that your algorithm will contain fewer errors *before* you begin coding. As a result, you will spend less time debugging your program.

You can formally prove that particular constructs such as *if* statements, loops, and assignments are correct. An important technique uses loop invariants to demonstrate the correctness of iterative algorithms. For example, we will prove that the following simple loop computes the sum of the first n elements in the array *item*:

```
// computes the sum of item[0], item[1], . . .
// item[n-1] for any n >= 1
int sum = 0;
int j = 0;
while (j < n) {
    sum += item[j];
    ++j;
} // end while
```

Before this loop begins execution, *sum* is 0 and *j* is 0. After the loop executes once, *sum* is *item[0]* and *j* is 1. In general,

sum is the sum of the elements *item[0]* through *item[j-1]*

Loop invariant

This statement is the invariant for this loop. The invariant for a correct loop is true at the following points:

- Initially, after any initialization steps, but before the loop begins execution
- Before every iteration of the loop
- After every iteration of the loop
- After the loop terminates

For the previous loop example, these points are as follows:

```

int sum = 0;
int j = 0;           ←the invariant is true here
while (j < n) {    ←the invariant is true here
    sum += item[j];
    ++j;             ←the invariant is true here
} // end while      ←the invariant is true here

```

You can use these observations to prove the correctness of an iterative algorithm. For the previous example, you must show that each of the following four points is true:

Steps to establish the correctness of an algorithm

1. **The invariant must be true initially**, before the loop begins execution for the first time. In the previous example, `sum` is 0 and `j` is 0 initially. In this case, the invariant states that `sum` contains the sum of the elements `item[0]` through `item[-1]`; the invariant is true because there are no elements in this range.
2. **An execution of the loop must preserve the invariant**. That is, if the invariant is true before any given iteration of the loop, you must show that it is true after the iteration. In the example, the loop adds `item[j]` to `sum` and then increments `j` by 1. Thus, after an execution of the loop, the most recent element added to `sum` is `item[j-1]`; that is, the invariant is true after the iteration.
3. **The invariant must capture the correctness of the algorithm**. That is, you must show that if the invariant is true when the loop terminates, the algorithm is correct. When the loop in the previous example terminates, `j` contains `n`, and the invariant is true: `sum` contains the sum of the elements `item[0]` through `item[n-1]`, which is the sum that you intended to compute.
4. **The loop must terminate**. That is, you must show that the loop will terminate after a finite number of iterations. In the example, `j` begins at 0 and then increases by 1 at each execution of the loop. Thus, `j` eventually will equal `n` for any $n \geq 1$. This fact and the nature of the `while` statement guarantee that the loop will terminate.

Not only can you use invariants to show that your loop is correct, but you can also use them to show that your loop is wrong. For example, suppose that the expression in the previous `while` statement was `j <= n` instead of `j < n`. Steps 1 and 2 of the previous demonstration would be the same, but Step 3 would differ: When the loop terminated, `j` would contain $n + 1$ and, because the invariant would be true, `sum` would contain the sum of the elements

item[0] through *item[n]*. Since this is not the desired sum, you know that something is wrong with your loop.

Notice the clear connection between Steps 1 through 4 and mathematical induction.³ Showing the invariant to be true initially, which establishes the **base case**, is analogous to establishing that a property of the natural numbers is true for zero. Showing that each iteration of the loop preserves the invariant is the **inductive step**. This step is analogous to showing that if a property is true for an arbitrary natural number *k*, then the property is true for the natural number *k* + 1. After proving the four points just described, you can conclude that the invariant is true after every iteration of the loop—just as mathematical induction allows you to conclude that a property is true for every natural number.

Identifying loop invariants will help you to write correct loops. You should state the invariant as a comment that either precedes or begins each loop, as appropriate. For example, in the previous example, you might write the following:

```
// Invariant: 0 <= j <= n and
// sum = item[0] +...+ item[j-1]
while (j < n)
    ...
}
```

State loop invariants
in your programs

You should confirm that the invariants for the following unrelated loops are correct. Remember that each invariant must be true both before the loop begins and after each iteration of the loop, including the final one. Also, you might find it easier to understand the invariant for a **for** loop if you temporarily convert it to an equivalent **while** loop.

For example, a **for** loop of the form

```
for (initialize; test; update) {
    statement(s)
} // end for
```

can be rewritten as

```
initialize;
while (test) {
    statement(s)
    update;
} // end while
```

3. A review of mathematical induction appears in Appendix D.

Here are a few more examples of loop invariants:

Examples of loop invariants

```
// Computes an approximation to ex for a real x
double t = 1.0, s = 1.0;
int k = 1;
// Invariant: t == xk-1/(k-1)! and
// s == 1+x+x2/2!+...+xk-1/(k-1)!
while (k <= n) {
    t *= x/k;
    s += t;
    ++k;
} // end while

// Computes n! for an integer n >= 0
int f = 1;
// Invariant: f == (j-1)!
for (int j = 1; j <= n; ++j) {
    f *= j;
} // end for
```

Coding is a relatively minor phase in the software life cycle

Phase 5: Coding. The coding phase involves translating the design into a particular programming language and removing the syntax errors. Although this phase is probably your concept of what programming is all about, it is important to realize that the coding phase is not the major part of the life cycle for most software—actually, it is a relatively minor part.

Design a set of test data to test your program

Phase 6: Testing. During the testing phase, you need to remove as many logical errors as you can. One approach is to test the individual methods of the objects first, using valid input data that leads to a known result. If certain data must lie within a range, include values at the endpoints of the range. For example, if the input value for n can range from 1 to 10, be sure to include test cases in which n is 1 and 10. Also, include invalid data to test the error-detection capability of the program. Try some random data, and finally try some actual data. Testing is both a science and an art. You will learn more about testing in subsequent courses.

Develop a working program under simplifying assumptions; then add refining sophistication

Phase 7: Refining the solution. The result of Phases 1 through 6 of the solution process is a working program, which you have tested extensively and debugged as necessary. If you have a program that solves your original problem, you might wonder about the significance of this phase of the solution process.

Often the best approach to solving a problem is first to make some simplifying assumptions during the design of the solution—for example, you could assume that the input will be in a certain format and will be correct—and next to develop a complete working program under these assumptions. You can then add more sophisticated input and output routines, additional features, and more error checks to the working program.

Thus, the approach of simplifying the problem initially makes a refinement step necessary in the solution process. Of course, you must take care to ensure that the final refinements do not require a complete redesign of the solution. You can usually make these additions cleanly, however, particularly when you have used a modular design. In fact, the ability to proceed in this manner is one of the key advantages of having a modular design! Also, realize that any time you modify a program—no matter how trivial the changes might seem—you must thoroughly test it again.

This discussion illustrates that the phases within the life cycle of software are not completely isolated from one another and are not linear. To make realistic simplifying assumptions early in the design process, you should have some idea of how you will account for those assumptions later on. Testing a program can suggest changes to its design, but changes to a program require that you test the program again.

Changes to a program require that you test it again

Phase 8: Production. When the software product is complete, it is distributed to its intended users, installed on their computers, and used.

Phase 9: Maintenance. Maintaining a program is not like maintaining a car. Software does not wear out if you neglect it. However, users of your software invariably will detect errors that you did not discover during the testing phase. Correcting these errors is part of maintaining the software. Another aspect of the maintenance phase involves enhancing the software by adding more features or by modifying existing portions to suit the users better. Rarely will the people who design and implement the original program perform this maintenance step. Good documentation then becomes even more important.

Correcting user-detected errors and adding features are aspects of software maintenance

Is a program's life cycle relevant to your life? It definitely should be! You should view Phases 1 through 7 as the steps in a problem-solving process. Using this strategy, you first design and implement a solution (Phases 1 through 6) based on some initial simplifying assumptions. The outcome is a well-structured program that solves a somewhat simplified problem. The last step of the solution process (Phase 7) refines your work into a sophisticated program that meets the original problem specifications.

What Is a Good Solution?

Before you devote your time and energy to the study of problem-solving techniques, it seems only fair that you see at the outset why mastery of these techniques will help to make you a good problem solver. An obvious statement is that the use of these techniques will produce good solutions. This statement, however, leads to the more fundamental question, what *is* a good solution? A brief attempt at answering this question concludes this section.

Because a computer program is the final form your solutions will take, consider what constitutes a good computer program. Presumably, you write a program to perform some task. In the course of performing that task, there is a real and tangible cost. This cost includes such factors as the computer resources

(computing time and memory) that the program consumes, any difficulties encountered by those who use the program, and the consequences of a program that does not behave correctly.

However, the costs just mentioned do not give the whole picture. They pertain to only one phase of the life cycle of a solution—the phase in which it is an operational program. In assessing whether or not a solution is good, you also must consider the phases during which you developed the solution and the phases after you wrote the initial program that implemented the solution. Each of these phases incurs costs, too. The total cost of a solution must take into account the value of the time of the people who developed, refined, coded, debugged, and tested it. A solution's cost must also include the cost of maintaining, modifying, and expanding it.

Thus, when calculating the overall cost of a solution, you must include a diverse set of factors. If you adopt such a multidimensional view of cost, it is reasonable to evaluate a solution against the following criterion:

A solution is good if the total cost it incurs over all phases of its life cycle is minimal.

It is interesting to consider how the relative importance of the various components of this cost has changed since the early days of computing. In the beginning, the cost of computer time relative to human time was extremely high. In addition, people tended to write programs to perform very specific, narrowly defined tasks. If the task changed somewhat, a new program was written. Program maintenance was probably not much of an issue, so there was little concern if a program was hard to read. A program typically had only one user, its author. As a consequence, programmers tended not to worry about misuse or ease of use of their programs; a program's interface generally was not considered important.

In this type of environment, one cost clearly overshadowed all others: computer resources. If two programs performed the same task, the one that required less time and memory was better. How things have changed! Since the early days of computers, computing costs have dropped dramatically, thus making the value of the problem solver's and programmer's time a much more significant factor in the cost of a solution. Another consequence of the drop in computing costs is that computers now are used to perform tasks in a wide variety of areas, many of them nonscientific. People who interact with computers often have no technical expertise and no knowledge of the workings of programs. People want their software to be easy to use.

Today, programs are larger and more complex than ever before. They are often so large that many people are involved in their design, use, and maintenance. Good structure and documentation are thus of the utmost importance. As programs perform more highly critical tasks, the prices for malfunctions will soar. Thus, society needs both well-structured programs and techniques for formally verifying their correctness. People will not and should not entrust their livelihoods—or their lives—to a program that only its authors can understand and maintain.

A multidimensional view of a solution's cost

Programs must be well structured and documented

These developments have made obsolete the notion that the most efficient solution is always the best. If two programs perform the same task, it is no longer true that the faster one is necessarily better. Programmers who use every trick in the book to save a few microseconds of computing time at the expense of clarity are not in tune with the cost structure of today's world. You must write programs with people as well as computers in mind.

At the same time, do not get the false impression that the efficiency of a solution is no longer important. To the contrary, in many situations efficiency is the prime determinant of whether a solution is even usable. The point is that a solution's efficiency is only one of many factors that you must consider. If two solutions have approximately the same efficiency, other factors should dominate the comparison. However, when the efficiencies of solutions differ significantly, this difference can be the overriding concern. The stages of the problem-solving process at which you should be most concerned about efficiency are those during which you develop the underlying methods of solution. The choice of a solution's components—the algorithms and ways to store data—rather than the code you write, leads to significant differences in efficiency.

Efficiency is only one aspect of a solution's cost

Another factor in software development costs is code reusability. Making use of existing code can reduce the cost and time needed to develop a solution. It also reduces maintenance costs since reused components are generally well designed and more comprehensively tested. Within the software development process, code reuse typically emerges in two ways. First, components available from code libraries and open source repositories can often be adapted and used in a system. Note that the original design of these off-the-shelf components is completely independent of the current software development activity, yet these components are adapted and refined to be part of the current solution. The second way that code reuse emerges is when components within a project are designed in such a way that allows them to be the basis for more specific components later in the development process.

Code reuse can reduce a solution's cost

This book advocates a problem-solving philosophy that views the cost of a solution as multidimensional. This philosophy is reasonable in today's world, and it likely will be reasonable in the years to come.

2.2 Achieving an Object-Oriented Design

You have seen the importance of specifying the objects during the design of a solution, but how do you determine the objects in the first place? The techniques that help you determine the objects for a particular solution are the subject of entire texts and future courses; these techniques quickly go beyond this book's scope. This section will provide an overview of two general design techniques—abstraction and information hiding—which is followed by a discussion of object-oriented design and functional decomposition.

Abstraction and Information Hiding

Specify what to do.
not how to do it

Specify what a
method does, not
how to do it

Specifications do
not indicate how to
implement a method

Procedural abstraction. When you design a method as part of a solution to a problem, each method begins as a box that states what it does but not how it does it. No one box may “know” how any other box performs its task—it may know only what that task is. For example, if one part of a solution is to sort some data, one of the boxes will be a sorting algorithm, as Figure 2-2 illustrates. The other boxes will know that the sorting box sorts, but they will not know how it sorts. In this way, the various components of a solution are kept isolated from one another.

Procedural abstraction separates the purpose of a method from its implementation. Abstraction specifies each method clearly *before* you implement it in a programming language. For example, what does the method assume and what action does it take? Such specifications will clarify the design of your solution because you will be able to focus on its high-level functionality without the distraction of implementation details. In addition, these principles allow you to modify one part of a solution without significantly affecting the other parts. For example, you should be able to change the sorting algorithm in the previous example without affecting the rest of the solution.

As the problem-solving process proceeds, you gradually refine the boxes until eventually you implement their actions by writing actual Java code. Once a method is written, you can use it without knowing the particulars of its algorithm as long as you have a statement of its purpose and a description of its parameters. Assuming that the method is documented properly, you will be able to use it knowing only its declaration and its initial descriptive comments; you will not need to look at its implementation.

Procedural abstraction is essential to team projects. After all, in a team situation, you will have to use methods written by others, frequently without knowledge of

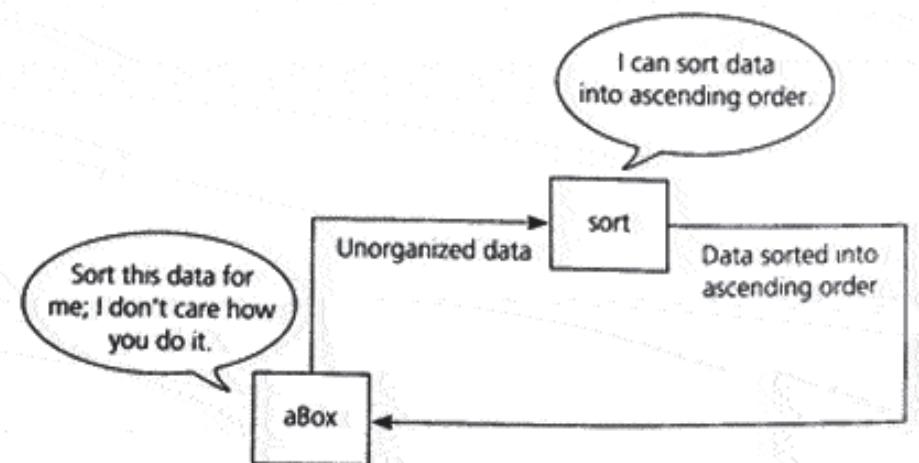


FIGURE 2-2

The details of the sorting algorithm are hidden from other parts of the solution

their algorithms. Will you actually be able to use such a method without studying its code? In fact, you do so each time you use a method from the Java API, such as `Math.sqrt`, as was noted earlier.

Data abstraction: Consider now a collection of data and a set of operations on the data. The operations might include ones that add new data to the collection, remove data from the collection, or search for some data. **Data abstraction** focuses on what the operations do instead of on how you will implement them. The other modules of the solution will “know” *what* operations they can perform, but they will not know *how* the data is stored or *how* the operations are performed.

For example, you have used an array, but have you ever stopped to think about what an array actually is? You will see many pictures of arrays throughout this book. This artist’s conception of an array might resemble the way a Java array is implemented on a computer, and then again it might not. The point is that you are able to use an array without knowing what it “looks like”—that is, how it is implemented. Although different systems may implement arrays in different ways, the differences are transparent to the programmer. For instance, regardless of how the array `years` is implemented, you can always store the value 1492 in location `index` of the array by using the statement

```
years[index] = 1492;
```

and later write out that value by using the statement

```
System.out.println(years[index]);
```

Thus, you can use an array without knowing the details of its implementation, just as you can use the method `Math.sqrt` without knowing the details of its implementation.

Most of this book is about data abstraction. To enable you to think abstractly about data—that is, to focus on what operations you will perform on the data instead of how you will perform them—you should define an **abstract data type**, or **ADT**. An ADT is a collection of data *and* a set of operations on the data. You can use an ADT’s operations, if you know their specifications, without knowing how the operations are implemented or how the data is stored.

Ultimately, someone—perhaps you—will implement the ADT by using a **data structure**, which is a construct that you can define within a programming language to store a collection of data. For example, you might store some data in a Java array of integers or in an array of objects or in an array of arrays.

Within problem solving, abstract data types support algorithms, and algorithms are part of what constitutes an abstract data type. As you design a solution, you should develop algorithms and ADTs in tandem. The global algorithm that solves a problem suggests operations that you need to perform on the data,

Specify what you
will do to data, not
how to do it

An ADT is not a
fancy name
for a data structure

Develop algorithms
and ADTs
in tandem

which in turn suggest ADTs and algorithms for performing the operations on the data. However, the development of the solution may proceed in the opposite direction as well. The kinds of ADTs that you are able to design can influence the strategy of your global algorithm for solving a problem. That is, your knowledge of which data operations are easy to perform and which are difficult can have a large effect on how you approach a problem.

As you probably have surmised from this discussion, you often cannot sharply distinguish between an “algorithms problem” and a “data structures problem.” Frequently, you can look at a program from one perspective and feel that the data structures support a clever algorithm and then look at the same program from another perspective and feel that the algorithms support a clever data structure.

All modules and ADTs should hide something

Information hiding. As you have seen, abstraction tells you to write specifications for each module that describe its outside, or **public**, view. However, abstraction also helps you to identify details that you should hide from public view—details that should not be in the specifications but should be **private**. The principle of **information hiding** tells you not only to hide such details within a module, but also ensures that no other module can tamper with these hidden details.

Information hiding limits the ways in which you need to deal with methods and data. As a user of a module, you do not worry about the details of its implementation. As an implementer of a module, you do not worry about its uses.

Object-Oriented Design

Objects encapsulate data and operations

One way to achieve an object-oriented design is to develop objects that combine data and operations to produce a representation of a real-life entity or abstraction. Such an **object-oriented** approach to modularity produces a collection of objects that have behaviors.

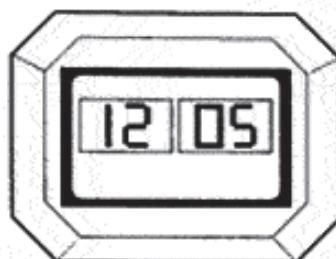
Although you may have never thought about it before, you can view many of the things around you as objects. The alarm clock that awoke you this morning **encapsulates** both time and operations such as “set the alarm.” To **encapsulate** means to encase or enclose; thus, encapsulation is a technique that hides inner details. Whereas methods encapsulate actions, objects encapsulate data as well as actions. Even though you request the clock to perform certain operations, you cannot see how it works. You see only the results of those operations.

Suppose that you want to write a program to display a clock on your computer screen. To simplify the example, consider a digital clock without an alarm, as Figure 2-3 illustrates. You would begin the task of designing a modular solution by identifying the objects in the problem.

Several techniques are available for identifying objects, but no single one is always the best approach. One simple technique⁴ considers the nouns and

Encapsulation hides inner details

4. This technique is not foolproof. The problem specification must use nouns and verbs consistently. If, for example, “display” is sometimes a verb and sometimes a noun, identifying objects and their operations can be unclear.

**FIGURE 2-3**

A digital clock

verbs in the problem specifications. The nouns will suggest objects whose actions are indicated by the verbs. For example, you could specify the clock problem as follows:

The program will maintain a digital clock that displays the time in hours and minutes. The hour indicator and minute indicator are both digital devices that display values from 1 to 12 and 0 to 59, respectively. You should be able to set the time by setting the hour and minute indicators, and the clock should maintain the time by updating these indicators.

Even without a detailed problem specification, you know that one of the objects is the clock itself. The clock performs operations such as

Set the time
Advance the time
Display the time

The hour indicator and minute indicator are also objects and are quite similar to each other. Each indicator performs operations such as

Set its value
Advance its value
Display its value

In fact, both indicators can be the same type of object. A set of objects that have the same type is called a class. Thus, what you need to specify is not a particular object, but a class of objects. In fact, you need a class of clocks and a class of indicators. A clock object, which is an instance of the clock class, will then contain two indicator objects, which are instances of the indicator class.

Chapter 4 discusses encapsulation further and, in particular, its relationship to Java classes. In subsequent chapters, you will study various ADTs and their implementations as Java classes. The focus will be on data abstraction and encapsulation. This approach to programming is object based.

Specifications for a program that displays a digital clock

An object is an instance of a class

Object-oriented programming, or OOP, adds two more principles to encapsulation:

KEY CONCEPTS**Three Principles of Object-Oriented Programming**

1. **Encapsulation:** Objects combine data and operations.
2. **Inheritance:** Classes can inherit properties from other classes.
3. **Polymorphism:** Objects can determine appropriate operations at execution time.

Classes can inherit properties from other classes. For example, once you have defined a class of clocks, you can design a class of alarm clocks that inherits the properties of a clock but adds operations to provide an alarm. You will be able to produce an alarm clock quickly because the clock portion is done. Thus, inheritance allows you to reuse classes that you defined earlier—perhaps for different but related purposes—with appropriate modification.

Inheritance may make it impossible for the compiler to determine which operation you require in a particular situation. However, polymorphism—which literally means *many forms*—enables this determination to be made at execution time. That is, the outcome of a particular operation depends upon the objects on which the operation acts. For example, if you use the + operator with numeric operands in Java, addition occurs, but if you use it with string operands, concatenation occurs. Although in this simple example, the compiler can determine the correct meaning of +, polymorphism allows situations in which the meaning of an operation is unknown until execution time.

Chapter 8 discusses inheritance and polymorphism further.

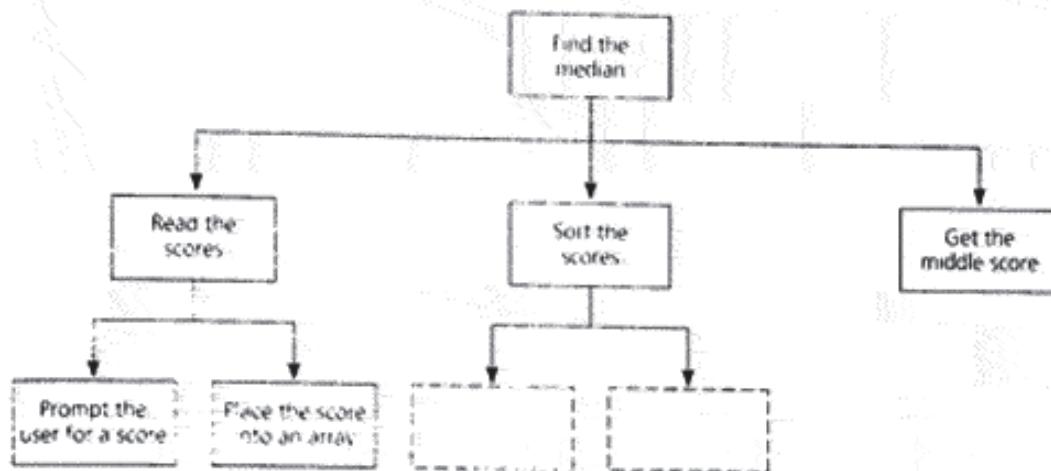
The + operator has multiple meanings

Functional Decomposition

Generally, an object-oriented approach initially focuses on the data aspects of the design. But equally important is the design of the methods that implement the behavior of the objects. Recall that we want the methods within a class to be highly cohesive—they should represent a single task to be performed in an object. Functional decomposition (also referred to as top down design) can help us break down complex tasks within an object into more manageable single-purpose tasks and subtasks.

The philosophy of functional decomposition is that you should address a task at successively lower levels of detail. Consider a simple example. Suppose that you wanted to find the median among a collection of test scores. Figure 2-4 uses a **structure chart** to illustrate the hierarchy of, and interaction among, the methods that solve this problem. At first, each method is little more than a statement of *what* it needs to solve and is devoid of detail. You refine each method by partitioning it into additional smaller methods. The result is a hierarchy of methods; each method is refined by its successors, which solve smaller problems and contain more detail about *how* to solve the problem than their predecessors. The refinement process continues

A structure chart shows the relationship among methods

**FIGURE 2-4**

A structure chart showing the hierarchy of methods

until the methods at the bottom of the hierarchy are simple enough for you to translate directly into Java code that solves very small, independent problems.

Notice in Figure 2-4 that you can break the solution down into three independent tasks:

*Read the test scores
Sort the scores
Get the "middle" score*

A solution consisting of independent tasks

If the three methods in this example perform their tasks, then by calling them in order you will correctly find the median, regardless of *how* each method performs its task.

You begin to develop each method by dividing it into subtasks. For example, you can refine the task of reading the test scores by dividing it into the following two subtasks:

*Prompt the user for a score
Place the score into an array*

Subtasks

You continue the solution process by developing, in a similar manner, methods for each of these two tasks. Finally, you can use pseudocode to specify the details of the algorithms.

General Design Guidelines

Typically, you use object-oriented design (OOD), functional decomposition (FD), abstraction, and information hiding when you design a solution to a

problem. The following design guidelines summarize an approach that leads to modular solutions.

KEY CONCEPTS**Design Guidelines**

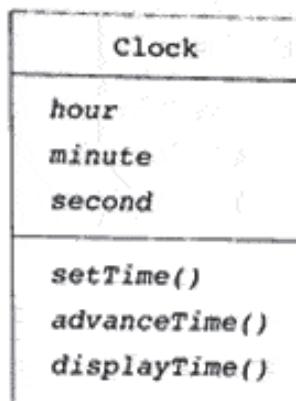
1. Use OOD and FD together to produce modular solutions. That is, develop abstract data types and algorithms in tandem.
2. Use OOD for problems that primarily involve data.
3. Use FD to design algorithms for an object's operations.
4. Consider FD to design solutions to problems that emphasize algorithms over data.
5. Focus on *what*, not *how*, when designing both ADTs and algorithms.
6. Consider incorporating previously written software components into your design.

Modeling Object-Oriented Designs Using UML

The Unified Modeling Language (UML) is a modeling language used to express object-oriented designs. UML provides specifications for both diagrams and text-based descriptions. The diagrams are particularly useful in showing the overall design of a solution, including class specifications and the various ways that the classes interact with each other. It is fairly common to have a number of classes involved in a solution, and thus the ability to show the interaction among classes is one of the strengths of UML.

This text focuses on the design of the classes themselves, and therefore only the class diagrams and associated syntax are presented here. Class diagrams specify the name of the class, the data members of the class, and the operations. Figure 2-5 shows a class diagram for the class `Clock` discussed earlier. The top section contains the class name. The middle section contains the data members that represent the data in the class, and the bottom section contains the operations. Note that the diagram is quite general; it does not really dictate how the class is actually implemented. It typically represents a conceptual model of the class that is language independent.

In conjunction with the class diagrams, UML also provides a text-based notation to represent the data members and operations for classes. This notation can be incorporated into the class diagrams, but usually not to the fullest extent because it tends to clutter the diagrams. This text-based representation is good to describe the classes in this text, because it provides more complete information than the diagrams. The UML notation for data members is

**FIGURE 2-5**

UML diagram for the class *Clock*

where

- *visibility* is + (*public*) or – (*private*). A third possibility is # (*protected*), which is discussed in Chapter 9.
- *name* is the name of the data member.
- *type* is the data type of the data member.
- *defaultValue* is an initial value for the data member.

As seen in the class diagrams, at a minimum the name should be provided. The *defaultValue* is used only in situations where a default value is appropriate. In some cases you may also want to omit the *type* of the data member and leave it to the implementation to provide that detail. This text will use the following names for common argument types: *integer* for integer values, *float* for floating-point values, *boolean* for boolean values, and *string* for string values. Note that these names do not necessarily match the corresponding Java data types because this notation is meant to be language independent.

Here is the text-based notation for the data members in the class *Clock* shown in Figure 2-5:

```

-hour: integer
-minute: integer
-second: integer
  
```

The data members *hour*, *minute*, and *second* are declared private, as suggested by the concept of information hiding.

The UML syntax for operations is more involved:

visibility name(parameter-list): return-type {property-string}

where

- *visibility* is the same as specified for data members.
- *name* is the name of the operation.
- *parameter-list* contains comma-separated parameters whose syntax is as follows:

direction name: type = defaultValue

where

- *direction* is used to show whether the parameter is used for input (*in*), output (*out*), or both (*inout*).
- *name* is the parameter.
- *type* is the data type of the parameter.
- *defaultValue* is a value that should be used for the parameter if no argument is provided.
- *return-type* is the data type of the result of the operation. If the operation does not return a value, this is left blank.
- *property-string* indicates property values that apply to the operation.

Like the class diagrams for data members, the class diagrams for operations at a minimum provide the *name* of the operation. Sometimes the *parameter-list* is included if it clarifies the understanding of the class functionality.

The *property-string* has a variety of possible values, but of interest in this text is the property *query*. It is a way to indicate that the operation does not modify any data in the class.

Here is the text-based notation for the operations in the class *Clock*:

```
+setTime(in hr: integer, in min: integer, in sec: integer)
-advanceTime()
+displayTime() {query}
```

Here we specified the operations *setTime* and *displayTime* as public, and *advanceTime* as private. The function *displayTime* also has the property *query* specified, as an indication that it does not change any of the data; the function is used only to display the data.

UML class diagrams provide additional notation to illustrate relationships between classes. Suppose that you are asked to model a banking system application. The specification is as follows:

Design a banking system that assigns checking and savings accounts to customers. The bank information includes a name and routing number. Both types of accounts allow balance retrieval, deposits, and withdrawals. A customer may have multiple accounts. Each customer's

name and address are stored in the system, and each account has a number assigned to it. Savings accounts earn interest and checking accounts charge for each check when the balance falls below a minimum amount. These adjustments are reflected when the customer requests the current account balance.

Several classes might be designed to represent the various aspects of a bank, as illustrated in Figure 2-6. These classes include a *Bank* class, an *Account* class, and a *Customer* class. Associations between classes are shown with a line, with the option to specify the cardinality between the associations. For example, a customer can have one or more accounts, which is illustrated with the notation "1...*" (one to many). Classes may also have different types of relationships with each other. For example, the *Savings* and *Checking* account classes are both derived from the *Account* class, and they inherit the *Account* class's data members and operations. Inheritance is represented with an open triangle pointing to the parent class. Note that the *Checking* and *Savings* classes have their

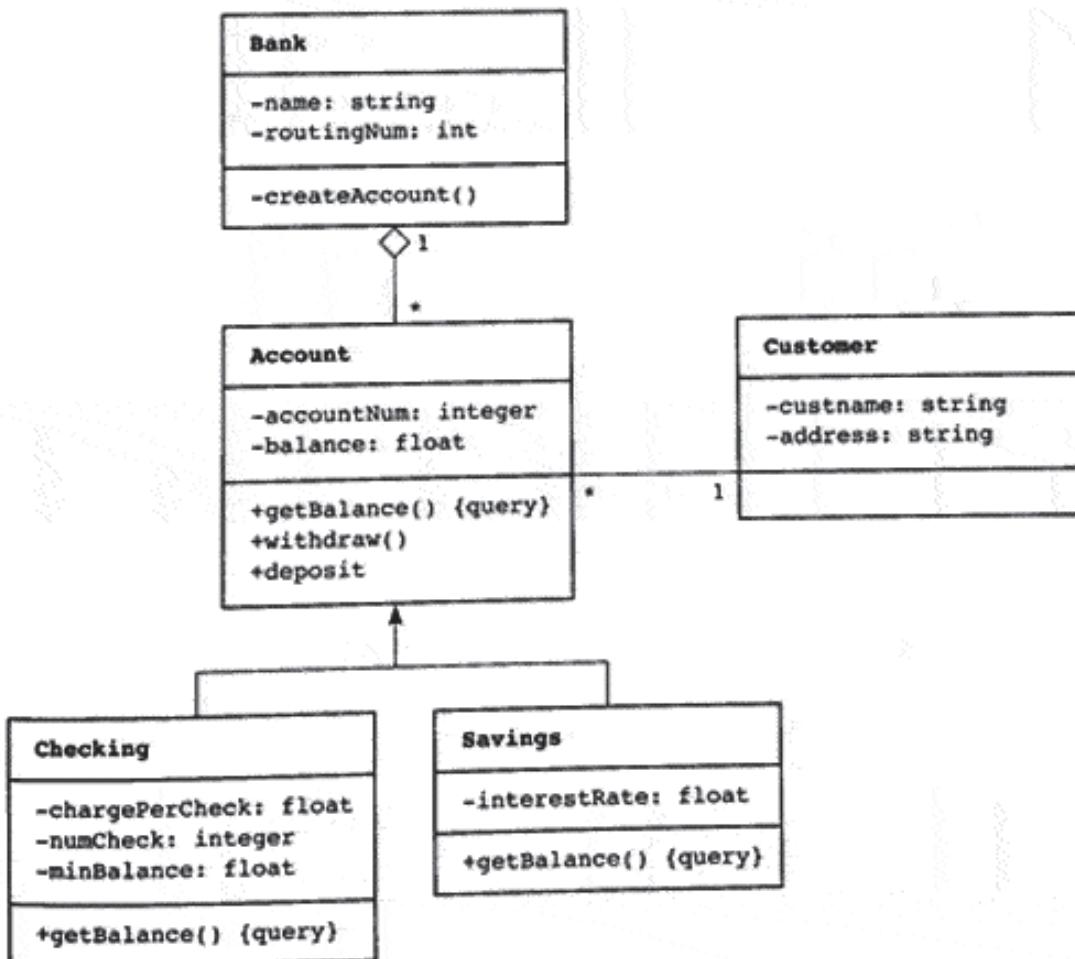


FIGURE 2-6

UML diagram for a banking system

own `getBalance` functions, which *override*, or replace, the `getBalance` function of the parent class, in order to make the necessary calculations for charges and interest. A class may also have a relationship with another class by containing an instance of that class as part of its definition. In the banking example, a bank contains one or more accounts. This type of relationship is called containment and is represented by positioning a diamond next to the containing class. Inheritance and containment are discussed in more detail in Chapter 9.

Advantages of an Object-Oriented Approach

The time that you expend on program design can increase when you use object-oriented programming (OOP). In addition, the solution that OOP techniques produce will typically be more general than is absolutely necessary to solve the problem at hand. The extra effort that OOP requires, however, is usually worth it.

When using object-oriented design in the solution to a problem, you need to identify the classes that are involved. You identify the purpose of each class and how it interacts with other classes. This leads to a specification for each class that identifies the operations and data. You then focus on the implementation details for each of the classes, including the use of top-down design to facilitate the development of the operations. It is easier to do the implementation when you focus on one class at a time.

Once you have implemented a class, you must test it at two different levels. First, you must test the class operations. This is usually done by writing a small program that calls the various operations and tests the results against the specifications provided for the operation. Once you have tested each individual class in this way, you should test scenarios in which the classes are expected to work together to solve the larger problem.

When you identify the classes involved in your solution, you will often find that you want a family of related classes. This stage of the design process is time-consuming, particularly if you have no existing classes upon which to build. Once you have implemented a class (called the ancestor class), the implementation of each new class (the descendant class) proceeds more rapidly, because you can reuse the properties and operations of the ancestor class. For example, as was mentioned earlier, once you have defined a class of clocks, you can design a class of alarm clocks that inherits the properties of a clock but adds operations to provide an alarm. The implementation of the class of alarm clocks would have been much more time-consuming if you did not have a class of clocks on which to base it. Looking ahead, you can reuse previously implemented classes in future programs, either as is or with modifications that can include new classes derived from your existing ones. This reuse of classes can actually reduce the time requirements of an object-oriented design.

OOP also has a positive effect on other phases of the software life cycle, such as program maintenance and verification. You can make one modification to an ancestor class and affect all of its descendants. Without inheritance, you would need to make the same change to many modules. In addition, you can add new features to a program by adding descendant classes that do not affect

A family of related classes

Reuse existing classes

Program maintenance and verification are easier when you use inheritance

their ancestors and, therefore, do not introduce errors into the rest of the program. You can also add a descendant class that modifies its ancestor's original behavior, even though that ancestor was written and compiled long ago.

2.3 A Summary of Key Issues in Programming

Given that a good solution is one that, in the course of its life cycle, incurs a small cost, the next questions to ask are, what are the specific characteristics of good solutions, and how can you construct good solutions? This section summarizes the answers to these very difficult questions.

The programming issues that this section discusses should be familiar to you. However, it is usually the case that the novice programmer does not truly appreciate their importance. After the first course in programming, many students still simply want to "get the thing to run." The discussion that follows should help you realize just how important these issues really are.

One of the most widespread misconceptions held by novice programmers is that a computer program is "read" only by a computer. As a consequence, they tend to consider only whether the computer will be able to "understand" the program—that is, will the program compile, execute, and produce the correct output? The truth is, of course, that other people often must read and modify programs. In a typical programming environment, many individuals share a program. One person may write a program, which other people use in conjunction with other programs written by other people, and a year later, a different person may modify the program. It is therefore essential that you take great care to design a program that is easy to read and understand.

You should always keep in mind the following six issues of program structure and design:

People read
programs, too

KEY CONCEPTS

Six Key Programming Issues

1. Modularity
2. Modifiability
3. Ease of use
4. Fail-safe programming
5. Style
6. Debugging

Modularity

Using object-oriented design for software development inherently leads to a modular design. As this book will continually emphasize, you should strive for modularity in all phases of the problem-solving process, beginning with the

initial design of a solution. You know from the earlier discussion of object-oriented design that many programming tasks become more difficult as the size and complexity of a program grows. Modularity slows the rate at which the level of difficulty grows. More specifically, modularity has a favorable impact on the following aspects of programming:

Modularity facilitates programming

- **Constructing the program.** The primary difference between a small modular program and a large modular program is simply the number of modules each contains. Because the modules are independent, writing one large modular program is not very different from writing many small, independent programs. On the other hand, working on a large nonmodular program is more like working on many interrelated programs simultaneously. Modularity also permits team programming, where several programmers work independently on their own modules before combining them into one program.

Modularity isolates errors

- **Debugging the program.** Debugging a large program can be a monstrous task. Imagine that you type a 10,000-line program and eventually get it to compile. Neither of these tasks would be much fun. Now imagine that you execute your program and, after a few hundred lines of output, you notice an incorrect number. You should anticipate spending the next day or so tracing through the intricacies of your program before discovering a problem such as an erroneous arithmetic expression.

A great advantage of modularity is that the task of debugging a large program is reduced to one of debugging many small programs. When you begin to code a module, you should be almost certain that all other modules coded so far are correct. That is, before you consider a module finished, you should test it extensively, both separately and in context with the other modules, by calling it with actual arguments carefully chosen to induce all possible behaviors of the modules. If this testing is done thoroughly, you can feel fairly sure that any problem is a result of an error in the last module added. *Modularity isolates errors.*

More theoretically, as was mentioned before, you can use formal methods to establish the correctness of a program. Modular programs are amenable to this verification process.

Modular programs are easy to read

- **Reading the program.** A person reading a large program may have trouble seeing the forest for the trees. Just as a modular design helps the programmer cope with the complexities of solving a problem, so too will a modular program help its reader understand how the program works. A modular program is easy to follow because the reader can get a good idea of what is going on without reading any of the code. A well-written method can be understood fairly well from only its name, initial comments, and the names of the other methods that it calls. Readers of a program need to study actual code only if they require a detailed understanding of how the program operates. Program readability is discussed further in the section on style later in this chapter.

■ **Modifying the program.** Modifiability is the topic of the next section, but as the modularity of a program has a direct bearing on its modifiability, a brief mention is appropriate here. A small change in the requirements of a program should require only a small change in the code. If this is not the case, it is likely that the program is poorly written and, in particular, that it is not modular. To accommodate a small change in the requirements, a modular program usually requires a change in only a few of its modules, particularly when the modules are independent (that is, loosely coupled) and each module performs a single well-defined task (that is, is highly cohesive).

When making changes to a program, it is best to make a few at a time. By maintaining a modular design, you can reduce a fairly large modification to a set of small and relatively simple modifications to isolated parts of the program. *Modularity isolates modifications*.

■ **Eliminating redundant code.** Another advantage of modular design is that you can identify a computation that occurs in many different parts of the program and implement it as a method. Thus, the code for the computation will appear only once, resulting in an increase in both readability and modifiability. The example in the next section demonstrates this point.

Modularity isolates modifications

Modularity eliminates redundancies

Modifiability

Imagine that the specification for a program changes after some period of time. Frequently, people require that a program do something differently than they specified originally, or they ask that it do more than they requested originally. This section offers two examples of how you can make a program easy to modify: through the use of methods and named constants.

Methods. Suppose that a library has a large program to catalog its books. At several points, the program displays the information about a requested book. At each of these points, the program could include a `System.out.println` statement to display the book's call number, author, and title. You could also replace each occurrence of this statement with a call to a method `displayBook` that displays the same information about the book. Alternatively, you could provide an implementation of the method `toString` in the book class that contains the information you want to display about a book. The method `toString` is invoked when a book object appears in a `System.out.println` statement.

Not only does the use of a method such as `displayBook` have the obvious advantage of eliminating redundant code, it also makes the resulting program easier to modify. For example, to change the format of the output, you need to change only the implementation of `displayBook` instead of numerous occurrences of the `System.out.println` statement. If you had not used a method, the modification would have required you to make changes at each point where the program displays the information. Merely finding each of these points could be difficult, and you probably would overlook a few. In this simple example, the advantages of using methods should be clear.

Methods make a program easier to modify

For another illustration, recall the earlier example of a solution that, as one of its tasks, sorted some data. Developing the sorting algorithm as an independent module and eventually implementing it as a method would make the program easier to modify. For instance, if you found that the sorting algorithm was too slow, you could replace the sort method without even looking at the rest of the program. You could simply “cut out” the old method and “paste in” the new one. If instead the sort was integrated into the program, the required surgery might be quite intricate.

In general, be concerned if you need to rewrite a program to accommodate small modifications. Usually, it is easy to modify a well-structured program slightly: Because each module solves only a small part of the overall problem, a small change in problem specifications usually affects only a few of the modules.

Named constants
make a program
easier to modify

Named constants. The use of named constants is another way to enhance the modifiability of a program. For example, the restriction that an array must be of a predefined, fixed size causes a bit of difficulty. Suppose that a program uses an array to process the SAT scores of the computer science majors at your university. When the program was written, there were 202 computer science majors, so the array was declared by

```
int [] scores = new int[202];
```

The program processes the array in several ways. For example, it reads the scores, writes the scores, and averages the scores. The pseudocode for each of these tasks contains a construct such as

```
for (index = 0 through 201)  
    Process the score
```

If the number of majors changes, not only must you revise the declaration of *scores*, but you must also change each loop that processes the array to reflect the new array size. In addition, other statements in the program might depend on the size of the array. A 202 here, a 201 there—which to change?

On the other hand, if you use a named constant such as

```
final int NUMBER_OF_MAJOR = 202;
```

you can declare the array by using

```
int [] scores = new int[NUMBER_OF_MAJOR];
```

and write the pseudocode for the processing loops in this form:

```
for (index = 0 through NUMBER_OF_MAJOR - 1)  
    Process the score
```

If you write expressions that depend on the size of the array in terms of the constant `NUMBER_OF_MAJOORS` (such as `NUMBER_OF_MAJOORS - 1`), you can change the array size simply by changing the definition of the constant and compiling the program again.

Ease of Use

Another area in which you need to keep people in mind is the design of the user interface. Humans often process a program's input and output. Here are a few obvious points:

- In an interactive environment, the program should always prompt the user for input in a manner that makes it quite clear what it expects. For example, the prompt "?" is not nearly as enlightening as the prompt "Please enter account number for deposit." You should never assume that the users of your program will know what response the program requires. Prompt the user for input
- A program should always echo its input. Whenever a program reads data, either from a user or from a file, the program should include the values it reads in its output. This inclusion serves two purposes: First, it gives the user a check on the data entered—a guard against typos and errors in data transmission. This check is particularly useful in the case of interactive input. Second, the output is more meaningful and self-explanatory when it contains a record of what input generated the output. Echo the input
- The output should be well labeled and easy to read. An output of Label the output

```
1800 < 6 > 18
Jones, Q. 223 2234.00 1088.19 N, J Smith, T. 111
110.23 I, Harris, V. 44 44000.00 22222.22
```

is more prone to misinterpretation than

CUSTOMER ACCOUNTS AS OF 1800 HOURS ON JUNE 1

Account status codes: N=new, J=joint, I=inactive

NAME	ACC#	CHECKING	SAVINGS	STATUS
Jones, Q.	223	\$ 2234.00	\$ 1088.19	N, J
Smith, T.	111	\$ 110.23	-----	I
Harris, V.	44	\$44000.00	\$22222.22	-----

These characteristics of a good user interface are only the basics. Several more subtle points separate a program that is merely usable from one that is user friendly. Students tend to ignore a good user interface, but by investing a little extra time here, you can make a big difference: the difference between a good program and one that only solves the problem. For example, consider a

A good user interface is important

program that requires a user to enter a line of data in some fixed format, with exactly one blank between the items. A free-form input that allows any number of blanks between the items would be much more convenient for the user. It takes so little time to write code that skips blanks, so why require the user to follow an exact format? Once you have made this small additional effort, it is a permanent part of both your program and your library of techniques. The user of your program never has to think about input format.

Fail-Safe Programming

A fail-safe program is one that will perform reasonably no matter how anyone uses it. Unfortunately, this goal is usually unattainable. A more realistic goal is to anticipate the ways that people might misuse the program and to guard carefully against these abuses.

Check for errors in input

This discussion considers two types of errors. The first type is an *error in input data*. For example, suppose that a program expects a nonnegative integer but reads -12. When a program encounters this type of problem, it should not produce incorrect results or abort with a vague error message. Instead, a fail-safe program provides a message such as

```
-12 is not a valid number of children.  
Please enter this number again.
```

Check for errors in logic

The second type of error is an *error in the program logic*. Although a discussion of this type of error belongs in the debugging section at the end of this chapter, detecting errors in program logic is also an issue of fail-safe programming. A program that appears to have been running correctly may, at some point, behave unexpectedly, even if the data that it reads is valid. For example, the program may not have accounted for the particular data that elicited the surprise behavior, even though you tried your best to test the program's logic. Or perhaps you modified the program and that modification invalidated an assumption that you made in some other part of the program. Whatever the difficulty, a program should have built-in safeguards against these kinds of errors. It should monitor itself and be able to indicate that something is wrong and you should not trust the results.

Guarding against errors in input data. Suppose that you are computing statistics about the people in income brackets between \$10,000 and \$100,000. The brackets are rounded to the nearest thousand dollars: \$10,000, \$11,000, and so on to \$100,000. The raw data is a file of one or more lines of the form

G N

where *N* is the number of people with an income that falls into the *G*-thousand-dollar group. If several people have compiled the data, several

entries for the same value of G might occur. As the user enters data, the program must add up and record the number of people for each value of G . From the problem's context, it is clear that G is an integer in the range 10 to 100 inclusive, and N is a nonnegative integer.

As an example of how to guard against errors in input, consider an input method for this problem. The first attempt at writing this method will illustrate several common ways in which a program can fall short of the fail-safe ideal. Eventually you will see an input method that is much closer to the fail-safe ideal than the original solution.

A first attempt at the class and methods might be

```
import java.util.Scanner;
public class IncomeStatistics {
    final static int LOW_END = 10;      // low end of incomes
    final static int HIGH_END = 100;     // high end of incomes
    final static int TABLE_SIZE = HIGH_END - LOW_END + 1;

    int[] incomeData;      // used to store the income data,
    // incomeData[G] stores the total number of
    // people that fall into the G-thousand-dollar
    // group

    public IncomeStatistics() {
        incomeData = new int[TABLE_SIZE];
    } // end constructor

    public void readData() {
        // -----
        // Reads and organizes income statistics.
        // Precondition: The calling code gives directions and
        // prompts the user. Input data is error-free, and each
        // input line is in the form G N, where N is the number of
        // people with an income in the G-thousand-dollar group
        // and LOW-END <= G <= HIGH-END. An input line with values
        // of zero for both G and N terminates the input.
        // Postcondition: incomeData[G-LOW-END] = total number of
        // people with an income in the G-thousand-dollar group
        // for each G read. The values read are displayed.
        // -----
        int group, number;                      // input values
        Scanner input = new Scanner(System.in);

        for (group = LOW_END; group <= HIGH_END; ++group) {
            // clear array
            incomeData[index(group)] = 0;
        } // end for
    }
}
```

This method is not
fail-safe

```

group = input.nextInt();
number = input.nextInt();

while ((group != 0) || (number != 0)) {
    System.out.println("Income group "+group+" contains " +
        number + " people.");
    incomeData[index(group)] += number;

    group = input.nextInt();
    number = input.nextInt();
} // end while
} // end readData

private int index(int group) {
// Returns the array index corresponding to group number.
return group - LOW_END;
} // end index

// other methods for class IncomeStatistics would follow

} // end IncomeStatistics

```

The `readData` method has some problems. If an input line contains unexpected data, the program will not behave reasonably. Consider two specific possibilities:

- The first integer on the input line, which the method assigns to `group`, is not in the range `LOW-END` to `HIGH-END`. The reference `incomeData[index(group)]` will then throw the exception `IndexOutOfBoundsException`.
- The second number on the input line, which the method assigns to `number`, is negative. Although a negative value for `number` is invalid because you cannot have a negative number of people in an income group, the method will add `number` to the group's array entry. Thus, the array `incomeData` will be incorrect.

After the method reads values for `group` and `number`, it must check to see whether `group` is in the range `LOW-END` to `HIGH-END` and whether `number` is positive. If either value is not in range, you must handle the input error.

Instead of checking `number`, you might think to check the value of `incomeData[index(group)]`, after adding `number`, to see whether it is positive. This approach is insufficient. First, notice that it is possible to add a negative value to an entry of `incomeData` without that entry becoming negative. For example, if `number` is `-4,000` and the corresponding entry in `incomeData` is `10,000`, the sum is `6,000`. Thus, a negative value for `number` could remain undetected and invalidate the results of the rest of the program.

Test for invalid input data

One possible course of action for the method to take when it detects invalid data is to raise an exception. If this is done when the bad input is detected, the exception is thrown to the point in the program where the method was called, and no further input is accepted from the user. Another possibility is for the method to set an error flag, ignore the bad input line, and continue taking input from the user. When the user has completed data entry, the method could either throw an exception back to the calling code, or simply return a Boolean value of false to indicate that an input error has occurred. If the input error is a rare event, using an exception would be a good way to handle this situation. But if the input errors are common, and it is acceptable to remove erroneous input from the results, the return of the Boolean value is a better choice.

The following *readData* method attempts to be as universally applicable as possible and to make the program that uses it as modifiable as possible. When the method encounters an error in input, it sets a flag, ignores the data line, and continues. By setting a flag and later returning its value, the method leaves it to the calling module to determine the appropriate action—such as abort or continue—when an input error occurs. Thus, you can use the same input method in many contexts and can easily modify the action taken upon encountering an error.

```
public boolean readData() {
    // -----
    // Reads and organizes income statistics.
    // Precondition: The calling code gives directions and
    // prompts the user. Each input line contains exactly two
    // integers in the form G N, where N is the number of
    // people with an income in the G-thousand-dollar group
    // and LOW-END <= G <= HIGH-END. An input line with
    // values of zero for both G and N terminates the input.
    // Postcondition: incomeData[G-LOW-END] = total number
    // of people with an income in the G-thousand-dollar
    // group. The values read are displayed. If either
    // G or N is erroneous (G and N are not both 0, and
    // either G < LOW-END, G > HIGH-END, or N < 0),
    // the method prints a message indicating the line
    // will be ignored, sets the return value to false, and
    // continues. In this case, the calling code should take
    // action. The return value is true if the data is error
    // free.
    // -----
    int group, number;          // input values
    boolean dataCorrect = true; // no data error found yet
    Scanner input = new Scanner(System.in);

    for (group = LOW-END; group <= HIGH-END; ++group)  {
        // clear array
        incomeData[index(group)] = 0;
    } // end for
```

A method
that includes
fail-safe
programming

```

group = input.nextInt();
number = input.nextInt();

while ((group != 0) || (number != 0)) {
    // Invariant: group and number are not both 0
    System.out.print("Income group "+group+" contains " +
                      number + " people. ");
    if ((group >= LOW_END) && (group <= HIGH_END) &&
        (number >=0)) {
        incomeData[index(group)] += number;
        System.out.println();
    }
    else {
        System.out.println("Data not valid - ignored.");
        dataCorrect = false;
    } // end if

    group = input.nextInt();
    number = input.nextInt();

} // end while
return dataCorrect;
} // end readData

```

Although this input method will behave gracefully in the face of most common input errors, it is not completely fail-safe. What happens if an input line contains only one integer? What happens if an input line contains a noninteger? The method would be more fail-safe if it read its input one line at a time, and verified that the line actually contains two integer values by using the *hasNextInt()* method from the *Scanner* class. In some contexts, this processing would be a bit extreme. However, if the people who enter the data frequently err by typing nonintegers, you could alter the input method easily because the method is an isolated module. In any case, the method's initial comments should include any assumptions it makes about the data and an indication of what might make the program abort abnormally.

Guarding against errors in program logic. Now consider the second type of error that a program should guard against: errors in its own logic. These are errors that you may not have caught when you debugged the program or that you may have introduced through program modification.

Unfortunately, a program cannot reliably let you know when something is wrong with it. (Could you rely on a program to tell you that something is wrong with its mechanism for telling you that something is wrong?) You can, however, build into a program checks that ensure that certain conditions always hold when the program is correctly implementing its algorithm. As was mentioned earlier, such conditions are called invariants.

As a simple example of an invariant, consider again the previous example. *All integers in the array incomeData must be greater than or equal to zero.* Although the previous discussion argued that the method `readData` should not check the validity of the entries of `incomeData` instead of checking `number`, it could do so *in addition to* checking `number`. For example, if the method finds that an element in the array `incomeData` is outside some range of believability, it can signal a potential problem to its users.

Another general way in which you should make a program fail-safe is to make each method check its precondition. For example, consider the following method, `factorial`, which returns the factorial of an integer:

```
public static int factorial(int n) {
    // -----
    // Computes the factorial of an integer.
    // Precondition: n >= 0.
    // Postcondition: Returns n * (n-1)*...*1, if n > 0;
    // returns 1 if n = 0.
    // -----
    int result = 1;

    for (int i = n; i > 1; --i) {
        result *= i;
    } // end for
    return result;
} // end factorial
```

The initial comments in this method contain a precondition—information about what assumptions are made—as should always be the case. The value that this method returns is valid only if the precondition is met. If `n` is less than zero, the method will return the incorrect value of 1.

In the context of the program for which this method was written, it may be reasonable to make the assumption that `n` will never be negative. That is, if the rest of the program is working correctly, it will call `factorial` only with correct values of `n`. Ironically, this last observation gives you a good reason for `factorial` to check the value of `n`: If `n` is less than zero, the warning that results from the check indicates that something may be wrong elsewhere in the program.

Another reason the method `factorial` should check whether `n` is less than zero is that the method should be correct outside the context of its program. That is, if you borrow the method for use in another program, the method should warn you if you use it incorrectly by passing it an `n` that is negative. A stronger check than simply the statement of the precondition in a comment is desirable. Thus, *a method should state its assumptions and, when possible, check that its arguments conform to these assumptions.*

Methods should check their invariants

Methods should enforce their preconditions

Methods should check the values of their arguments

In this example, `factorial` could check the value of `n` and, if it is negative, return zero, since factorials are never zero. The program that uses `factorial` could then check for this unusual value.

Alternatively, `factorial` could abort execution if its argument was negative. Many programming languages, such as Java, support a mechanism for error handling called an exception. As discussed in Chapter 1, a module indicates that an error has occurred by throwing an exception. A module reacts to an exception that another module throws by catching the exception and executing code to deal with the error condition. Error handling is discussed further in the next section about programming style.

Style

This section considers the following five issues of personal style in programming:

KEY CONCEPTS

Five Issues of Style

1. Extensive use of methods
 2. Use of private data fields
 3. Error handling
 4. Readability
 5. Documentation
-

Admittedly, much of the following discussion reflects the personal taste of the authors; certainly other good programming styles are possible.

It is difficult to overuse methods

Extensive use of methods. It is difficult to overuse methods. If a set of statements performs an identifiable, recurring task, it should be a method. However, a task need not be recurrent to justify the use of a method.

Although a program with all its code in-line runs faster than one that calls methods, programs without methods are not cheaper to use. The use of methods is cost-effective if you consider human time as a significant component of the program's cost. You have already seen the advantages of a modular program.

Use of private data fields. Each object has a set of methods that represents the operations that can be performed on the object. The object also contains data fields to store information within the object. You should hide the exact representation of these data fields from modules that use the object by making all of the data fields private. Doing so supports the principle of information hiding. The details of the object's implementation are hidden from view, with methods providing the only mechanism to get information to and from the object. Even in a situation where the only operations involved with a particular data field are read and write, the object should

provide a simple method—called an **accessor**—that returns the data field's value and another method—called a **mutator**—that sets the data field's value. For example, a `Person` object could provide access to the data field `theName` through the methods `getname()` to return the person's name and `setName()` to change the person's name.

Error handling. A fail-safe program checks for errors in both its input and its logic and attempts to behave gracefully when it encounters them. A method should check for certain types of errors, such as invalid input or argument values. What action should a method take when it encounters an error? Depending on context, the appropriate action in the face of an error can range from ignoring erroneous data and continuing execution to terminating the program. The method `readData` in the income statistics program earlier in this chapter returned a boolean value to the calling module to indicate that it had encountered an invalid line of data. Thus, the method left it to the calling module to decide on the appropriate action. In general, methods should return a value or throw an exception instead of displaying a message when an error occurs.

In some situations, it is more appropriate for the method itself to take the action in case of an error. In the case of a fatal error that calls for termination of the program, Java provides a class `java.lang.Error`. The program will throw an object of type `java.lang.Error` when the error encountered in the program is too severe to warrant continued execution of the program. Dividing an integer by zero is a simple example of a situation that will cause the program to terminate abnormally in this manner.

In case of an error,
methods should
return a value or
throw an exception,
but not display a
message

Readability. For a program to be easy to follow, it should have a good structure and design, a good choice of identifiers, good indentation and use of blank lines, and good documentation. Avoid clever programming tricks that save a little computer time at the expense of much human time. You will see examples of these points in programs throughout the book.

Choose identifiers that describe their purpose, that is, are self-documenting. Distinguish between keywords, such as `int`, and user-defined identifiers. This book uses the following conventions:

- Keywords are lowercase and appear in boldface.
- User-defined identifiers use both upper- and lowercase letters, as follows:
 - Class names are nouns, with each word in the identifier capitalized.
 - Method names are verbs, with the first letter lowercase and subsequent internal words capitalized. Variables begin with a lowercase letter. Remaining words in multiple-word identifiers each begin with an uppercase letter.
 - Named constants are entirely uppercase and use underscores to separate words.

Identifier style

Use a good indentation style to enhance the readability of a program. The layout of a program should make it easy for a reader to identify the

program's modules. Use blank lines to offset each method. Also, within methods, you should indent individual blocks of code visibly and offset them with blank lines. These blocks are generally—but are not limited to—the actions performed within a control structure, such as a *while* loop or an *if* statement.

You can choose from among several good indentation styles. The five most important general requirements of an indentation style are as follows:

Guidelines for indentation style

- Blocks should be indented sufficiently so that they stand out clearly.
- Indentation should be consistent: Always indent the same type of construct in the same manner.
- The indentation style should provide a reasonable way to handle the problem of rightward drift, the problem of nested blocks bumping against the right-hand margin of the page.
- In a compound statement, the opening brace should be at the end of the line that begins the compound statement; the closing brace should line up with the beginning of the line that begins the compound statement:

```
while (i > 0) {  
    statement(s)  
} // end while
```

- Braces are used around all statements, even single statements, when they are part of a control structure, such as an *if-else* or *for* statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

Within these guidelines there is room for personal taste. Here is a summary of the style you will see in this book.

Indentation style in this book

- A *for* or *while* statement is written for a simple or compound action as

```
while (expression) {  
    statement(s)  
} // end while
```

- A *do* statement is written for a simple or compound action as

```
do {  
    statement(s)  
} while (expression);
```

- An *if* statement is written for simple or compound actions as

```
if (expression) {  
    statement(s)  
}  
else {  
    statement(s)  
} // end if
```

Nested *if* statements that choose among three or more different courses of action are written as

```
if (condition1) {  
    action1  
}  
else if (condition2) {  
    action2  
}  
else if (condition3) {  
    action3  
} // end if
```

This indentation style better reflects the nature of the construct, which is like a generalized *switch* statement:

```
switch (expression) {  
    case constant1 : action1; break;  
    case constant2 : action2; break;  
    case constant3 : action3; break;  
}  
// end switch
```

Documentation. A program should be well documented so that others can read, use, and modify it easily. Many acceptable styles for documentation are in use today, and exactly what you should include often depends on the particular

program or your individual circumstances. The following are the essential features of any program's documentation:

KEY CONCEPTS**Essential Features of Program Documentation**

1. An initial comment for the program that includes
 - a. Statement of purpose
 - b. Author and date
 - c. Description of the program's input and output
 - d. Description of how to use the program
 - e. Assumptions such as the type of data expected
 - f. Statement of exceptions, that is, what could go wrong
 - g. Brief description of the major classes
2. Initial comments in each class that state its purpose and describe the data contained in the class (constants and variables)
3. Initial comments in each method that state its purpose, preconditions, postconditions, and methods called
4. Comments in the body of each method to explain important features or subtle logic

Consider who will read your comments when you write them

You benefit from your own documentation by writing it now instead of later

Beginning programmers tend to downplay the importance of documentation because the computer does not read comments. By now, you should realize that people also read programs. Your comments must be clear enough for someone else to either use your method in a program or modify it. Thus, some of your comments are for people who want to use your method, while others are for people who will revise its implementation. You should distinguish between different kinds of comments.

Beginners have a tendency to document programs as a last step. You should, however, write documentation as you develop the program. Since the task of writing a large program might extend over a period of several weeks, you may find that the method that seemed so obvious when you wrote it seems confusing when you try to revise it a week later. Why not benefit from your own documentation by writing it now, while it's fresh in your mind?

Debugging

No matter how much care you take in writing a program, it will contain errors that you need to track down. Fortunately, programs that are modular, clear, and well documented are generally amenable to debugging. Fail-safe techniques, which guard against certain errors and report them when they are encountered, are also a great aid in debugging.

Many students seem to be totally baffled by errors in their programs and have no idea how to proceed. These students simply have not learned to track

down errors systematically. Without a systematic approach, finding a small mistake in a large program can indeed be a difficult task.

The difficulty that many people have in debugging a program is perhaps due in part to a desire to believe that their program is really doing what it is supposed to do. For example, on receiving an execution-time error message at line 1098, a student might say, "That's impossible. The statement at line 1098 was not even executed, because it is in the *else* clause, and I am positive that it was not executed." This student must do more than simply protest. The proper approach is either to trace the program's execution by using available debugging facilities or to add *System.out.println* statements that show which part of the *if* statement was executed. By doing so, you verify the value of the expression in the *if* statement. If the expression is 0 when you expect it to be 1, the next step is to determine how it became 0.

How can you find the point in a program where something becomes other than what it should be? Typically, a programming environment allows you to trace a program's execution either by single-stepping through the statements in the program or by setting breakpoints at which execution will halt. You also can examine the contents of particular variables by either establishing watches or inserting temporary *System.out.println* statements. The key to debugging is simply to use these techniques to tell you what is going on. This may sound pretty mundane, but the real trick is to use these debugging aids in an effective manner. After all, you do not simply put breakpoints, watches, and *System.out.println* statements at random points in the program and have them report random information.

The main idea is systematically to locate the points of the program that cause the problem. A program's logic implies that certain conditions should be true at various points in the program. (Recall that these conditions are called invariants.) If the program's results differ from your expectations as stated in the invariants (you *did* write invariants, didn't you?), an error occurs. To correct the error, you must find the first point in the program at which this difference is evident. By inserting either breakpoints and watches or *System.out.println* statements at strategic locations of a program—such as at the entry and departure points of loops and methods—you can systematically isolate the error.

These diagnostic techniques should inform you whether things start going wrong before or after a given point in the program. Thus, after you run the program with an initial set of diagnostics, you should be able to trap the error between two points. For example, suppose that things are fine before you call method *M*₁, but something is wrong by the time you call *M*₂. This kind of information allows you to focus your attention between these two points. You continue the process until eventually the search is limited to only a few statements. There is really no place in a program for an error to hide.

The ability to place breakpoints, watches, and *System.out.println* statements in appropriate locations and to have them report appropriate information comes in part from thinking logically about the problem and in part from experience. Here are a few general guidelines.

Use either watches or temporary *System.out.println* statements to find logic

Systematically check a program's logic to determine where an error occurs

Debugging methods. You should examine the values of a method's arguments at its beginning and end by using either watches or `System.out.println` statements. Ideally, you should debug each major method separately before using it in your program.

Debugging loops. You should examine the values of key variables at the beginnings and ends of loops, as the comments in this example indicate:

```
// check values of start and stop before entering loop
for (index = start; index <= stop; ++index) {
    // check values of index and key variables
    // at beginning of iteration
    .
    .
    .
    // check values of index and key variables
    // at end of iteration
} // end for
// check values of start and stop after exiting loop
```

Debugging *if* statements. Just before an *if* statement, you should examine the values of the variables within its expression. You can use either breakpoints or *System.out.println* statements to determine which branch the *if* statement takes, as this example indicates:

```
// check variables within expression before executing if
if (expression) {
    System.out.println("Value of expression is true");
    ...
}
else {
    System.out.println("Value of expression is false");
    ...
}
// end if
```

Using `System.out.println` statements. `System.out.println` statements can sometimes be more convenient than watches. Such statements should report both the values of key variables and the location in the program at which the variables have those values. You can use a comment to label the location, as follows:

```
// This is point A  
System.out.println("At point A method compute:\n" +  
    "x = " + x + ", y = " + y);
```

Remember to either comment out or remove these statements when your program finally works.

Using special dump methods. Often the variables whose values you wish to examine are arrays or other, more complex data structures. If so, you should write dump methods to display the data structures in a highly readable manner. You can easily move the single statement that calls each dump method from one point in the program to another as you track down an error. The time that you spend on these methods often proves to be worthwhile, as you can call them repeatedly while debugging different parts of the program.

Hopefully, this discussion has conveyed the importance of the *effective use of diagnostic aids in debugging*. Even the best programmers have to spend some time debugging. Thus, to be a truly good programmer, you must be a good debugger.

Summary

1. Software engineering is a branch of computer science that studies ways to facilitate the development of computer programs.
2. The life cycle of software consists of several phases: specifying the problem, designing the algorithm, analyzing the risks, verifying the algorithm, coding the programs, testing the programs, refining the solution, using the software, and maintaining the software.
3. A loop invariant is a property of an algorithm that is true before and after each iteration of a loop. Loop invariants are useful in developing iterative algorithms and establishing their correctness.
4. Java supports an `assert` statement that allows you to verify that a condition is true at a given point in a program.
5. When evaluating the quality of a solution, you must consider a diverse set of factors: the solution's correctness, its efficiency, the time that went into its development, its ease of use, and the cost of modifying and expanding it.
6. A combination of object-oriented and functional decomposition techniques will lead to a modular solution. For problems that primarily involve data management, encapsulate data with operations on that data by designing classes. The nouns in the problem statement can help you to identify appropriate classes. Break algorithmic tasks into independent subtasks that you gradually refine. In all cases, practice abstraction; that is, focus on what a module does instead of how it does it.
7. Take great care to ensure that your final solution is as easy to modify as possible. Generally, a modular program is easy to modify because changes in the problem's requirements frequently affect only a handful of the modules. Programs should not depend on the particular implementations of its modules.
8. A method should be as independent as possible and perform one well-defined task.

9. A method should always include an initial comment that states its purpose, its precondition—that is, the conditions that must exist at the beginning of a module—and its postcondition—the conditions at the end of a module.
10. A program should be as fail-safe as possible. For example, a program should guard against errors in input and errors in its own logic. By checking invariants—which are conditions that are true at certain points in a program—you can monitor correct program execution.
11. The effective use of available diagnostic aids is one of the keys to debugging. You should use watches or `System.out.println` statements to report the values of key variables at key locations. These locations include the beginnings and ends of methods and loops, and the branches of selection statements.
12. To make it easier to examine the contents of arrays and other, more complex data structures while debugging, you should write dump methods that display the contents of the data structures. You can easily move calls to such methods as you track down an error.

Cautions

1. Your programs should guard against errors. A fail-safe program checks that an input value is within some acceptable range and reports if it is not. An error in input should not cause a program to terminate before it clearly reports what the error was. A fail-safe program also attempts to detect errors in its own logic. For example, in many situations, methods should check that their arguments have valid values.
2. If you want to use the `assert` statement in a Java program, you must make sure that assertions are enabled when you compile the code.
3. You can write better, correct programs in less time if you pay attention to the following guidelines: Write precise specifications for the program. Use a modular design. Write pre- and postconditions for each method before you implement it. Use meaningful identifiers and consistent indentation. Write comments, including assertions and invariants.

Self-Test Exercises

The answers to all Self-Test Exercises are at the back of this book.

1. Think about the way that a cell phone manages a contact list.
 - a. Write a specification for this problem. Be sure to account for the fact that there may be many different ways to contact a single person, including e-mail.
 - b. Design a solution to this problem. Design at least one class.

2. What is the loop invariant for the following?

```
int index = 0;
int sum = item[0];

while (index < n) {
    ++index;
    sum += item[index];
} // end while
```

3. What is the loop invariant for the following? (Hint: Convert to a `while` loop first.)

```
for (int index = 0; index < n; index++) {
    sum += item[index];
}
```

4. Consider the following method, which interactively reads and writes the identification number, age, salary (in thousands of dollars), and name of each individual in a group of employees. How can you improve the program? Some of the issues are obvious, while others are more subtle. Try to keep in mind all the topics discussed in this chapter.

```
public static void main(String args[]) {
    int x1, x2, x3;
    String name;
    Scanner input = new Scanner(System.in);

    x1 = input.nextInt();
    x2 = input.nextInt();
    x3 = input.nextInt();

    while (x1 != 0) {
        name = input.next();
        System.out.println(x1 + " " + x2 + " " + x3 + name);

        x1 = input.nextInt();
        x2 = input.nextInt();
        x3 = input.nextInt();
    } // end while

} // end main
```

5. Suppose that, due to some severe error, you must abort a program from a location deep inside nested method calls, while loops, and if statements. Write a diagnostic method that you can call from anywhere in a program. This method should take an error code as an argument, display an appropriate error message, and terminate program execution.

Exercises

1. The greatest common divisor (GCD) of two integers is the largest integer that divides both numbers without a remainder. Write a recursive method that computes the GCD of two integers. The method should take two positive integers as arguments and return the GCD of the two integers.

2. The price of an item that you want to buy is given in dollars and cents. You pay for it in cash by giving the clerk d dollars and c cents. Write specifications for a method that computes the change, if any, that you should receive. Include a statement of purpose, the pre- and postconditions, and a description of the parameters.
3. The time of day consists of an hour, minute, second, and time zone. Integer values can be used to represent the hour (0-23) and minute (0-59). Seconds can be represented using a decimal value depending on how often the time is updated (for example, every millisecond). The time zone can be represented using a string (for example, 2:30 p.m. Eastern time is 14 hours, 30 minutes with a time zone of "EDT." The seconds are not noted here, but will be a value between 0 and 60—for example, 03.120 is 3 seconds and 120 milliseconds.
 - a. Write specifications for a method that advances any given date by one day. Include a statement of purpose, the pre- and postconditions, and a description of the parameters.
 - b. Write a Java implementation of this method. Design and specify any other methods that you need. Include comments that will be helpful to someone who will maintain your implementation in the future.
4. The following program counts the number of occurrences of a given word in a file, but has many problems. It is written using poor style, and it contains syntax errors and even some logic errors. Identify and provide corrections for these problems so that the program follows the style guideline in this chapter and executes properly.

```
import java.io.*;

public class countWord {

    static final String
        prompt1 = "Enter the name of the text file: ";
    static final String
        PROMPT2 = "Enter the word to be counted in the file: ";

    public static void main(String args) {
        // Purpose: To count the number of occurrences of a word in a
        // text file. The user provides both the word and the text
        // file.

        Scanner input = new Scanner(System.out);

        String Filename = input.nextLine();
        System.out.print(prompt1);

        Scanner fileInput = null;
        try {
            fileInput = new Scanner(new File(Filename));
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        // Use anything but a letter as a delimiter
        fileInput.useDelimiter("[^a-zA-Z]+");
```

```
String word = input.next();
System.out.print(PROMPT2);

while (fileInput.hasNext()) {
    String fileWord = fileInput.next();
    System.out.println(fileWord);
    int color = 0;
    if (Word.equalsIgnoreCase(fileWord)) {
        color++;
    }
} // end while
fileInput.close();
System.out.println("The word " + word + " appeared "
    + color + " times in the file" + Filename);
}
} // end countWord
```

5. Social networking sites make it easy for people to stay in touch. Suppose you need to design a very simple social networking program that allows a user to post messages to all of his or her friends, so that when one of the user's friends logs into the program, the friend sees the message. Assume that friends must be confirmed, and that once a friendship is confirmed, each user is a friend of the other.
 - a. Write a specification for this problem. Note that a person may have many friends.
 - b. Design a solution to this problem. Determine what objects (with their data fields and methods) would be necessary to solve this problem.
 - c. Draw a UML diagram that reflects your design in part b.
6. Draw a UML diagram for an `Automobile` class that could be used by a car rental company. The data members should include the make, model, year, license plate, mileage, amount of gasoline in the tank, and current location. When deciding on the operations to include, be aware that only some of these data members should be updateable, while others are read only.
7. Design a student registration system that manages student enrollment in courses. Each course can have many students enrolled in it, but only one instructor. Information stored on students and instructors is similar in that they all have a name, home address, and ID. Students also have their campus address and major stored, while instructors have department and salary. Courses have a title, course code, meeting time, and a list of the students enrolled in the class. Model this application using UML.
8. One way to compute the quotient and remainder of two numbers `num` and `den` is to use repeated subtraction as shown in the following code:

```
int q = 0;      // the quotient
int rem = num; // the remainder

while (rem >= den) {
    rem = rem - den;
    q++;
} // end while
```

The invariant for this code is

```
rem >= 0 and
num = q * den + rem
```

For the values $num = 17$, $den = 4$, prove that the invariant is true before the loop begins, after each iteration (pass) of the loop, and when the loop terminates by completing the following table (you may not need all passes shown). The initial values are shown:

pass	q	rem	rem >= 0	num = q * den + rem
initially	0	17	true	$17 = 0 * 17 + 17$, true
1				
2				
3				
4				
5				

9. Write pre- and postconditions for the following methods:
 - a. A method that takes an array of test scores and returns the average
 - b. A method that computes a person's Body Mass Index (BMI), given the height in inches and the body weight in pounds
 - c. A method that computes the monthly payment for a loan given the loan amount, the interest rate, and the loan term in months
 10. Do you think that the assertion statement in Java can help you debug your program? Explain your answer.
 11. What is the problem with the following code fragment?
- ```
num = 50;
while (num >= 0) {
 System.out.println (num);
 num = num + 1;
} // end while
```
12. Add a **transaction** class to the banking example in the section "Modeling Object-Oriented Designs Using UML." This class keeps track of the date, time, amount, and type of transaction (checking or savings).
  13. This chapter stressed the importance of adding fail-safe checks to a program wherever possible. What can go wrong with the following method? How can you protect yourself?

```
public static double getAverage(double[] x, int numItems) {
 double sum = 0;
 for (int i=0; i<numItems; i++){
 sum += x[i];
 } // end for
 return sum/numItems;
} // end getAverage
```

14. Write the loop invariants for the code segments shown:

a. 

```
int i = 10;
while (i < 100)
 i++;
```

b. 

```
int product = 1;
for (int i = 2; i <= n; i++)
 product = product * (2 * i - 1);
```

c. 

```
int c = 0;
int p = 1;
while (c < b) {
 p = p * a;
 c++;
} // end while
```

15. In the following code, assume that the array item has been initialized with random integer values. Write the loop invariant for the following:

```
int index = 0;
while (index < item.length)
 if (item[index] > 0) {
 sum += item[index];
 } // end if
 ++index;
} // end while
```

16. Using a **for** loop, write a program that displays the squares of prime numbers greater than 0 and less than or equal to a given number *n*.

17. The following code is supposed to compute the floor of the square root of its input value *x*. (The floor of a number *n* is the largest integer less than or equal to *n*.)

```
// Computes and writes floor(sqrt(x)) for
// an input value x >= 0.
public static void main(String args[]) {
 int x; // input value
 Scanner input = new Scanner(System.in);

 // initialize
 int result = 0; // will equal floor of sqrt(x)
 int templ = 1;
 int temp2 = 1;

 // read input
 x = input.nextInt();

 // compute floor
 while (templ < x) {
 ++result;
 temp2 += 2;
 templ += temp2;
 } // end while
 System.out.println("The floor of the square root of "
 + x + " is " + result);

} // end main
```

This program contains an error.

- a. What output does the program produce when  $x = 64$ ?
  - b. Run the program and remove the error. Describe the steps that you took to find the error.
  - c. How can you make the program more user friendly and fail-safe?
18. As part of a larger banking application, you have been asked to consider the task of completing a loan application. Use functional decomposition to break down the task into more manageable subtasks.
19. Suppose that you are writing a program that asks the user their age. Write a segment of code that would verify that an integer value of an appropriate age range has been entered. If the user enters something other than an integer, or the integer is less than zero or is greater than 122 (the oldest documented age on record), the user should be prompted to enter his or her age again.

## Programming Problems

1. Consider a program that will read student information into an array of objects, sort the array by student identification number, write out the sorted array, and compute various statistics on the data, such as the average GPA of a student. Write complete UML specifications for this problem that reflect an object-oriented solution. What classes and methods did you identify during the design of your solution? Write specifications, including preconditions and postconditions, for each method.
2. Write a program that sorts and evaluates bridge hands.

The input is a stream of character pairs that represent playing cards. For example,

2C QD TC AD 6C 3D TD 3H 5H 7H AS JH KH

represents the 2 of clubs, queen of diamonds, 10 of clubs, ace of diamonds, and so on. Each pair consists of a rank followed by a suit, where rank is A, 2, . . . , 9, T, J, Q, or K, and suit is C, D, H, or S. You can assume that each input line represents exactly 13 cards and is error-free. Input is terminated by an end-of-file.

For each line of input, form a hand of 13 cards. Display each hand in a readable form arranged both by suits and by rank within suit (aces are high). Then evaluate the hand by using the following standard bridge values:

- Aces count 4
- Kings count 3
- Queens count 2
- Jacks count 1
- Voids (no cards in a suit) count 3
- Singletons (one card in a suit) count 2
- Doubletons (two cards in a suit) count 1
- Long suits with more than 5 cards in the suit  
count 1 for each card over 5 in number

For example, for the previous sample input line, the program should produce the output

```

CLUBS 10 6 2
DIAMONDS A Q 10 3
HEARTS K J 7 5 3
SPADES A
Points = 16

```

because there are 2 aces, 1 king, 1 queen, 1 jack, 1 singleton, no doubletons, and no long suits. (The singleton ace of spades counts as both an ace and a singleton.)

*Optional.* See how much more flexible and fail-safe you can make your program. That is, try to remove as many of the previous assumptions in input as you can.

3. A *polynomial* of a single variable  $x$  with integer coefficients is an expression of the form

$$p(x) = c_0 + c_1x + c_2x^2 + \dots + c_nx^n,$$

where  $c_i, i = 0, 1, \dots, n$ , are integers.

Create a class for polynomials up to the  $n^{th}$  degree. A specification of the methods for this class is provided next:

■ **public Polynomial(int maxDegree)**

Constructs a new polynomial of degree `maxDegree` with all of the coefficients set to zero

■ **public Polynomial(int[] coef)**

Constructs a new polynomial with the corresponding coefficients passed in the `coef` array, with the highest degree as `coef[0]` and the constant term in `coef[coef.length-1]`. So the array [3, 2, 1] creates the polynomial  $3x^2 + 2x + 1$ .

■ **public int getCoefficient(int power)**

Returns an integer representing the coefficient of the  $x^{power}$  term

■ **public void setCoefficient(int coef, int power)**

Sets the coefficient of the  $x^{power}$  term to `coef`

■ **public String toString()**

Returns the `String` representation of the polynomial. For example,  $3x^2 + 2x + 1$  would be returned as `3 * x^2 + 2 * x + 1` or, more simply, `3x^2 + 2x + 1`. Any term whose coefficient is zero should not appear in the string unless the polynomial has only a single constant term of zero.

■ **public double evaluate(double x)**

Evaluates the polynomial for the value `x` and returns the result `p(x)`

■ **public static Polynomial derivative(Polynomial p)**

Returns a `Polynomial` representing the derivative of the polynomial `p`

■ **public double bisection(double a, double b)**

`throws java.lang.IllegalArgumentException`

Returns a `double` representing the root of the `Polynomial` using the Bisection Method. The bisection method requires two initial points `a` and `b` such that `p(a)` and `p(b)` have opposite signs; if they do not, then `java.lang.IllegalArgumentException` should be thrown. Though there could be multiple roots between `a` and `b`, the method will return the first root it finds such that evaluating the polynomial at `p(root) < 0.000001`.

One method that can be used to easily evaluate a polynomial is based upon Horner's rule. Note that the fourth degree polynomial  $2x^4 + x^3 - 4x^2 + 3x + 5$  can be evaluated as

```
result = (((((2 * x) + 1) * x) - 4) * x) + 3) * x) + 5
```

or, in more general terms

```
result = (((((c4 * x) + c3) * x) + c2) * x) + c1) * x) + c0
```

To find the derivative of a polynomial, you simply find the derivative of each term in the polynomial. The derivative of a term  $c_i x^i$  is  $(i \cdot c_i)x^{i-1}$ . So for example, the derivative of

$$p(x) = 4x^2 + 3x + 1 \text{ is}$$

$$p'(x) = 8x + 3$$

To test the *Polynomial* class, create a second class called *TestPolynomial*. This class should support the following interaction with the user (user input is shaded):

Please enter the polynomial you wish to work with. You will be prompted to enter the coefficient for each term in the polynomial. You may enter zero if the term is absent from the polynomial.

What degree polynomial would you like to create? 4

Please enter the coefficients:

Coefficient for  $x^4$ : 2

Coefficient for  $x^3$ : 1

Coefficient for  $x^2$ : -4

Coefficient for  $x^1$ : 3

Coefficient for  $x^0$ : 5

$p(x) = 2x^4 + x^3 - x^2 + 3x + 5$

What would you like to do with this polynomial?

(E/e) Evaluate it for a particular value of x

(D/d) Get the derivative of the polynomial

(R/r) Find the root of the polynomial

Enter option: e

For what value of x would you like to evaluate the polynomial?

2.5

$p(2.5) = 81.25$

Would you like to select another option? (y/n) y

What would you like to do with this polynomial?

(E/e) Evaluate it for a particular value of x

(D/d) Get the derivative of the polynomial

(R/r) Find the root of the polynomial

```

Enter option: d
p'(x) = 8 x^3 + 3 x^2 - 8 x + 3

Would you like to select another option? (y/n) n
Would you like to enter another polynomial? (y/n) n

```

4. Java provides a class `java.lang.BigInteger` that can be used to handle very large integers. Implement a similar class, called `BigInt`, that can be used to do simple calculations with very large nonnegative integers. Design this class carefully. You will need the following:

- A data structure to represent large numbers: for example, a string or an array for the digits in a number.
- **public BigInt(String val)**  
A constructor that uses a string representation of the integer for initialization. The string may contain leading zeros. Do not forget that zero is a valid number.
- **public String toString()**  
Returns the string representation of this `BigInt`. It should not include leading zeros, but if the number consists of all zeros, it should return a String with a single zero.
- **public BigInt max(BigInt val)**  
A method that returns a `BigInt` whose value is the maximum of `val` and the instance of `BigInt` that invokes `max`.
- **public BigInt min(BigInt val)**  
A method that returns a `BigInt` whose value is the minimum of `val` and the instance of `BigInt` that invokes `min`.
- **public BigInt add(BigInt val)**  
A method that returns a `BigInt` whose value is the sum of `val` and the instance of `BigInt` that invokes `add`.
- **public BigInt multiply(BigInt val)**  
A method that returns a `BigInt` whose value is the product of `val` and the instance of `BigInt` that invokes `multiply`.

Write a program that acts as an interactive calculator capable of handling very large nonnegative integers that uses the `BigInt` class. This calculator need perform only the operations of addition and multiplication.

In this program each input line is of the form

`num1 op num2`

and should produce output such as

```

num1
op num2

num3

```

where `num1` and `num2` are (possibly very large) nonnegative integers, `op` is the single character `+` or `*`, and `num3` is the integer that results from the desired calculation. Be sure your interface is user friendly.

*Optional:* Allow signed integers (negative as well as positive integers), and write a method for subtraction.

## CHAPTER 3

# Recursion: The Mirrors

The goal of this chapter is to ensure that you have a basic understanding of recursion, which is one of the most powerful techniques of solution available to the computer scientist. This chapter assumes that you have had little or no previous introduction to recursion. If, however, you already have studied recursion, you can review this chapter as necessary.

By presenting several relatively simple problems, the chapter demonstrates the thought processes that lead to recursive solutions. These problems are diverse and include examples of counting, searching, and organizing data. In addition to presenting recursion from a conceptual viewpoint, this chapter discusses techniques that will help you to understand the mechanics of recursion. These techniques are particularly useful for tracing and debugging recursive methods.

Some recursive solutions are far more elegant and concise than the best of their nonrecursive counterparts. For example, the classic Towers of Hanoi problem appears to be quite difficult, yet it has an extremely simple recursive solution. On the other hand, some recursive solutions are terribly inefficient, as you will see, and should not be used.

Chapter 6 continues the formal discussion of recursion by examining more-difficult problems. Recursion will play a major role in many of the solutions that appear throughout the remainder of this book.

### 3.1 Recursive Solutions

A Recursive Valued Method:  
The Factorial of  $n$

A Recursive void Method: Writing a String Backward

### 3.2 Counting Things

Multiplying Rabbits (The Fibonacci Sequence)

Organizing a Parade

Mr. Spock's Dilemma (Choosing  $k$  out of  $n$  Things)

### 3.3 Searching an Array

Finding the Largest Item in an Array  
Binary Search

Finding the  $k^{\text{th}}$  Smallest Item in an Array

### 3.4 Organizing Data

The Towers of Hanoi

### 3.5 Recursion and Efficiency

Summary

Cautions

Self-Test Exercises

Exercises

Programming Problems

### 3.1 Recursive Solutions

Recursion is an extremely powerful problem-solving technique. Problems that at first appear to be quite difficult often have simple recursive solutions. Like top-down design, recursion breaks a problem into several smaller problems. What is striking about recursion is that these smaller problems are of *exactly the same type* as the original problem—mirror images, so to speak.

Did you ever hold a mirror in front of another mirror so that the two mirrors face each other? You will see many images of yourself, each behind and slightly smaller than the other. Recursion is like these mirror images. That is, a recursive solution solves a problem by solving a smaller instance of the same problem! It solves this new problem by solving an even smaller instance of the same problem. Eventually, the new problem will be so small that its solution will be either obvious or known. This solution will lead to the solution of the original problem.

For example, suppose that you could solve problem  $P_1$  if you had the solution to problem  $P_2$ , which is a smaller instance of  $P_1$ . Suppose further that you could solve problem  $P_2$  if you had the solution to problem  $P_3$ , which is a smaller instance of  $P_2$ . If you knew the solution to  $P_3$  because it was small enough to be trivial, you would be able to solve  $P_2$ . You could then use the solution to  $P_2$  to solve the original problem  $P_1$ .

Recursion can seem like magic, especially at first, but as you will see, it is a very real and important problem-solving approach that is an alternative to iteration. An iterative solution involves loops. You should know at the outset that not all recursive solutions are better than iterative solutions. In fact, some recursive solutions are impractical because they are so inefficient. Recursion, however, can provide elegantly simple solutions to problems of great complexity.

As an illustration of the elements in a recursive solution, consider the problem of looking up a word in a dictionary. Suppose you wanted to look up the word “vademecum.” Imagine starting at the beginning of the dictionary and looking at every word in order until you found “vademecum.” That is precisely what a sequential search does, and, for obvious reasons, you want a faster way to perform the search.

One such method is the **binary search**, which in spirit is similar to the way in which you actually use a dictionary. You open the dictionary—probably to a point near its middle—and by glancing at the page, determine which “half” of the dictionary contains the desired word. The following pseudocode is a first attempt to formalize this process:

A binary search of a dictionary

```
// Search a dictionary for a word by using a recursive
// binary search
```

```
if (the dictionary contains only one page) {
 Scan the page for the word
}
else {
 Open the dictionary to a point near the middle
 Determine which half of the dictionary contains the
 word
```

```

if (the word is in the first half of the dictionary) {
 Search the first half of the dictionary for the word
}
else {
 Search the second half of the dictionary for the word
} // end if
} // end if

```

Parts of this solution are intentionally vague: How do you scan a single page? How do you find the middle of the dictionary? Once the middle is found, how do you determine which half contains the word? The answers to these questions are not difficult, but they will only obscure the solution strategy right now.

The previous search strategy reduces the problem of searching the dictionary for a word to a problem of searching half of the dictionary for the word, as Figure 3-1 illustrates. Notice two important points. First, once you have divided the dictionary in half, you already know how to search the appropriate half: You can use exactly the same strategy that you employed to search the original dictionary. Second, note that there is a special case that is different from all the other cases: After you have divided the dictionary so many times that you are left with only a single page, the halving ceases. At this point, the problem is sufficiently small that you can solve it directly by scanning the single page that remains for the word. This special case is called the **base case** (or **basis** or **degenerate case**).

This strategy is one of **divide and conquer**. You solve the dictionary search problem by first *dividing* the dictionary into two halves and then *conquering* the appropriate half. You solve the smaller problem by using the same divide-and-conquer strategy. The dividing continues until you reach the base case. As you will see, this strategy is inherent in many recursive solutions.

To further explore the nature of the solution to the dictionary problem, consider a slightly more rigorous formulation.

A base case is a special case whose solution you know

A binary search uses a divide-and-conquer strategy

```
search(in theDictionary:Dictionary, in aWord: string)
```

```

if (theDictionary is one page in size) {
 Scan the page for aWord
}
```

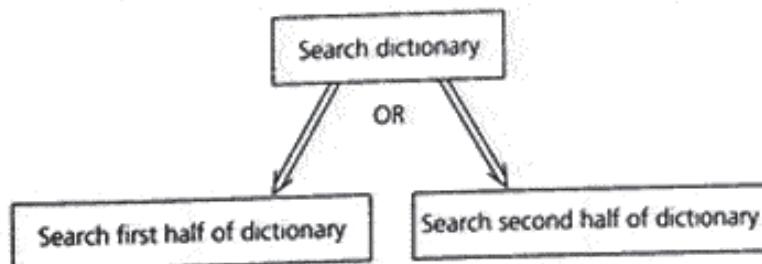


FIGURE 3-1

A recursive solution

```

 }
else {
 Open theDictionary to a point near the middle
 Determine which half of theDictionary contains aWord
 if (aWord is in the first half of theDictionary) {
 search(first half of theDictionary, aWord)
 }
 else {
 search(second half of theDictionary, aWord)
 } // end if
} // end if

```

Writing the solution as a method allows several important observations:

A recursive method calls itself

- One of the actions of the method is to call itself; that is, the method *search* calls the method *search*. This action is what makes the solution recursive. The solution strategy is to split *theDictionary* in half, determine which half contains *aWord*, and apply the same strategy to the appropriate half.

Each recursive call solves an identical, but smaller, problem

- Each call to the method *search* made from within the method *search* passes a dictionary that is one-half the size of the previous dictionary. That is, at each successive call to *search* (*theDictionary*, *aWord*), the size of *theDictionary* is cut in half. The method solves the search problem by solving another search problem that is identical in nature but smaller in size.

A test for the base case enables the recursive calls to stop

- There is one search problem that you handle differently from all of the others. When *theDictionary* contains only a single page, you use another technique: You scan the page directly. Searching a one-page dictionary is the base case of the search problem. When you reach the base case, the recursive calls stop and you solve the problem directly.
- The manner in which the size of the problem diminishes ensures that you will eventually reach the base case.

Eventually, one of the smaller problems must be the base case

These facts describe the general form of a recursive solution. Though not all recursive solutions fit these criteria as nicely as this solution does, the similarities are far greater than the differences. As you attempt to construct a new recursive solution, you should keep in mind the following four questions:

#### KEY CONCEPTS

### Four Questions for Constructing Recursive Solutions

- How can you define the problem in terms of a smaller problem of the same type?
- How does each recursive call diminish the size of the problem?
- What instance of the problem can serve as the base case?
- As the problem size diminishes, will you reach this base case?

Now consider two relatively simple problems: computing the factorial of a number and writing a string backward. Their recursive solutions further illustrate the points raised by the solution to the dictionary search problem. These examples also illustrate the difference between a recursive valued method and a recursive *void* method.

## A Recursive Valued Method: The Factorial of $n$

Consider a recursive solution to the problem of computing the factorial of an integer  $n$ . This problem is a good first example because its recursive solution is easy to understand and neatly fits the mold described earlier. However, because the problem has a simple and efficient iterative solution, you should not use the recursive solution in practice.

To begin, consider the familiar iterative definition of  $\text{factorial}(n)$  (more commonly written  $n!$ ):

$$\begin{aligned}\text{factorial}(n) &= n * (n - 1) * (n - 2) * \dots * 1 \quad \text{for any integer } n > 0 \\ \text{factorial}(0) &= 1\end{aligned}$$

The factorial of a negative integer is undefined. You should have no trouble writing an iterative factorial method based on this definition.

To define  $\text{factorial}(n)$  recursively, you first need to define  $\text{factorial}(n)$  in terms of the factorial of a smaller number. To do so, simply observe that the factorial of  $n$  is equal to the factorial of  $(n - 1)$  multiplied by  $n$ ; that is,

$$\begin{aligned}\text{factorial}(n) &= n * [(n - 1) * (n - 2) * \dots * 1] \\ &= n * \text{factorial}(n - 1)\end{aligned}$$

Do not use recursion if a problem has a simple, efficient iterative solution

An iterative definition of factorial

A recurrence relation

The definition of  $\text{factorial}(n)$  in terms of  $\text{factorial}(n - 1)$ , which is an example of a **recurrence relation**, implies that you can also define  $\text{factorial}(n - 1)$  in terms of  $\text{factorial}(n - 2)$ , and so on. This process is analogous to the dictionary search solution, in which you search a dictionary by searching a smaller dictionary in exactly the same way.

The definition of  $\text{factorial}(n)$  lacks one key element: the base case. As was done in the dictionary search solution, here you must define one case differently from all the others, or else the recursion will never stop. The base case for the factorial method is  $\text{factorial}(0)$ , which you know is 1. Because  $n$  originally is greater than or equal to zero and each call to  $\text{factorial}$  decrements  $n$  by 1, you will always reach the base case. With the addition of the base case, the complete recursive definition of the factorial method is

$$\text{factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n * \text{factorial}(n - 1) & \text{if } n > 0 \end{cases}$$

A recursive definition of factorial

To be sure that you understand this recursive definition, apply it to the computation of  $\text{factorial}(4)$ . Since  $4 > 0$ , the recursive definition states that

$$\text{factorial}(4) = 4 * \text{factorial}(3)$$

Similarly,

$$\text{factorial}(3) = 3 * \text{factorial}(2)$$

$$\text{factorial}(2) = 2 * \text{factorial}(1)$$

$$\text{factorial}(1) = 1 * \text{factorial}(0)$$

You have reached the base case, and the definition directly states that

$$\text{factorial}(0) = 1$$

At this point, the application of the recursive definition stops and you still do not know the answer to the original question: What is  $\text{factorial}(4)$ ? However, the information to answer this question is now available:

$$\text{Since } \text{factorial}(0) = 1, \text{ then } \text{factorial}(1) = 1 * 1 = 1$$

$$\text{Since } \text{factorial}(1) = 1, \text{ then } \text{factorial}(2) = 2 * 1 = 2$$

$$\text{Since } \text{factorial}(2) = 2, \text{ then } \text{factorial}(3) = 3 * 2 = 6$$

$$\text{Since } \text{factorial}(3) = 6, \text{ then } \text{factorial}(4) = 4 * 6 = 24$$

You can think of recursion as a process that divides a problem into a task that you can do and a task that a friend can do for you. For example, if I ask you to compute  $\text{factorial}(4)$ , you could first determine whether you know the answer immediately. You know immediately that  $\text{factorial}(0)$  is 1—that is, you know the base case—but you do not know the value of  $\text{factorial}(4)$  immediately. However, if your friend computes  $\text{factorial}(3)$  for you, you could compute  $\text{factorial}(4)$  by multiplying  $\text{factorial}(3)$  by 4. Thus, your task will be to do this multiplication, and your friend's task will be to compute  $\text{factorial}(3)$ .

Your friend now uses the same process to compute  $\text{factorial}(3)$  as you are using to compute  $\text{factorial}(4)$ . Thus, your friend determines that  $\text{factorial}(3)$  is not the base case, and so asks another friend to compute  $\text{factorial}(2)$ . Knowing  $\text{factorial}(2)$  enables your friend to compute  $\text{factorial}(3)$ , and when you learn the value of  $\text{factorial}(3)$  from your friend, you can compute  $\text{factorial}(4)$ .

Notice that the recursive definition of  $\text{factorial}(4)$  yields the same result as the iterative definition, which gives  $4 * 3 * 2 * 1 = 24$ . To prove that the two definitions of  $\text{factorial}$  are equivalent for all nonnegative integers, you would use mathematical induction. (See Appendix D.) Chapter 5 discusses the close tie between recursion and mathematical induction.

The recursive definition of the factorial method has illustrated two points. (1) *Intuitively*, you can define  $\text{factorial}(n)$  in terms of  $\text{factorial}(n - 1)$ , and (2) *mechanically*, you can apply the definition to determine the value of a given factorial. Even in this simple example, applying the recursive definition required quite a bit of work. That, of course, is where the computer comes in.

Once you have a recursive definition of *factorial(n)*, it is easy to construct a Java method that implements the definition:

```
public static int fact(int n) {
// -----
// Computes the factorial of a nonnegative integer.
// Precondition: n must be greater than or equal to 0.
// Postcondition: Returns the factorial of n.
// -----
 if (n == 0) {
 return 1;
 }
 else {
 return n * fact(n-1);
 } // end if
} // end fact
```

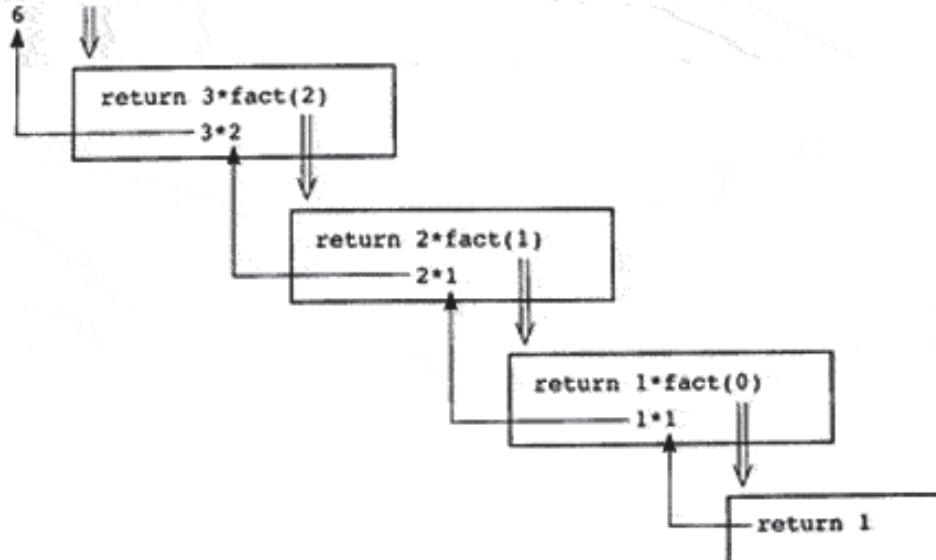
Suppose that you use the statement

```
System.out.println(fact(3));
```

to call the method. Figure 3-2 depicts the sequence of computations that this call would require.

Note that the *fact* method is defined as static. This means that *fact* is a class method; it is invoked independently of any instance of the class that contains *fact*. Instances of the class share the *fact* method, and if the static method is

```
System.out.println(fact(3));
```



**FIGURE 3-2**

`fact(3)`

public, other objects can access it through the class name. For example, `java.lang.Math.sqrt` provides access to the static method `sqrt` contained in the class `java.lang.Math`. Methods that don't need access to instance variables and are self-contained (except for parameter input) are good candidates to be designated as static methods. For this reason, all of the recursive methods in this chapter are declared `static`.

The `fact` method fits the model of a recursive solution given earlier in this chapter as follows:

1. One action of `fact` is to *call itself*.
2. At each recursive call to `fact`, the integer whose factorial you need to compute is *diminished by 1*.
3. The method handles the factorial of 0 differently from all the other factorials: It does not generate a recursive call. Rather, you know that `fact(0)` is 1. Thus, the *base case* occurs when  $n$  is 0.
4. Given that  $n$  is nonnegative, item 2 of this list assures you that you will always *reach the base case*.

`fact` satisfies the four criteria of a recursive solution

At an intuitive level, it should be clear that the method `fact` implements the recursive definition of `factorial`. Now consider the mechanics of executing this recursive method. The logic of `fact` is straightforward except perhaps for the expression in the `else` clause. This expression has the following effect:

1. Each operand of the product  $n * \text{fact}(n-1)$  is evaluated.
2. The second operand—`fact(n-1)`—is a call to the method `fact`. Although this is a recursive call (the method `fact` calls the method `fact`), there really is nothing special about it. Imagine substituting a call to another method—the Java API method `java.lang.Math.abs`, for example—for the recursive call to `fact`. The principle is the same: Simply evaluate the method.

In theory, evaluating a recursive method is no more difficult than evaluating a nonrecursive method. In practice, however, the bookkeeping can quickly get out of hand. The *box trace* is a systematic way to trace the actions of a recursive method. You can use the box trace both to help you to understand recursion and to debug recursive methods. However, such a mechanical device is no substitute for an intuitive understanding of recursion. The box trace illustrates how compilers frequently implement recursion. As you read the following description of the method, realize that each box roughly corresponds to an *activation record*, which a compiler typically uses in its implementation of a method call. Chapter 7 will discuss this implementation further.

An activation record is created for each method call

The *box trace*: The box trace is illustrated here for the `fact` method. As you will see in the next section, a box trace is a record of method calls and values needed to be returned.

|                           |                          |
|---------------------------|--------------------------|
| Initial activation record | Activation record body   |
| Method call               | Activation record occurs |

guish among them. These labels help you to keep track of the correct place to which you must return after a method call completes. For example, mark the expression `fact(n-1)` within the body of the method with the letter A:

```
if (n == 0) {
 return 1;
}
else {
 return n * fact(n-1);
} // end if A
```

Label each recursive call in the method

You return to point A after each recursive call, substitute the computed value for `fact(n-1)`, and continue execution by evaluating the expression `n * fact(n-1)`.

2. Represent each call to the method during the course of execution by a new box in which you note the method's local environment. More specifically, each box will contain

Each time a method is called, a new box represents its local environment

- The values of the references and primitive types of the method's arguments.
- The method's local variables.
- A placeholder for the value returned by each recursive call from the current box. Label this placeholder to correspond to the labeling in Step 1.
- The value of the method itself.

When you first create a box, you will know only the values of the input arguments. You fill in the values of the other items as you determine them from the method's execution. For example, you would create the box in Figure 3-3 for the call `fact(3)`. (You will see in later examples that you must handle reference arguments [objects] somewhat differently from value arguments [primitive types] and local variables.)

3. Draw an arrow from the statement that initiates the recursive process to the first box. Then, when you create a new box after a recursive call, as described in Step 2, you draw an arrow from the box that makes the call to the newly created box. Label each arrow to correspond to the label (from Step 1) of the recursive call; this label indicates exactly where to return after the call completes. For example, Figure 3-4 shows the first two boxes generated by the call to `fact` in the statement `System.out.println(fact(3))`.

|                  |
|------------------|
| n = 3            |
| A: fact(n-1) = ? |
| return ?         |

FIGURE 3-3

A box

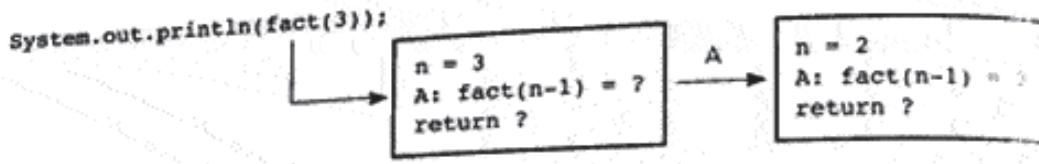


FIGURE 3-4

The beginning of the box trace

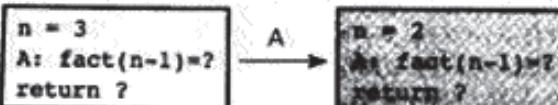
4. After you create the new box and arrow as described in Steps 2 and 3, start executing the body of the method. Each reference to an item in the method's local environment references the corresponding value in the current box, regardless of how you generated the current box.
5. On exiting the method, cross off the current box and follow its arrow back to the box that called the method. This box now becomes the current box, and the label on the arrow specifies the exact location at which execution of the method should continue. Substitute the value returned by the just terminated method call into the appropriate item in the current box.

Figure 3-5 is a complete box trace for the call `fact(3)`. In the sequence of diagrams in this figure, the current box is the deepest along the path of arrows and is shaded, whereas crossed-off boxes have a dashed outline.

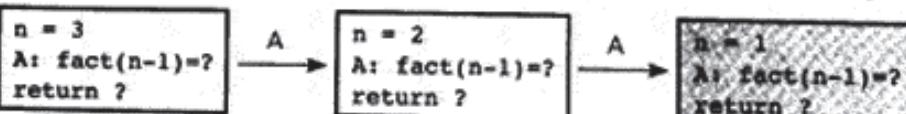
The initial call is made, and method `fact` begins execution:



At point A a recursive call is made, and the new invocation of the method `fact` begins execution:



At point A a recursive call is made, and the new invocation of the method `fact` begins execution:



At point A a recursive call is made, and the new invocation of the method `fact` begins execution:

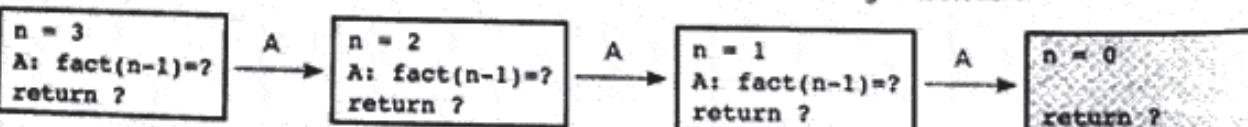


FIGURE 3-5

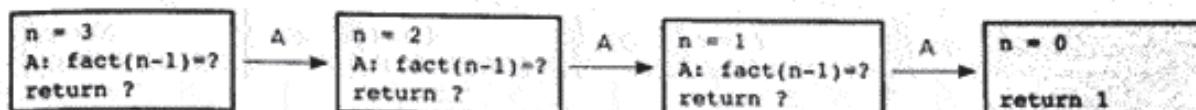
Box trace of `fact(3)`

(continues)

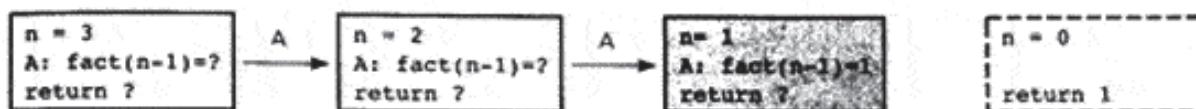
**FIGURE 3-5**

(continued)

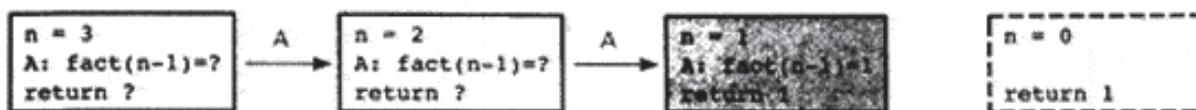
This is the base case, so this invocation of fact completes:



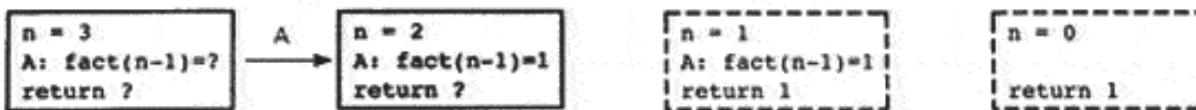
The method value is returned to the calling box, which continues execution:



The current invocation of fact completes:



The method value is returned to the calling box, which continues execution:



The current invocation of fact completes:



The method value is returned to the calling box, which continues execution:



The current invocation of fact completes:



The value 6 is returned to the initial call.

**Invariants.** Writing invariants for recursive methods is as important as writing them for iterative methods, and is often simpler. For example, consider the recursive method `fact`:

```
public static int fact(int n) {
 // Precondition: n must be greater than or equal to 0.
 // Postcondition: Returns the factorial of n.
 if (n == 0) {
 return 1;
 }
 else { // Invariant: n > 0, so n-1 >= 0.
 // Thus, fact(n-1) returns (n-1)!
 return n * fact(n-1); // n * (n-1)! is n!
 } // end if
} // end fact
```

Expect a recursive call's postcondition to be true if the precondition is true

Violating `fact`'s precondition causes "infinite" recursion

The method requires as its precondition a nonnegative value of  $n$ . At the time of the recursive call `fact(n-1)`,  $n$  is positive, so  $n - 1$  is nonnegative. Since the recursive call satisfies `fact`'s precondition, you can expect from the postcondition that `fact(n-1)` will return the factorial of  $n - 1$ . Therefore,  $n * \text{fact}(n-1)$  is the factorial of  $n$ . Chapter 6 uses mathematical induction to prove formally that `fact(n)` returns the factorial of  $n$ .

If you ever violated `fact`'s precondition, the method would not behave correctly. That is, if the calling program ever passed a negative value to `fact`, an infinite sequence of recursive calls, terminated only by a system-defined limit, would occur because the method would never reach the base case. For example, `fact(-4)` would call `fact(-5)`, which would call `fact(-6)`, and so on.

The method ideally should protect itself by testing for a negative  $n$ . If  $n < 0$ , the method could, for example, either return zero to indicate an error or throw an exception. Chapter 2 discussed error checking in the two sections "Fail-Safe Programming" and "Style"; you might want to review that discussion at this time.

## A Recursive void Method: Writing a String Backward

Now consider a problem that is slightly more difficult: Given a string of characters, write it in reverse order. For example, write the string "cat" as "tac". To construct a recursive solution, you should ask the four questions in the Key Concepts box on page 140.

You can construct a solution to the problem of writing a string of length  $n$  backward in terms of the problem of writing a string of length  $n - 1$  backward. That is, each recursive step of the solution diminishes by 1 the length of the string to be written backward. The fact that the strings get shorter and shorter suggests that the problem of writing some very short strings backward can serve as the base case. One very short string is the empty string, the string of length zero. Thus, you can choose for the base case the problem

The base case

Write the empty string backward

The solution to this problem is to do nothing at all—a very straightforward solution indeed! (Alternatively, you could use the string of length 1 as the base case.)

Exactly how can you use the solution to the problem of writing a string of length  $n - 1$  backward to solve the problem of writing a string of length  $n$  backward? This approach is analogous to the one used to construct the solution to the factorial problem, where you specified how to use *factorial*( $n - 1$ ) in the computation of *factorial*( $n$ ). Unlike the factorial problem, however, the string problem does not suggest an immediately clear way to proceed. Obviously, not any string of length  $n - 1$  will do. For example, there is no relation between writing “apple” (a string of length 5) backward and writing “pear” (a string of length 4) backward. You must choose the smaller problem carefully so that you can use its solution in the solution to the original problem.

The string of length  $n - 1$  that you choose must be a substring (part) of the original string. Suppose that you strip away one character from the original string, leaving a substring of length  $n - 1$ . For the recursive solution to be valid, the ability to write the substring backward, combined with the ability to perform some minor task, must result in the ability to write the original string backward. Compare this approach with the way you computed *factorial* recursively: The ability to compute *factorial*( $n - 1$ ), combined with the ability to multiply this value by  $n$ , resulted in the ability to compute *factorial*( $n$ ).

You need to decide which character to strip away and which minor task to perform. Consider the minor task first. Since you are writing characters, a likely candidate for the minor task is writing a single character. As for the character that you should strip away from the string, there are several possible alternatives. Two of the more intuitive alternatives are

*Strip away the last character*

or

*Strip away the first character*

Consider the first of these alternatives, stripping away the last character, as Figure 3-6 illustrates.

How can you write an  $n$ -character string backward, if you can write an  $(n - 1)$ -character string backward?

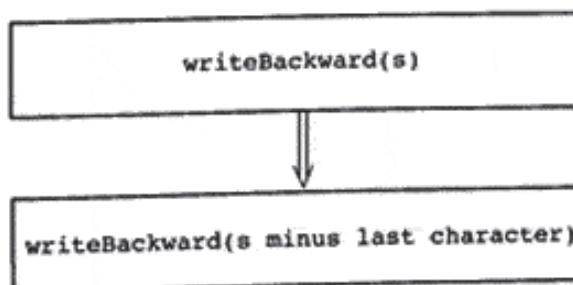


FIGURE 3-6

A recursive solution

For the solution to be valid, you must write the last character in the string first. Therefore, you must write the last character before you write the remainder of the string backward. A high-level recursive solution, given the string

```
writeBackward
writes a string
backward

writeBackward(in s:string)

 if (the string s is empty) {
 Do nothing -- this is the base case
 }
 else {
 Write the last character of s
 writeBackward(s minus its last character)
 } // end if
```

This solution to the problem is conceptual. To obtain a Java method, we must resolve a few implementation issues. Suppose that the method receives one argument: a string *s* to be written backward. Note that the string begins at position 0 and ends at position *s.length()* - 1. That is, all characters, including blanks, in that range are part of the string. The Java method *writeBackward* appears as follows:

```
public static void writeBackward(String s) {
// -----
// Writes a character string backward.
// Precondition: None.
// Postcondition: The string is written backward
// -----
 if (s.length() > 0) {
 // write the last character
 System.out.println(s.substring(s.charAt(s.length()-1)));

 // write the rest of the string backward,
 // s minus the last character
 writeBackward(s.substring(0, s.length()-1)); // Point A
 } // end if
 // size == 0 is the base case - do nothing
} // end writeBackward
```

Notice that the recursive calls to *writeBackward* use successively smaller strings. Each recursive call has the effect of stripping away the last character of the string, which ensures that the base case will be reached.

You can trace the execution of *writeBackward* by using the box trace. As was true for the method *fact*, each box contains the local environment of the recursive call—in this case, the input argument. The trace will differ somewhat from the trace of *fact* shown in Figure 3-5 because, as a *void* method, *writeBackward*

**writeBackward**  
does not return a  
computed value

does not use a `return` statement to return a computed value. Figure 3-7 traces the call to the method `writeBackward` with the string "cat".

Now consider a slightly different approach to the problem. Recall the two alternatives for the character that you could strip away from the string: the last character or the first character. The solution just given strips away the last character of the string. It will now be interesting to construct a solution based on the second alternative:

#### *Strip away the first character*

To begin, consider a simple modification of the previous pseudocode solution that replaces each occurrence of "last" with "first." Thus, the method writes the first character rather than the last and then recursively writes the remainder of the string backward.

```
writeBackward1(in s:string)

if (the string s is empty) {
 Do nothing -- this is the base case
}
else {
 Write the first character of s
 writeBackward1(s minus its first character)
} // end if
```

Does this solution do what you want it to? If you think about this method, you will realize that it writes the string in its normal left-to-right direction instead of backward. After all, the steps in the pseudocode are

*Write the first character of s  
Write the rest of s*

These steps simply write the string *s*. Naming the method `writeBackward1` does not guarantee that it will actually write the string backward—recursion really is not magic!

You can write *s* backward correctly by using the following recursive formulation:

*Write string s minus its first character backward  
Write the first character of string s*

In other words, you write the first character of *s* only *after* you have written the rest of *s* backward. This approach leads to the following pseudocode solution:

`writeBackward2(in s:string)`

**writeBackward2**  
writes a string  
backward

The following diagram shows the step-by-step design execution:

```
s = "cat"
s.length() = 3
```

Initial state

Call to `writeBackward(s)`. `s` reaches, and the recursive call is made.

```
s = "cat"
s.length() = 3
```

```
s = "ca"
s.length() = 2
```

Initial state

Call to `writeBackward(s)`. `s` reaches, and the recursive call is made.

The recursive call continues:

```
s = "cat"
s.length() = 3
```

```
s = "ca"
s.length() = 2
```

```
s = "c"
s.length() = 1
```

Initial state

Call to `writeBackward(s)`. `s` reaches, and the recursive call is made.

The recursive call continues:

```
s = "cat"
s.length() = 3
```

```
s = "ca"
s.length() = 2
```

```
s = "c"
s.length() = 1
```

```
s = ""
s.length() = 0
```

This is the base case, so this invocation completes.

Control returns to the calling box, which continues execution:

```
s = "cat"
s.length() = 3
```

```
s = "ca"
s.length() = 2
```

```
s = "c"
s.length() = 1
```

```
s = ""
s.length() = 0
```

This invocation completes. Control returns to the calling box, which continues execution:

```
s = "cat"
s.length() = 3
```

```
s = "ca"
s.length() = 2
```

```
s = "c"
s.length() = 1
```

```
s = ""
s.length() = 0
```

This invocation completes. Control returns to the calling box, which continues execution:

```
s = "cat"
s.length() = 3
```

```
s = "ca"
s.length() = 2
```

```
s = "c"
s.length() = 1
```

```
s = ""
s.length() = 0
```

This invocation completes. Control returns to the statement following the initial call.

**FIGURE 3-7**

Box trace of `writeBackward("cat")`

```

if (the string s is empty) {
 Do nothing -- this is the base case
}
else {
 writeBackward2(s minus its first character)
 Write the first character of s
} // end if

```

The translation of `writeBackward2` into Java is similar to that of the original `writeBackward` method and is left as an exercise.

It is instructive to carefully trace the actions of the two pseudocode methods `writeBackward` and `writeBackward2`. First, add statements to each method to provide output that is useful to the trace, as follows:

```

writeBackward(in s:string)

System.out.println("Enter writeBackward, string: " + s);
if (the string s is empty) {
 Do nothing -- this is the base case
}
else {
 System.out.println("About to write last character of " +
 "string: " + s);
 Write the last character of s
 writeBackward(s minus its last character) // Point A
} // end if
System.out.println("Leave writeBackward, string: " + s);

writeBackward2(in s:string)

System.out.println("Enter writeBackward2, string: " + s);
if (the string s is empty) {
 Do nothing -- this is the base case
}
else {
 writeBackward2(s minus its first character) // Point A
 System.out.println("About to write first character of" +
 "string: " + s);
 Write the first character of s
} // end if
System.out.println("Leave writeBackward2, string: " + s);

```

Output statements can help you trace the logic of a recursive method

Figures 3-8 and 3-9 show the output of the revised pseudocode methods `writeBackward` and `writeBackward2`, when initially given the string "cat".

You need to be comfortable with the differences between these two methods. The recursive calls that the two methods make generate a different sequence

The initial call is made, and the method begins execution

`s = "cat"`

Output stream:

```
Enter writeBackward, string: cat
About to write last character of string: cat
t
```

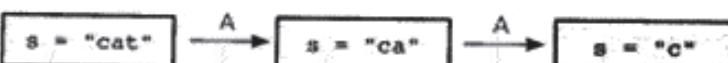
Point A is reached, and the recursive call is made. The new invocation begins execution



Output stream:

```
Enter writeBackward, string: cat
About to write last character of string: cat
t
Enter writeBackward, string: ca
About to write last character of string: ca
a
```

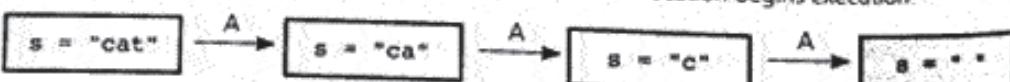
Point A is reached, and the recursive call is made. The new invocation begins execution



Output stream:

```
Enter writeBackward, string: cat
About to write last character of string: cat
t
Enter writeBackward, string: ca
About to write last character of string: ca
a
Enter writeBackward, string: c
About to write last character of string: c
c
```

Point A is reached, and the recursive call is made. The new invocation begins execution



This invocation completes execution, and a return is made.

Output stream:

```
Enter writeBackward, string: cat
About to write last character of string: cat
t
```

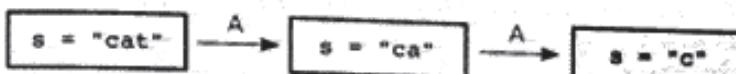
**FIGURE 3-8**

Box trace of `writeBackward("cat")` in pseudocode

```

Enter writeBackward, string: ca
About to write last character of string: ca
a
Enter writeBackward, string: c
About to write last character of string: c
c
Enter writeBackward, string:
Leave writeBackward, string:

```



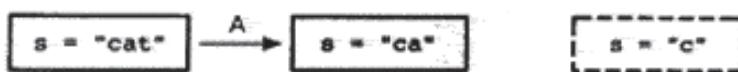
This invocation completes execution, and a return is made.

Output stream:

```

Enter writeBackward, string: cat
About to write last character of string: cat
t
Enter writeBackward, string: ca
About to write last character of string: ca
a
Enter writeBackward, string: c
About to write last character of string: c
c
Enter writeBackward, string:
Leave writeBackward, string:
Leave writeBackward, string: c

```



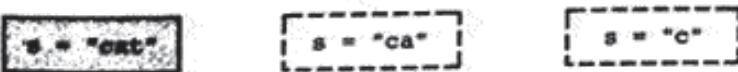
This invocation completes execution, and a return is made.

Output stream:

```

Enter writeBackward, string: cat
About to write last character of string: cat
t
Enter writeBackward, string: ca
About to write last character of string: ca
a
Enter writeBackward, string: c
About to write last character of string: c
c
Enter writeBackward, string:
Leave writeBackward, string:
Leave writeBackward, string: c
Leave writeBackward, string: ca

```



This invocation completes execution, and a return is made.

(continues)

**FIGURE 3-8**

(continued)

Output stream

```

Enter writeBackward, string: cat
About to write last character of string: cat
t
Enter writeBackward, string: ca
About to write last character of string: ca
a
Enter writeBackward, string: c
About to write last character of string: c
c
Enter writeBackward, string:
Leave writeBackward, string:
Leave writeBackward, string: c
Leave writeBackward, string: ca
Leave writeBackward, string: cat

```

The initial call is made, and the method begins execution

**s = "cat"**

Output stream

Enter writeBackward2, string: cat

Point A is reached, and the recursive call is made. The new invocation begins execution



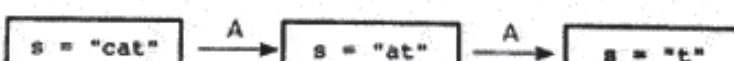
Output stream

```

Enter writeBackward2, string: cat
Enter writeBackward2, string: at

```

Point A is reached, and the recursive call is made. The new invocation begins execution:



Output stream

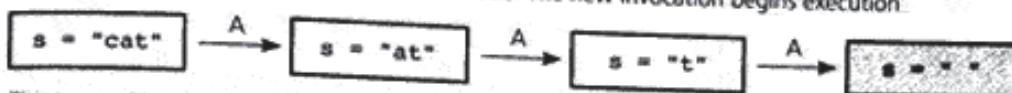
```

Enter writeBackward2, string: cat
Enter writeBackward2, string: at
Enter writeBackward2, string: t

```

**FIGURE 3-9**Box trace of `writeBackward2("cat")` in pseudocode

Point A is reached, and the recursive call is made. The new invocation begins execution:



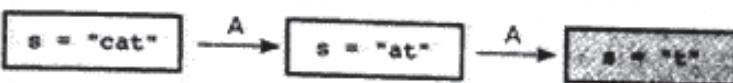
This invocation completes execution, and a return is made.

Output stream:

```

Enter writeBackward2, string: cat
Enter writeBackward2, string: at
Enter writeBackward2, string: t
Enter writeBackward2, string:
Leave writeBackward2, string:

```



This invocation completes execution, and a return is made.

Output stream:

```

Enter writeBackward2, string: cat
Enter writeBackward2, string: at
Enter writeBackward2, string: t
Enter writeBackward2, string:
Leave writeBackward2, string:
About to write first character of string: t
t
Leave writeBackward2, string: t

```



This invocation completes execution, and a return is made.

Output stream:

```

Enter writeBackward2, string: cat
Enter writeBackward2, string: at
Enter writeBackward2, string: t
Enter writeBackward2, string:
Leave writeBackward2, string:
About to write first character of string: t
t
Leave writeBackward2, string: t
About to write first character of string: at
a
Leave writeBackward2, string: at

```

(continues)

**FIGURE 3-9**

(continued)



This invocation completes execution, and a return is made.

Output stream:

```

Enter writeBackward2, string: cat
Enter writeBackward2, string: at
Enter writeBackward2, string: t
Enter writeBackward2, string:
Leave writeBackward2, string:
About to write first character of string: t
t
Leave writeBackward2, string: t
About to write first character of string: at
a
Leave writeBackward2, string: at
About to write first character of string: cat
c
Leave writeBackward2, string: cat

```

of values for the argument *s*. Despite this fact, both methods correctly write the string argument backward. They compensate for the difference in the sequence of values for *s* by writing different characters in the string at different times relative to the recursive calls. In terms of the box traces in Figures 3-8 and 3-9, *writeBackward* writes a character just before generating a new box (just before a new recursive call), whereas *writeBackward2* writes a character just after crossing off a box (just after returning from a recursive call). When these differences are put together, the result is two methods that employ different strategies to accomplish the same task.

This example also illustrates the value of the box trace, combined with well-placed *System.out.println* statements, in debugging recursive methods. The *System.out.println* statements at the beginning, interior, and end of the recursive methods report the value of the argument *s*. In general, when debugging a recursive method, you should also report both the values of local variables and the point in the method where each recursive call occurred, as in this example:

Well-placed but temporary *System.out.println* statements can help you to debug a recursive method

```

abc(...)

System.out.println("Calling abc from point A.");
abc(...) // this is point A

System.out.println("Calling abc from point B.");
abc(...) // this is point B

```

Realize that the `System.out.println` statements do not belong in the final version of the method.

Remove `Sys-  
tem.out.println`  
statements after you  
have debugged  
the method

## 3.2 Counting Things

The next three problems require you to count certain events or combinations of events or things. They are good examples of problems with more than one base case. They also provide good examples of tremendously inefficient recursive solutions. Do not let this inefficiency discourage you. Your goal right now is to understand recursion by examining simple problems. Soon you will see useful and efficient recursive solutions.

### Multiplying Rabbits (The Fibonacci Sequence)

Rabbits are very prolific breeders. If rabbits did not die, their population would quickly get out of hand. Suppose we assume the following "facts," which were obtained in a recent survey of randomly selected rabbits:

- Rabbits never die.
- A rabbit reaches sexual maturity exactly two months after birth, that is, at the beginning of its third month of life.
- Rabbits are always born in male-female pairs. At the beginning of every month, each sexually mature male-female pair gives birth to exactly one male-female pair.

Suppose you started with a single newborn male-female pair. How many pairs would there be in month 6, counting the births that took place at the beginning of month 6? Since 6 is a relatively small number, you can figure out the solution easily:

- |          |                                                                                                                                                          |
|----------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| Month 1: | 1 pair, the original rabbits.                                                                                                                            |
| Month 2: | 1 pair still, since it is not yet sexually mature.                                                                                                       |
| Month 3: | 2 pairs; the original pair has reached sexual maturity and has given birth to a second pair.                                                             |
| Month 4: | 3 pairs; the original pair has given birth again, but the pair born at the beginning of month 3 is not yet sexually mature.                              |
| Month 5: | 5 pairs; all rabbits alive in month 3 (2 pairs) are now sexually mature. Add their offspring to those pairs alive in month 4 (3 pairs) to yield 5 pairs. |
| Month 6: | 8 pairs; 3 newborn pairs from the pairs alive in month 4 plus 5 pairs alive in month 5.                                                                  |

You can now construct a recursive solution for computing `rabbit(n)`, the number of pairs alive in month  $n$ . You must determine how you can use `rab-`

$\text{bit}(n - 1)$  to compute  $\text{rabbit}(n)$ . Observe that  $\text{rabbit}(n)$  is the sum of the number of pairs alive just prior to the start of month  $n$  and the number of pairs born at the start of month  $n$ . Just prior to the start of month  $n$ , there are  $\text{rabbit}(n - 1)$  pairs of rabbits. Not all of these rabbits are sexually mature at the start of month  $n$ . Only those who were alive in month  $n - 2$  are ready to reproduce at the start of month  $n$ . That is, the number of pairs born at the start of month  $n$  is  $\text{rabbit}(n - 2)$ . Therefore, you have the recurrence relation

$$\text{rabbit}(n) = \text{rabbit}(n - 1) + \text{rabbit}(n - 2)$$

Figure 3-10 illustrates this relationship.

This recurrence relation introduces a new point. In some cases, you solve a problem by solving more than one smaller problem of the same type. This change does not add much conceptual difficulty, but you must be very careful when selecting the base case. The temptation is simply to say that  $\text{rabbit}(1)$  should be the base case because its value is 1 according to the problem's statement. But what about  $\text{rabbit}(2)$ ? Applying the recursive definition to  $\text{rabbit}(2)$  would yield

$$\text{rabbit}(2) = \text{rabbit}(1) + \text{rabbit}(0)$$

Thus, the recursive definition would need to specify the number of pairs alive in month 0—an undefined quantity.

One possible solution is to define  $\text{rabbit}(0)$  to be 0, but this approach seems artificial. A slightly more attractive alternative is to treat  $\text{rabbit}(2)$  itself as a special case with the value of 1. Thus, the recursive definition has two base cases,  $\text{rabbit}(2)$  and  $\text{rabbit}(1)$ . The recursive definition becomes

$$\text{rabbit}(n) = \begin{cases} 1 & \text{if } n \text{ is 1 or 2} \\ \text{rabbit}(n - 1) + \text{rabbit}(n - 2) & \text{if } n > 2 \end{cases}$$

Incidentally, the series of numbers  $\text{rabbit}(1)$ ,  $\text{rabbit}(2)$ ,  $\text{rabbit}(3)$ , and so on is known as the **Fibonacci sequence**, which models many naturally occurring phenomena.

A Java method to compute  $\text{rabbit}(n)$  is easy to write from the previous definition:

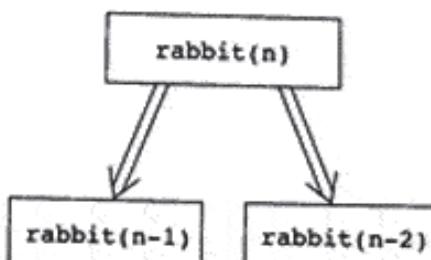


FIGURE 3-10

Recursive solution to the rabbit problem

Two base cases are necessary because there are two smaller problems of the same type

```

public static int rabbit(int n) {
// -----
// Computes a term in the Fibonacci sequence.
// Precondition: n is a positive integer.
// Postcondition: Returns the nth Fibonacci number.
// -----
 if (n <= 2) {
 return 1;
 }
 else { // n > 2, so n-1 > 0 and n-2 > 0
 return rabbit(n-1) + rabbit(n-2);
 } // end if
} // end rabbit

```

*rabbit* computes the Fibonacci sequence but does so inefficiently

Should you actually use this method? Figure 3-11 illustrates the recursive calls that *rabbit(7)* generates. Think about the number of recursive calls that *rabbit(10)* generates. At best, the method *rabbit* is inefficient. Thus, its use is not feasible for large values of *n*. This problem is discussed in more detail at the end of this chapter, at which time you will see some techniques for generating a more efficient solution from this same recursive relationship.

## Organizing a Parade

You have been asked to organize the Fourth of July parade, which will consist of bands and floats in a single line. Last year, adjacent bands tried to outplay each other. To avoid this problem, the sponsors have asked you never to place one band immediately after another. In how many ways can you organize a parade of length *n*?

Assume that you have at least *n* marching bands and *n* floats from which to choose. When counting the number of ways to organize the parade, assume that the sequences *band-float* and *float-band*, for example, are different entities and count as two ways.

The parade can end with either a float or a band. The number of ways to organize the parade is simply the sum of the number of parades of each type. That is, let

*P(n)* be the number of ways to organize a parade of length *n*

*R(n)* be the number of parades of length *n* that end with a float

*B(n)* be the number of parades of length *n* that end with a band

Then

$$P(n) = R(n) + B(n)$$

First, consider *R(n)*. You will have a parade of length *n* that ends with a float simply by placing a float at the end of *any* acceptable parade of length *n* - 1.

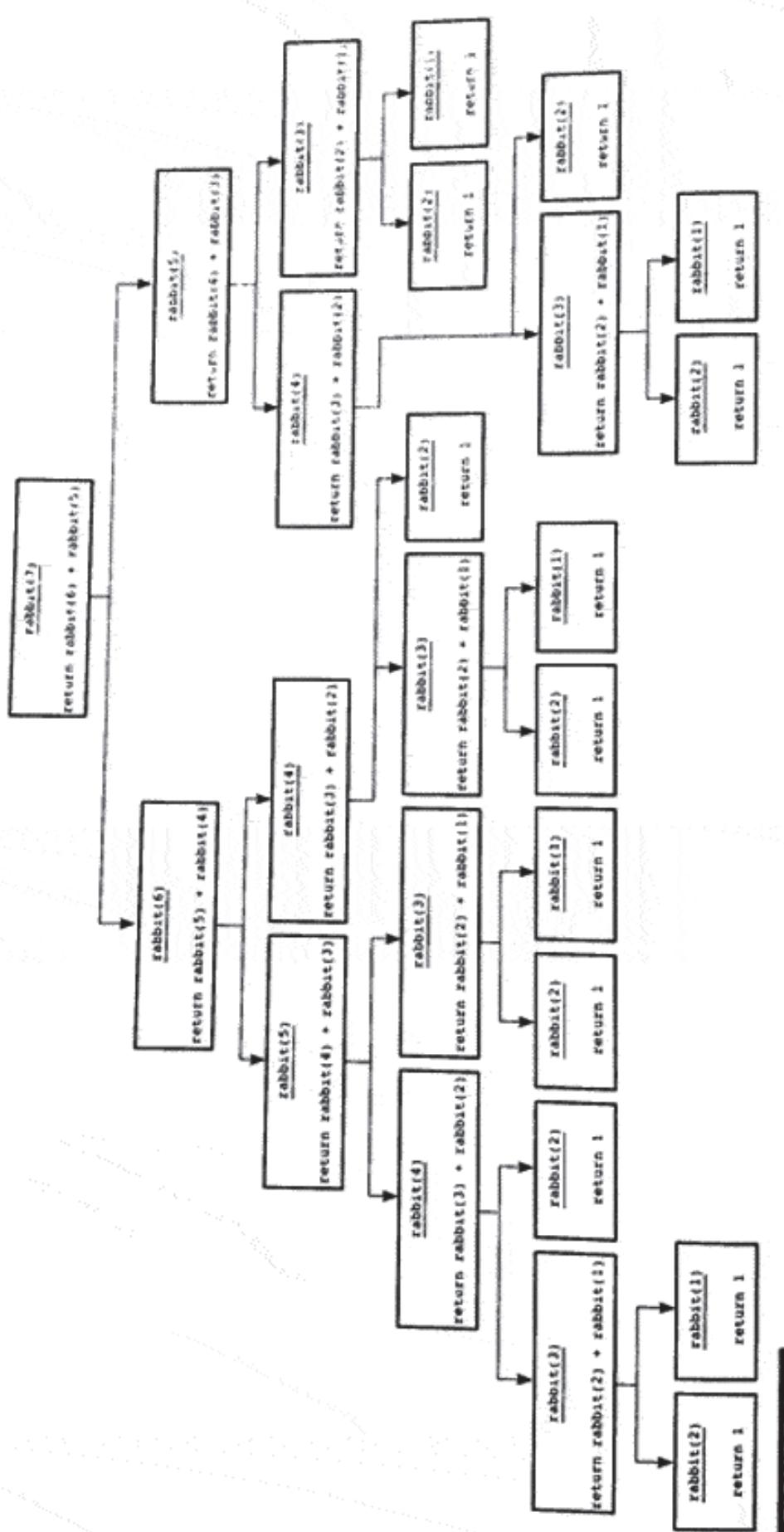


FIGURE 3-11

The recursive calls that rabbit(7) generates

Hence, the number of acceptable parades of length  $n$  that end with a float is precisely equal to the total number of acceptable parades of length  $n - 1$ ; that is,

$$F(n) = P(n - 1)$$

Next, consider  $B(n)$ . The only way a parade can end with a band is if the unit just before the end is a float. (If it is a band, you will have two adjacent bands.) Thus, the only way to organize an acceptable parade of length  $n$  that ends with a band is first to organize a parade of length  $n - 1$  that ends with a float and then add a band to the end. Therefore, the number of acceptable parades of length  $n$  that end with a band is precisely equal to the number of acceptable parades of length  $n - 1$  that end with a float:

$$B(n) = F(n - 1)$$

The number of acceptable parades of length  $n$  that end with a float

The number of acceptable parades of length  $n$  that end with a band

You use the earlier fact that  $F(n) = P(n - 1)$  to obtain

$$B(n) = P(n - 2)$$

Thus, you have solved  $F(n)$  and  $B(n)$  in terms of the smaller problems  $P(n - 1)$  and  $P(n - 2)$ , respectively. You then use

$$P(n) = F(n) + B(n)$$

to obtain

$$P(n) = P(n - 1) + P(n - 2)$$

The form of this recurrence relation is identical to the solution for the multiplying rabbits problem.

As you saw in the rabbit problem, two base cases are necessary because the recurrence relation defines a problem in terms of two smaller problems. As you did for the rabbit problem, you can choose  $n = 1$  and  $n = 2$  for the base cases. Although both problems use the same  $n$ 's for their base cases, there is no reason to expect that they use the same values for these base cases. That is, there is no reason to expect that  $\text{rabbit}(1)$  is equal to  $P(1)$  and that  $\text{rabbit}(2)$  is equal to  $P(2)$ .

A little thought reveals that for the parade problem,

$$P(1) = 2 \text{ (The parades of length 1 are float and band.)}$$

$$P(2) = 3 \text{ (The parades of length 2 are float-float, band-float, and float-band.)}$$

The number of acceptable parades of length  $n$

Two base cases are necessary because there are two smaller problems of the same type

In summary, the solution to this problem is

$$P(1) = 2$$

$$P(2) = 3$$

$$P(n) = P(n - 1) + P(n - 2) \quad \text{for } n > 2$$

A recursive solution

This example demonstrates the following points about recursion:

- Sometimes you can solve a problem by breaking it up into cases—for example, parades that end with a float and parades that end with a band.
- The values that you use for the base cases are extremely important. Although the recurrence relations for  $P$  and  $rabbit$  are the same, the different values for their base cases ( $n = 1$  or  $2$ ) cause different values for larger values of  $n$ . For example,  $rabbit(20) = 6,765$ , while  $P(20) = 17,711$ . The larger the value of  $n$ , the larger the discrepancy. You should think about why this is so.

### Mr. Spock's Dilemma (Choosing $k$ out of $n$ Things)

The five-year mission of the *U.S.S. Enterprise* is to explore new worlds. The five years are almost up, but the *Enterprise* has just entered an unexplored solar system that contains  $n$  planets. Unfortunately, time will allow for visits to only  $k$  planets. Mr. Spock begins to ponder how many different choices are possible for exploring  $k$  planets out of the  $n$  planets in the solar system. Because time is short, he does not care about the order in which he visits the same  $k$  planets.

Mr. Spock is especially fascinated by one particular planet, Planet  $X$ . He begins to think—in terms of Planet  $X$ —about how to pick  $k$  planets out of the  $n$ . “There are two possibilities: Either we visit Planet  $X$ , or we do not visit Planet  $X$ . If we do visit Planet  $X$ , I will have to choose  $k - 1$  other planets to visit from the  $n - 1$  remaining planets. On the other hand, if we do not visit Planet  $X$ , I will have to choose  $k$  planets to visit from the remaining  $n - 1$  planets.”

Mr. Spock is on his way to a recursive method of counting how many groups of  $k$  planets he can possibly choose out of  $n$ . Let  $c(n, k)$  be the number of groups of  $k$  planets chosen from  $n$ . Then, in terms of Planet  $X$ , Mr. Spock deduces that

$$c(n, k) = \text{(the number of groups of } k \text{ planets that include Planet } X\text{)}$$

+

$$\text{(the number of groups of } k \text{ planets that do not include Planet } X\text{)}$$

But Mr. Spock has already reasoned that the number of groups that include Planet  $X$  is  $c(n - 1, k - 1)$ , and the number of groups that do not include Planet  $X$  is  $c(n - 1, k)$ . Mr. Spock has figured out a way to solve his counting problem in terms of two smaller counting problems of the same type:

$$c(n, k) = c(n - 1, k - 1) + c(n - 1, k)$$

Mr. Spock now has to worry about the base case(s). He also needs to demonstrate that each of the two smaller problems eventually reaches a base case. First, what selection problem does he immediately know the answer to? If the *Enterprise* had time to visit all the planets (that is, if  $k = n$ ), no decision

The number of ways to choose  $k$  out of  $n$  things is the sum of the number of ways to choose  $k - 1$  out of  $n - 1$  things and the number of ways to choose  $k$  out of  $n - 1$  things

would be necessary; there is only one group of all the planets. Thus, the first base case is

$$c(k, k) = 1$$

If  $k < n$ , it is easy to see that the second term in the recursive definition,  $c(n - 1, k)$ , is "closer" to the base case  $c(k, k)$  than is  $c(n, k)$ . However, the first term,  $c(n - 1, k - 1)$ , is not closer to  $c(k, k)$  than is  $c(n, k)$ —they are the same "distance" apart. *When you solve a problem by solving two (or more) smaller problems, each of the smaller problems must be closer to a base case than the original problem.*

Base case: There is one group of everything

Mr. Spock realizes that the first term does, in fact, approach another trivial selection problem. This problem is the counterpart of his first base case,  $c(k, k)$ . Just as there is only one group of all the planets ( $k = n$ ), there is also only one group of zero planets ( $k = 0$ ). When there is no time to visit any of the planets, the *Enterprise* must head home without any exploration. Thus, the second base case is

$$c(n, 0) = 1$$

Base case: There is one group of nothing

This base case does indeed have the property that  $c(n - 1, k - 1)$  is closer to it than is  $c(n, k)$ . (Alternatively, you could define the second base case to be  $c(n, 1) = n$ .)

Mr. Spock adds one final part to his solution:

$$c(n, k) = 0 \quad \text{if } k > n$$

Although  $k$  could not be greater than  $n$  in the context of this problem, the addition of this case makes the recursive solution more generally applicable.

To summarize, the following recursive solution solves the problem of choosing  $k$  out of  $n$  things:

$$c(n, k) = \begin{cases} 1 & \text{if } k = 0 \\ 1 & \text{if } k = n \\ 0 & \text{if } k > n \\ c(n - 1, k - 1) + c(n - 1, k) & \text{if } 0 < k < n \end{cases}$$

The number of groups of  $k$  things recursively chosen out of  $n$  things

You can easily derive the following method from this recursive definition:

```
public static int c(int n, int k) {
// -----
// Computes the number of groups of k out of n things.
// Precondition: n and k are nonnegative integers.
// Postcondition: Returns c(n, k).
// -----
 if ((k == 0) || (k == n)) {
 return 1;
 }
 else {
 return c(n - 1, k - 1) + c(n - 1, k);
 }
}
```

```

else if (k > n) {
 return 0;
}
else {
 return c(n-1, k-1) + c(n-1, k);
} // end if
} // end c

```

Like the rabbit method, this method is inefficient and not practical to use. Figure 3-12 shows the number of recursive calls that the computation of  $c(4, 2)$  requires.

### 3.3 Searching an Array

Searching is an important task that occurs frequently. This chapter began with an intuitive approach to a binary search algorithm. This section develops the binary search and examines other searching problems that have recursive solutions. The goal is to develop further your notion of recursion.

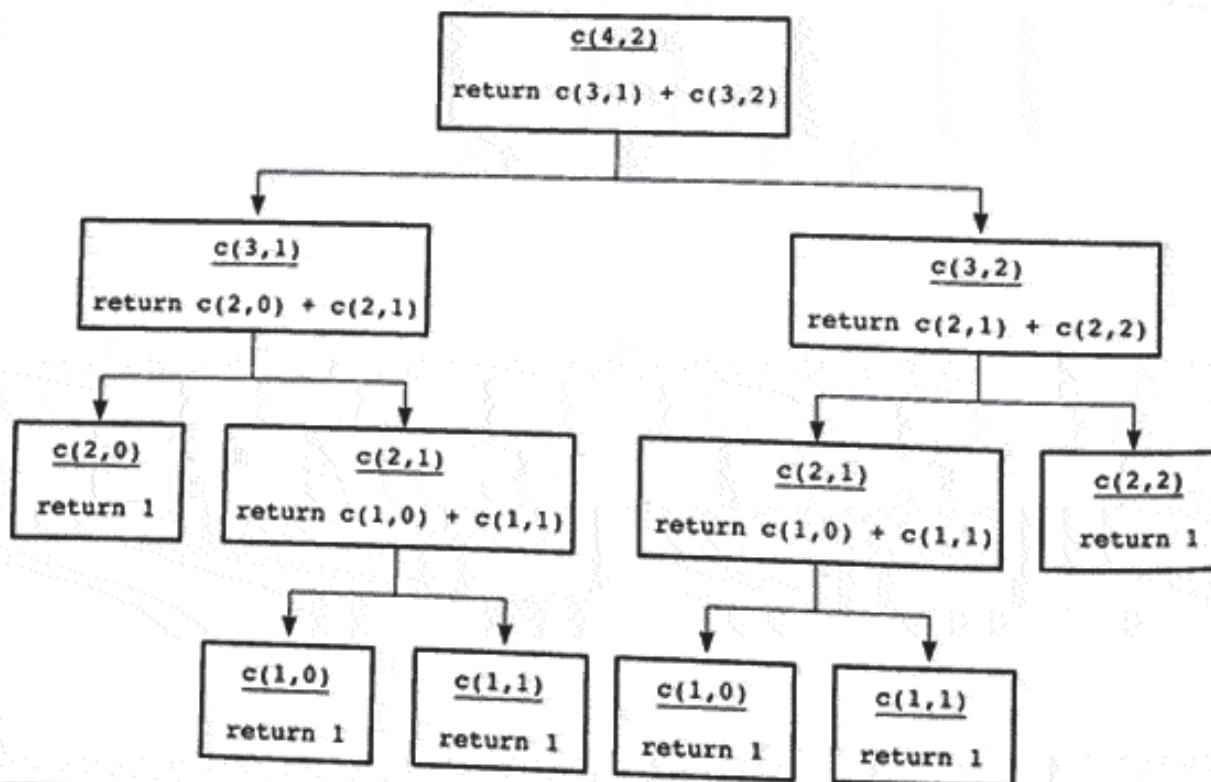


FIGURE 3-12

The recursive calls that  $c(4, 2)$  generates

## Finding the Largest Item in an Array

Suppose that you have an array *anArray* of integers and you want to find the largest one. You could construct an iterative solution without too much difficulty, but instead consider a recursive formulation:

```
if (anArray has only one item) {
 maxArray(anArray) is the item in anArray
}
else if (anArray has more than one item) {
 maxArray(anArray) is the maximum of
 maxArray(left half of anArray) and
 maxArray(right half of anArray)
} // end if
```

Notice that this strategy fits the divide-and-conquer model that the binary search algorithm used at the beginning of this chapter. That is, the algorithm proceeds by dividing the problem and conquering the subproblems, as Figure 3-13 illustrates. However, there is a difference between this algorithm and the binary search algorithm. While the binary search algorithm conquers only one of its subproblems at each step, *maxArray* conquers both. In addition, after *maxArray* conquers the subproblems, it must reconcile the two solutions—that is, it must find the maximum of the two maximums. Figure 3-14 illustrates the computations that are necessary to find the largest integer in the array that contains 1, 6, 8, and 3 (denoted here by <1, 6, 8, 3>).

You should develop a recursive solution based on this strategy. In so doing, you may stumble on several subtle programming issues. The binary search problem that follows raises virtually all of these issues, but this is a good opportunity for you to get some practice implementing a recursive solution.

**maxArray** conquers both of its subproblems at each step

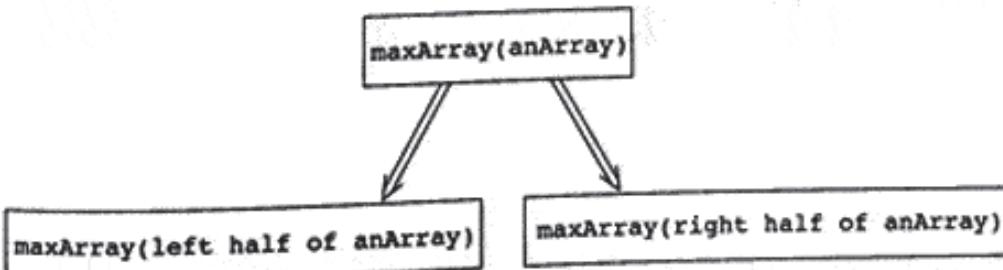
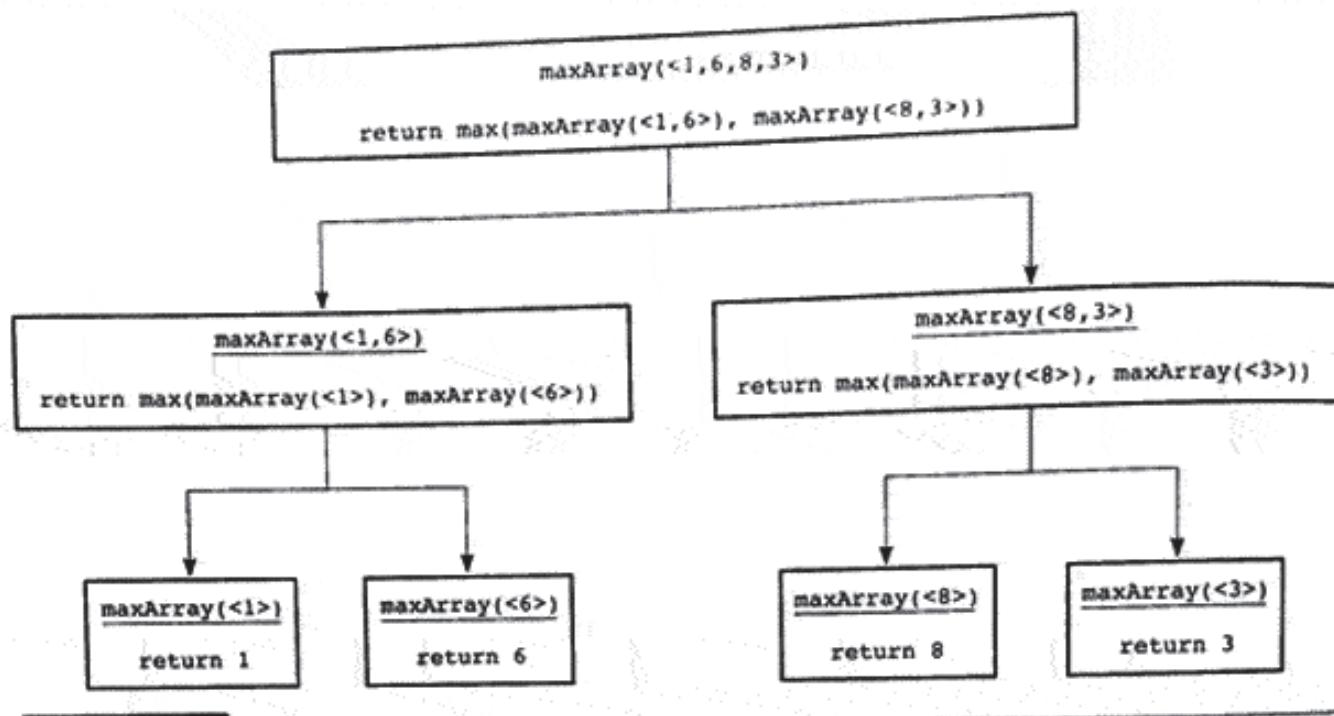


FIGURE 3-13

Recursive solution to the largest-item problem

**FIGURE 3-14**

The recursive calls that `maxArray(<1,6,8,3>)` generates

## Binary Search

The beginning of this chapter presented—at a high level—a recursive binary search algorithm for finding a word in a dictionary. We now develop this algorithm fully and illustrate some important programming issues.

Recall the earlier solution to the dictionary problem:

```

search(in theDictionary:Dictionary, in aWord: string)
 if (theDictionary is one page in size) {
 Scan the page for aWord
 }
 else {
 Scan theDictionary to a point near the middle.
 Determine which half of theDictionary contains
 theWord.
 If Word is in first half, then
 search(first half of theDictionary, aWord)
 If Word is in second half, then
 search(second half of theDictionary, aWord)
 }
}

```

Now alter the problem slightly by searching an array *anArray* of integers for a given value. The array, like the dictionary, must be sorted, or else a binary search is not applicable. Hence, assume that

*anArray[0] ≤ anArray[1] ≤ anArray[2] ≤ . . . ≤ anArray[size-1]*

where *size* is the size of the array. A high-level binary search for the array problem is

An array must be sorted before you can apply a binary search to it.

*binarySearch(in anArray:ArrayType, in value:Item Type)*

```
if (anArray is of size 1) {
 Determine if anArray's item is equal to value
}
else {
 Find the midpoint of anArray
 Determine which half of anArray contains value
 if (value is in the first half of anArray) {
 binarySearch(first half of anArray, value)
 }
 else {
 binarySearch(second half of anArray, value)
 } // end if
} // end if
```

Although the solution is conceptually sound, you must consider several details before you can implement the algorithm:

1. How will you pass "half of *anArray*" to the recursive calls to *binarySearch*? You can pass the entire array at each call but have *binarySearch* search only *anArray[first..last]*,<sup>1</sup> that is, the portion *anArray[first]* through *anArray[last]*. Thus, you would also pass the integers *first* and *last* to *binarySearch*:

*binarySearch(anArray, first, last, value)*

With this convention, the new midpoint is given by

*mid = (first + last)/2*

Then *binarySearch(first half of anArray, value)* becomes

*binarySearch(anArray, first, mid-1, value)*

The array halves are *anArray[first..mid-1]* and *anArray[mid+1..last]*; neither half contains *anArray[mid]*.

1. You will see this notation in the rest of the book to represent a portion of an array.

and `binarySearch(second half of anArray, value)` becomes  
`binarySearch(anArray, mid+1, last, value)`

2. How do you determine which half of the array contains `value`? One possible implementation of

`if (value is in the first half of anArray)`

is

`if (value < anArray[mid])`

However, there is no test for equality between `value` and `anArray[mid]`. This omission can cause the algorithm to miss `value`. After the previous halving algorithm splits `anArray` into halves, `anArray[mid]` is not in either half of the array. (In this case, two halves do not make a whole!) Therefore, you must determine whether `anArray[mid]` is the value you seek *now* because later it will not be in the remaining half of the array. The interaction between the halving criterion and the termination condition (the base case) is subtle and is often a source of error. We need to rethink the base case.

3. What should the base case(s) be? As it is written, `binarySearch` terminates only when an array of size 1 occurs; this is the only base case. By changing the halving process so that `anArray[mid]` remains in one of the halves, it is possible to implement the binary search correctly so that it has only this single base case. However, it can be clearer to have two distinct base cases as follows:

Two base cases

- `first > last`. You will reach this base case when `value` is not in the original array.
- `value == anArray[mid]`. You will reach this base case when `value` is in the original array.

These base cases are a bit different from any you have encountered previously. In a sense, the algorithm determines the answer to the problem from the base case it reaches. Many search problems have this flavor.

4. How will `binarySearch` indicate the result of the search? If `binarySearch` successfully locates `value` in the array, it could return the index of the array item that is equal to `value`. Since this index would never be negative, `binarySearch` could return a negative value if it does not find `value` in the array.

The Java method `binarySearch` that follows implements these ideas. The two recursive calls to `binarySearch` are labeled as `X` and `Y` for use in a later box trace of this method.

Determine whether  
`anArray[mid]` is  
the value you seek

```

public static int binarySearch(int anArray[], int first,
 int last, int value) {
 // Searches the array items anArray[first] through
 // anArray[last] for value by using a binary search.
 // Precondition: 0 <= first, last <= SIZE-1, where
 // SIZE is the maximum size of the array, and
 // anArray[first] <= anArray[first+1] <= ... <=
 // anArray[last].
 // Postcondition: If value is in the array, the method
 // returns the index of the array item that equals value;
 // otherwise the method returns -1.
 int index;
 if (first > last) {
 index = -1; // value not in original array
 }
 else {
 // Invariant: If value is in anArray,
 // anArray[first] <= value <= anArray[last]
 int mid = (first + last)/2;
 if (value == anArray[mid]) {
 index = mid; // value found at anArray[mid]
 }
 else if (value < anArray[mid]) {
 // point X
 index = binarySearch(anArray, first, mid-1, value);
 }
 else {
 // point Y
 index = binarySearch(anArray, mid+1, last, value);
 } // end if
 } // end if

 return index;
} // end binarySearch

```

Notice that *binarySearch* has the following invariant: If *value* occurs in the array, then *anArray[first] ≤ value ≤ anArray[last]*.

Figure 3-15 shows box traces of *binarySearch* when it searches the array containing 1, 5, 9, 12, 15, 21, 29, and 31. Notice how the labels *X* and *Y* of the two recursive calls to *binarySearch* appear in the diagram. Exercise 16 at the end of this chapter asks you to perform other box traces with this method.

There is another implementation issue—one that deals specifically with Java—to consider. Recall that an array is an object, and when the method *binarySearch* is called, only the reference to the array is copied to the method, not the entire array contents. This aspect of Java is particularly useful in a recursive method such as *binarySearch*. If the array *anArray* is large,

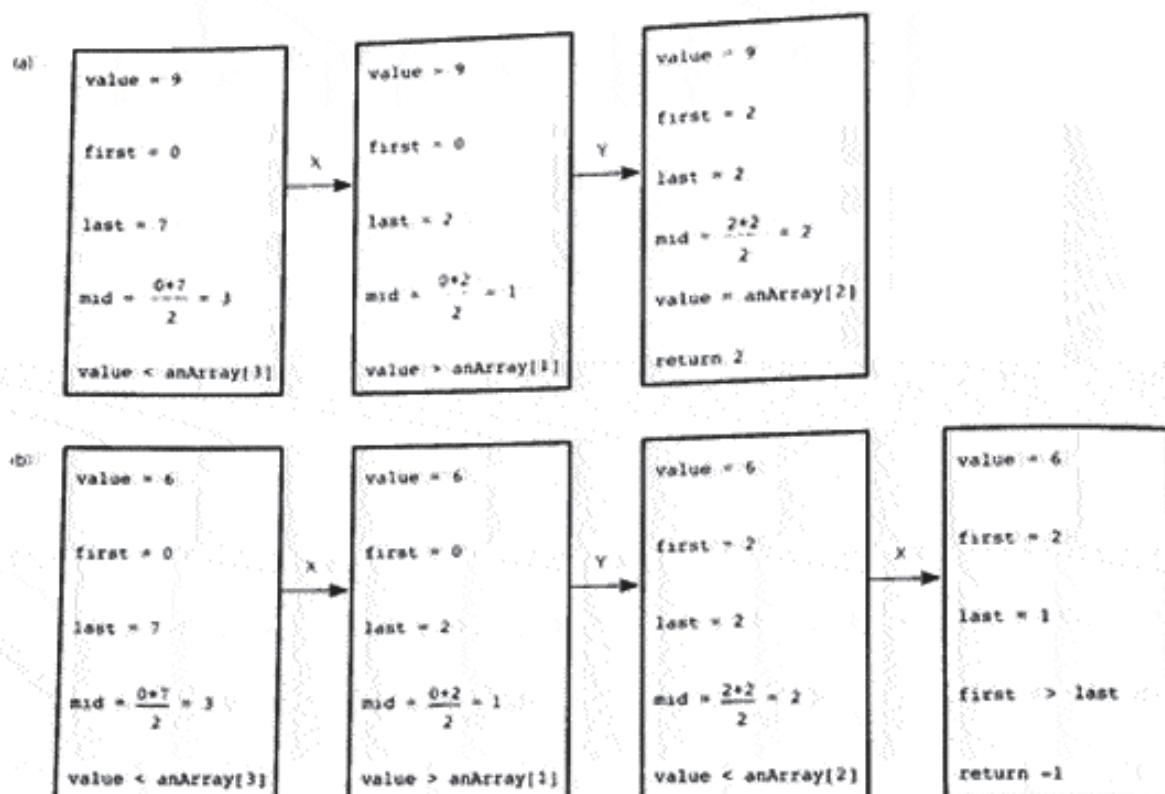


FIGURE 3-15

Box traces of `binarySearch` with `anArray = <1, 5, 9, 12, 15, 21, 29, 31>`:  
 (a) a successful search for 9; (b) an unsuccessful search for 6

many recursive calls to `binarySearch` may be necessary. If each call copied `anArray`, much memory and time would be wasted.

A box trace of a recursive method that has an array argument requires a new consideration. Because only the reference to `anArray` is passed and it is not a local variable, the contents of the array are not a part of the method's local environment and should not appear within each box. Therefore, as Figure 3-16 shows, you represent `anArray` outside the boxes, and all references to `anArray` affect this single representation.

### Finding the $k^{\text{th}}$ Smallest Item in an Array

Our discussion of searching concludes with a more difficult problem. Although you could skip this example now, Chapter 10 uses aspects of it in a sorting algorithm.

The previous two examples presented recursive methods for finding the largest item in an arbitrary array and for finding an arbitrary item in a sorted array. This example describes a recursive solution for finding the  $k^{\text{th}}$  smallest item in an arbitrary array `anArray`. Would you ever be interested in such an

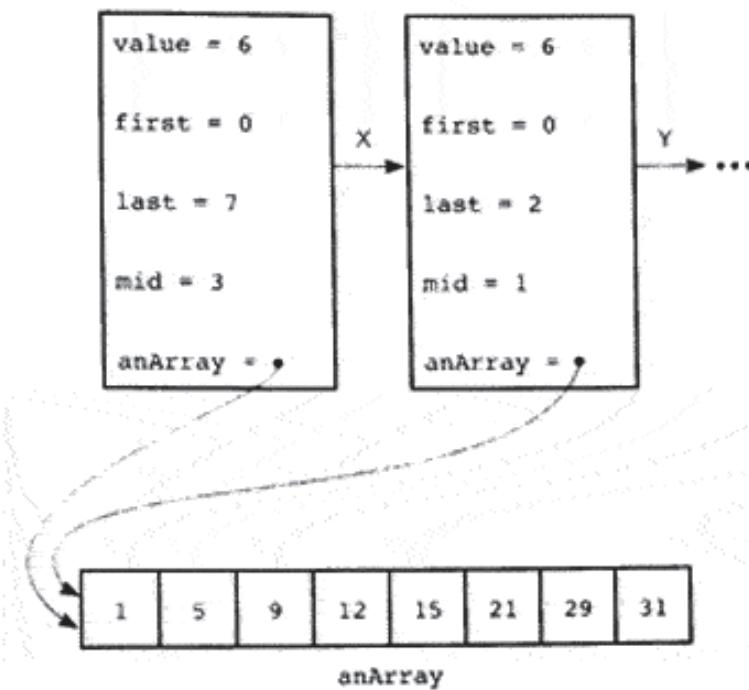


FIGURE 3-16

Box trace with a reference to an array

item? Statisticians often want the median value in a collection of data. The median value in an ordered collection of data occurs in the middle of the collection. In an unordered collection of data, there are about the same number of values smaller than the median value as there are larger values. Thus, if you have 49 items, the 25<sup>th</sup> smallest item is the median value.

Obviously, you could solve this problem by sorting the array. Then the  $k^{\text{th}}$  smallest item would be  $\text{anArray}[k-1]$ . Although this approach is a legitimate solution, it does more than the problem requires; a more efficient solution is possible. The solution outlined here finds the  $k^{\text{th}}$  smallest item without completely sorting the array.

By now, you know that you solve a problem recursively by writing its solution in terms of one or more smaller problems of the same type in such a way that this notion of *smaller* ensures that you will always reach a base case. For all of the earlier recursive solutions, the reduction in problem size between recursive calls is *predictable*. For example, the factorial method always decreases the problem size by 1; the binary search always halves the problem size. In addition, the base cases for all the previous problems except the binary search have a static, predefined size. Thus, by knowing only the size of the original problem, you can determine the number of recursive calls that are necessary before you reach the base case.

The solution that you are about to see for finding the  $k^{\text{th}}$  smallest item departs from these traditions. Although you solve the problem in terms of a smaller problem, just how much smaller this problem is depends on the items in the array and cannot be predicted in advance. Also, the size of the base

For all previous examples, you know the amount of reduction made in the problem size by each recursive call

You cannot predict in advance the size of either the smaller problems or the base case in the recursive solution to the  $k^{\text{th}}$  smallest item problem

case depends on the items in the array, as it did for the binary search. (Recall that you reach a base case for a binary search when the middle item is the one sought.)

This “unpredictable” type of solution is caused by the nature of the problem: The relationship between the rankings of the items in any predetermined parts of the array and the ranking of the items in the entire array is not strong enough to determine the  $k^{\text{th}}$  smallest item. For example, suppose that *anArray* contains the items shown in Figure 3-17. Notice that 6, which is in *anArray[3]*, is the third smallest item in the first half of *anArray* and that 8, which is in *anArray[4]*, is the third smallest item in the second half of *anArray*. Can you conclude from these observations anything about the location of the third smallest item in all of *anArray*? The answer is no; these facts about parts of the array do not allow you to draw any useful conclusions about the entire array. You should experiment with other fixed splitting schemes as well.

The recursive solution proceeds by

1. Selecting a pivot item in the array
2. Cleverly arranging, or partitioning, the items in the array about this pivot item
3. Recursively applying the strategy to *one* of the partitions

Consider the details of the recursive solution: You want to find the  $k^{\text{th}}$  smallest item in the array segment *anArray[first..last]*. Let the pivot *p* be any item of the array segment. (For now, ignore how to choose *p*.) You can partition the items of *anArray[first..last]* into three regions:  $S_1$ , which contains the items less than *p*; the pivot *p* itself; and  $S_2$ , which contains the items greater than or equal to *p*. This partition implies that all the items in  $S_1$  are smaller than all the items in  $S_2$ . Figure 3-18 illustrates this partition.

All items in *anArray[first..pivotIndex-1]*, in terms of array subscripts, are less than *p*, and all items in *anArray[pivotIndex+1..last]* are greater than or equal to *p*. Notice that the sizes of the regions  $S_1$  and  $S_2$  depend on both *p* and the other items of *anArray[first..last]*.

This partition induces three “smaller problems,” such that the solution to one of the problems will solve the original problem:

1. If  $S_1$  contains  $k$  or more items,  $S_1$  contains the  $k$  smallest items of the array segment *anArray[first..last]*. In this case, the  $k^{\text{th}}$  smallest item must

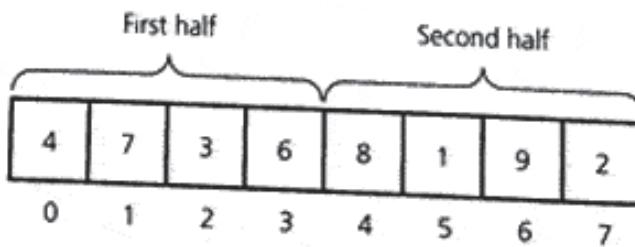


FIGURE 3-17

A sample array

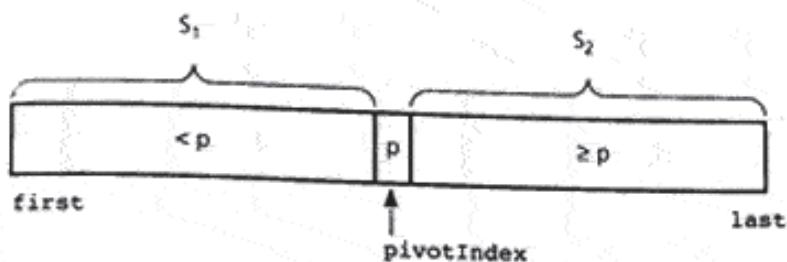


FIGURE 3-18

- Partition about a pivot

be in  $S_1$ . Since  $S_1$  is the array segment `anArray[first..pivotIndex-1]`, this case occurs if  $k < \text{pivotIndex} - \text{first} + 1$ .

2. If  $S_1$  contains  $k - 1$  items, the  $k^{\text{th}}$  smallest item must be the pivot `p`. This is the base case; it occurs if  $k = \text{pivotIndex} - \text{first} + 1$ .
3. If  $S_1$  contains fewer than  $k - 1$  items, the  $k^{\text{th}}$  smallest item in `anArray[first..last]` must be in  $S_2$ . Because  $S_1$  contains  $\text{pivotIndex} - \text{first}$  items, the  $k^{\text{th}}$  smallest item in `anArray[first..last]` is the  $(k - \text{pivotIndex} + \text{first} + 1)^{\text{th}}$  smallest item in  $S_2$ . This case occurs if  $k > \text{pivotIndex} - \text{first} + 1$ .

A recursive definition can summarize this discussion. Let

`kSmall(k, anArray, first, last)` =  

$$\begin{aligned} &\text{ } \\ &\text{ } \end{aligned}$$
  

$$k^{\text{th}} \text{ smallest item in } \text{anArray}[first..last]$$

After you select the pivot item `p` and partition `anArray[first..last]` into  $S_1$  and  $S_2$ , you have that

`kSmall(k, anArray, first, last)`  

$$= \begin{cases} \text{kSmall}(k, \text{anArray}, \text{first}, \text{pivotIndex}-1) & \text{if } k < \text{pivotIndex} - \text{first} + 1 \\ p & \text{if } k = \text{pivotIndex} - \text{first} + 1 \\ \text{kSmall}(k-(\text{pivotIndex}-\text{first}+1), \text{anArray}, \text{pivotIndex}+1, \text{last}) & \text{if } k > \text{pivotIndex} - \text{first} + 1 \end{cases}$$

The  $k^{\text{th}}$  smallest item in `anArray[first..last]`

There is always a pivot, and since it is not part of either  $S_1$  or  $S_2$ , the size of the array segment to be searched decreases by at least 1 at each step. Thus, you will eventually reach the base case: The desired item is a pivot. A high-level pseudocode solution is as follows:

```
kSmall(in k:integer, in anArray:ArrayType, in first:integer,
 in last:integer)
// Returns the k^{th} smallest value in anArray[first..last].
```

Choose a pivot item `p` from `anArray[first..last]`  
 Partition the items of `anArray[first..last]` about `p`

```
if (k < pivotIndex - first + 1) {
 return kSmall(k, anArray, first, pivotIndex-1)
}
else if (k == pivotIndex - first + 1) {
 return p
}
else {
 return kSmall(k-(pivotIndex-first+1), anArray,
 pivotIndex+1, last)
} // end if
```

This pseudocode is not far from a Java method. The only questions that remain are how to choose the pivot item  $p$  and how to partition the array about the chosen  $p$ . The choice of  $p$  is arbitrary. Any  $p$  in the array will work, although the sequence of choices will affect how soon you reach the base case. Chapter 10 gives an algorithm for partitioning the items about  $p$ . There you will see how to turn the method *kSmall* into a sorting algorithm.

### 3.4 Organizing Data

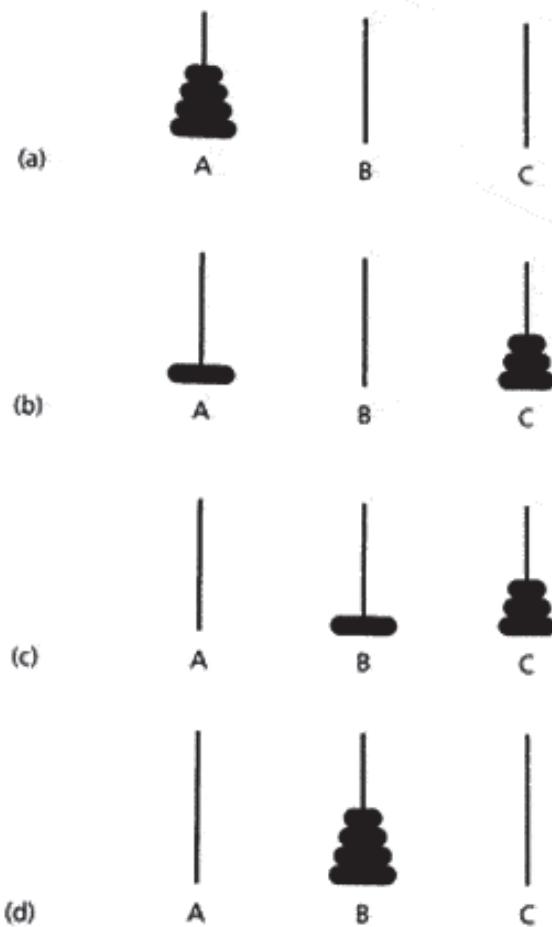
Given some data organized in one way, you might need to organize the data in another way. Thus, you will actually change some aspect of the data and not, for example, simply search it. The problem in this section is called the Towers of Hanoi. Although this classic problem probably has no direct real-world application, we consider it because its solution so well illustrates the use of recursion.

#### The Towers of Hanoi

Many, many years ago, in a distant part of the Orient—in the Vietnamese city of Hanoi—the Emperor's wiseperson passed on to join his ancestors. The Emperor needed a replacement wiseperson. Being a rather wise person himself, the Emperor devised a puzzle, declaring that its solver could have the job of wiseperson.

The Emperor's puzzle consisted of  $n$  disks (he didn't say exactly how many) and three poles:  $A$  (the source),  $B$  (the destination), and  $C$  (the spare). The disks were of different sizes and had holes in the middle so that they could fit on the poles. Because of their great weight, the disks could be placed only on top of disks larger than themselves. Initially, all the disks were on pole  $A$ , as shown in Figure 3-19a. The puzzle was to move the disks, one by one, from pole  $A$  to pole  $B$ . A person could also use pole  $C$  in the course of the transfer, but again a disk could be placed only on top of a disk larger than itself.

As the position of wiseperson was generally known to be a soft job, there were many applicants. Scholars and peasants alike brought the Emperor their solutions. Many solutions were thousands of steps long, and many contained *goto*'s. "I can't understand these solutions," bellowed the Emperor. "There must be an easy way to solve this puzzle."

**FIGURE 3-19**

(a) The initial state; (b) move  $n - 1$  disks from A to C; (c) move one disk from A to B. (d) move  $n - 1$  disks from C to B

And indeed there was. A great Buddhist monk came out of the mountains to see the Emperor. "My son," he said, "the puzzle is so easy, it almost solves itself." The Emperor's security chief wanted to throw this strange person out, but the Emperor let him continue.

"If you have only one disk (that is,  $n = 1$ ), move it from pole A to pole B." So far, so good, but even the village idiot got that part right. "If you have more than one disk (that is,  $n > 1$ ), simply

1. "Ignore the bottom disk and solve the problem for  $n - 1$  disks, with the small modification that pole C is the destination and pole B is the spare." (See Figure 3-19b.)
2. "After you have done this,  $n - 1$  disks will be on pole C, and the largest disk will remain on pole A. So solve the problem for  $n - 1$  (recall that even the village idiot could do this) by moving the large disk from A to B." (See Figure 3-19c.)

3. "Now all you have to do is move the  $n - 1$  disks from pole C to pole B—that is, solve the problem with pole C as the source, pole B as the destination, and pole A as the spare." (See Figure 3-19d.)

There was silence for a few moments, and finally the Emperor said impatiently, "Well, are you going to tell us your solution or not?" The monk simply gave an all-knowing smile and vanished.

The Emperor obviously was not a recursive thinker, but you should realize that the monk's solution is perfectly correct. The key to the solution is the observation that you can solve the Towers problem of  $n$  disks by solving three smaller—in the sense of number of disks—Towers problems. Let `towers(count, source, destination, spare)` denote the problem of moving `count` disks from pole `source` to pole `destination`, using pole `spare` as a spare. Notice that this definition makes sense even if there are more than `count` disks on pole `source`; in this case, you concern yourself with only the top `count` disks and ignore the others. Similarly, the poles `destination` and `spare` might have disks on them before you begin; you ignore these, too, except that you may place only smaller disks on top of them.

You can restate the Emperor's problem as follows: Beginning with  $n$  disks on pole A and 0 disks on poles B and C, solve `towers(n, A, B, C)`. You can state the monk's solution as follows:

- Step 1.** Starting in the initial state—with all the disks on pole A—solve the problem

`towers(n-1, A, C, B)`

That is, ignore the bottom (largest) disk and move the top  $n - 1$  disks from pole A to pole C, using pole B as a spare. When you are finished, the largest disk will remain on pole A, and all the other disks will be on pole C.

- Step 2.** Now, with the largest disk on pole A and all others on pole C, solve the problem

`towers(1, A, B, C)`

That is, move the largest disk from pole A to pole B. Because this disk is larger than the disks already on the spare pole C, you really could not use the spare. However, fortunately—and obviously—you do not need to use the spare in this base case. When you are done, the largest disk will be on pole B and all other disks will remain on pole C.

- Step 3.** Finally, with the largest disk on pole B and all the other disks on pole C, solve the problem

`towers(n-1, C, B, A)`

That is, move the  $n - 1$  disks from pole C to pole B, using A as a spare. Notice that the destination pole B already has the largest disk, which you ignore. When you are done, you will have solved the original problem: All the disks will be on pole B.

The problem statement

The solution

The problem `solveTowers(count, source, destination, spare)` has the following pseudocode solution:

```

solveTowers(in count:integer, in source:Pole,
 in destination:Pole, in spare:Pole)

if (count is 1) {
 Move a disk directly from source to destination
}
else {
 solveTowers(count-1, source, spare, destination)
 solveTowers(1, source, destination, spare)
 solveTowers(count-1, spare, destination, source)
} // end if

```

This recursive solution follows the same basic pattern as the recursive solutions you saw earlier in this chapter:

1. You solve a Towers problem by solving other Towers problems.
2. These other Towers problems are smaller than the original problem; they have fewer disks to move. In particular, the number of disks decreases by 1 at each recursive call.
3. When a problem has only one disk—the base case—the solution is easy to solve directly.
4. The way that the problems become smaller ensures that you will reach a base case.

The solution to the Towers problem satisfies the four criteria of a recursive solution

Solving the Towers problem requires you to solve many smaller Towers problems recursively. Figure 3-20 illustrates the resulting recursive calls and their order when you solve the problem for three disks.

Now consider a Java implementation of this algorithm. Notice that since most computers do not have arms (at the time of this writing), the method moves a disk by giving directions to a human. Thus, the formal parameters that represent the poles are of type `char`, and the corresponding actual arguments could be '`A`', '`B`', and '`C`'. The call `solveTowers(3, 'A', 'B', 'C')` produces this output:

```

Move top disk from pole A to pole B
Move top disk from pole A to pole C
Move top disk from pole B to pole C
Move top disk from pole A to pole B
Move top disk from pole C to pole A
Move top disk from pole C to pole B
Move top disk from pole A to pole B

```

The solution for three disks

The Java method follows:

```
public static void solveTowers(int count, char source,
 char destination, char spare) {
 if (count == 1) {
 System.out.println("Move top disk from pole " + source +
 " to pole " + destination);
 }
 else {
 solveTowers(count-1, source, spare, destination); // X
 solveTowers(1, source, destination, spare); // Y
 solveTowers(count-1, spare, destination, source); // Z
 } // end if
} // end solveTowers
```

The three recursive calls in the method are labeled *X*, *Y*, and *Z*. These labels appear in the box trace of *solveTowers(3, 'A', 'B', 'C')* in Figure 3-21. The recursive calls are also numbered to correspond to the numbers used in Figure 3-20. (Figure 3-21 abbreviates *destination* as *dest* to save space.)

### 3.5 Recursion and Efficiency

Recursion is a powerful problem-solving technique that often produces very clean solutions to even the most complex problems. Recursive solutions can be easier to understand and to describe than iterative solutions. By using recursion, you can often write simple, short implementations of your solution.

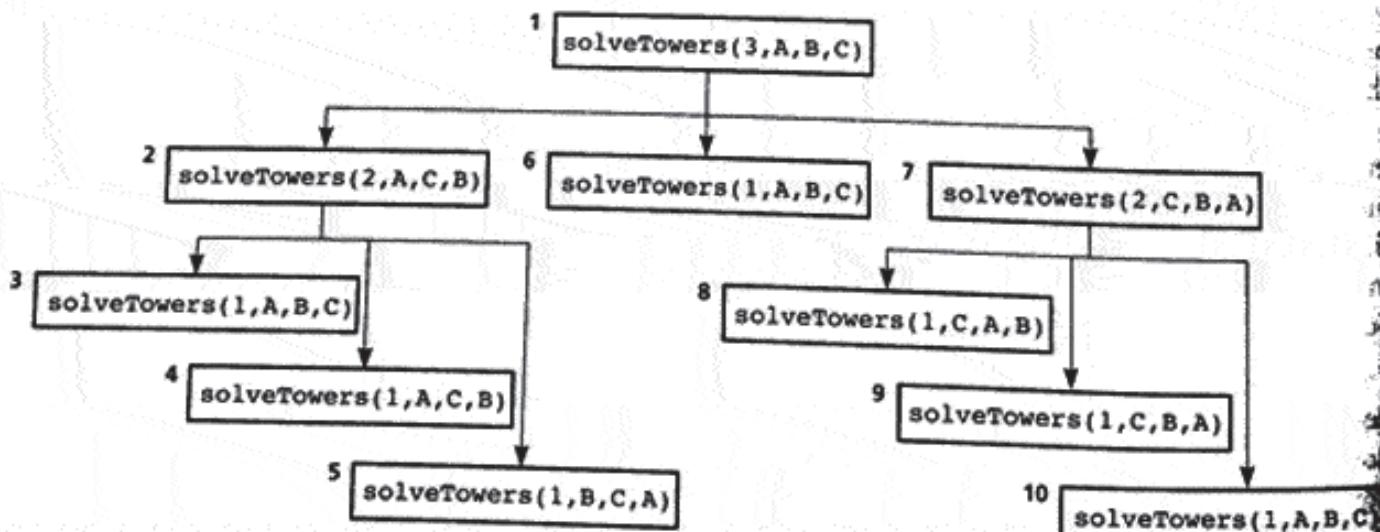


FIGURE 3-20

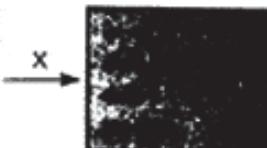
The order of recursive calls that results from *solveTowers(3, A, B, C)*

...at call 1 is made, and solveTowers begins execution:

```
count = 3
source = A
dest = B
spare = C
```

...at X, recursive call 2 is made, and the new invocation of the method begins execution:

```
count = 3
source = A
dest = B
spare = C
```



...at X, recursive call 3 is made, and the new invocation of the method begins execution:

```
count = 3
source = A
dest = B
spare = C
```

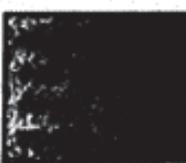
```
count = 2
source = A
dest = C
spare = B
```



...at Y, the base case, so a disk is moved, the return is made, and the method continues execution:

```
count = 3
source = A
dest = B
spare = C
```

```
count = 1
source = A
dest = B
spare = C
```



...at Z, recursive call 4 is made, and the new invocation of the method begins execution:

```
count = 3
source = A
dest = B
spare = C
```

```
count = 2
source = A
dest = C
spare = B
```



...at Y, the base case, so a disk is moved, the return is made, and the method continues execution:

```
count = 3
source = A
dest = B
spare = C
```

```
count = 1
source = A
dest = C
spare = B
```



...at Z, recursive call 5 is made, and the new invocation of the method begins execution:

```
count = 3
source = A
dest = B
spare = C
```

```
count = 2
source = A
dest = C
spare = B
```



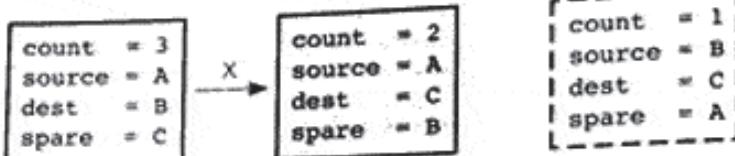
FIGURE 3-21

Box trace of `solveTowers(3, 'A', 'B', 'C')`

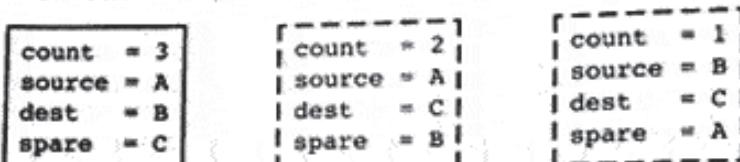
(continues)

(continued)

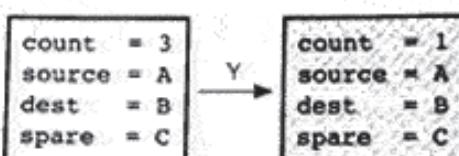
This is the base case, so a disk is moved, the return is made, and the method continues execution.



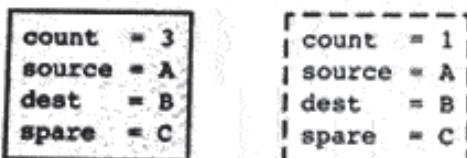
This invocation completes, the return is made, and the method continues execution.



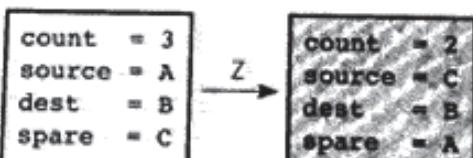
At point Y, recursive call 6 is made, and the new invocation of the method begins execution:



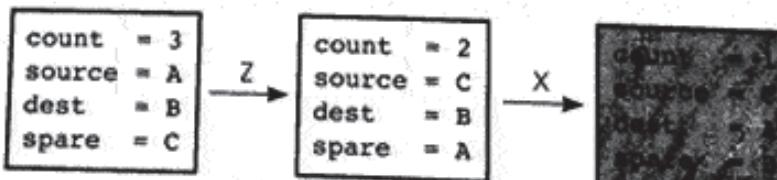
This is the base case, so a disk is moved, the return is made, and the method continues execution.



At point Z, recursive call 7 is made, and the new invocation of the method begins execution:



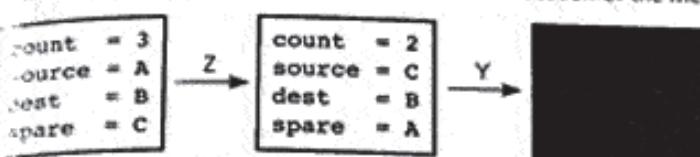
At point X, recursive call 8 is made, and the new invocation of the method begins execution:



This is the base case, so a disk is moved, the return is made, and the method continues execution.



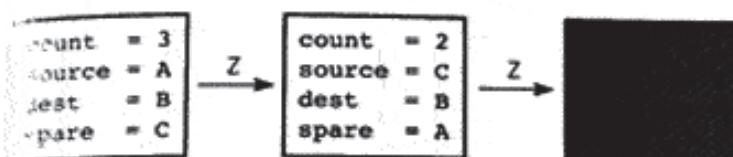
At point Y, recursive call 9 is made, and the new invocation of the method begins execution:



At point Z, it is the base case, so a disk is moved, the return is made, and the method continues execution.



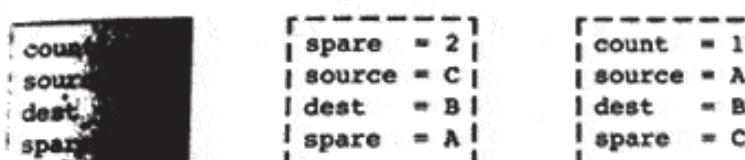
At point Z, recursive call 10 is made, and the new invocation of the method begins execution:



At point Z, it is the base case, so a disk is moved, the return is made, and the method continues execution.



The invocation completes, the return is made, and the method continues execution.



**FIGURE 3-21**

The overriding concern of this chapter has been to give you a solid understanding of recursion so that you will be able to construct recursive solutions on your own. Most of our examples, therefore, have been simple. Unfortunately, many of the recursive solutions in this chapter are so inefficient that you should not use them. The recursive methods *binarySearch* and *solveTowers* are the notable exceptions, as they are quite efficient.<sup>2</sup>

Two factors contribute to the inefficiency of some recursive solutions:

- The overhead associated with method calls
- The inherent inefficiency of some recursive algorithms

Factors that contribute to the inefficiency of some recursive solutions

2. Chapters 6 and 10 present other practical, efficient applications of recursion.

The first of these factors does not pertain specifically to recursive methods, but is true of methods in general. In most implementations of Java and other high-level programming languages, a method call incurs a bookkeeping overhead. As was mentioned earlier, each method call produces an activation record, which is analogous to a box in the box trace. Recursive methods magnify this overhead because a single initial call to the method can generate a large number of recursive calls. For example, the call `fact(n)` generates many recursive calls.

How the parameters are passed to a recursive method can also increase the amount of overhead. Look at the implementation of `writeBackward` that was presented earlier in the chapter. In each recursive call, a new string was generated that was one character shorter than the previous string, and sent as a parameter to the next recursive call. An alternative way to implement `writeBackward` is to include a second parameter `size`, which indicates the size of the string stored in `s` to write backward:

```
public static void writeBackward(String s, int size) {
 if (size > 0) {
 // write the last character
 System.out.println(s.charAt(size-1));
 // write the rest of the string backward
 writeBackward(s, size-1);
 } // end if
 // size == 0 is the base case - do nothing
} // end writeBackward
```

Note that, in this implementation, just the reference to the string `s` is sent to the next recursive call—not another string object. Hence, this reduces the overhead associated with each recursive call. This technique was also employed by many of the recursive algorithms in this chapter that involved arrays.

Note that the use of recursion, as is true with modularity in general, can greatly clarify complex programs. This clarification frequently more than compensates for the additional overhead. Thus, the use of recursion is often consistent with the multidimensional view of the cost of a computer program, which Chapter 2 describes.

However, you should not use recursion just for the sake of using recursion. For example, you probably should not use the recursive factorial method in practice. You easily can write an iterative factorial method given the iterative definition that was stated earlier in this chapter. The iterative method is almost as clear as the recursive one and is more efficient. There is no reason to incur the overhead of recursion when its use does not gain anything. *Recursion is truly valuable when a problem has no simple iterative solutions.*

The second point about recursion and efficiency is that some recursive algorithms are inherently inefficient. This inefficiency is a very different issue than that of overhead. It has nothing to do with how a compiler happens to

Recursion can clarify complex solutions

Do not use a recursive solution if it is inefficient and you have a clear, efficient iterative solution

implement a recursive method but rather is related to the method of solution that the algorithm employs.

As an example, recall the recursive solution for the multiplying rabbits problem that you saw earlier in this chapter:

$$\text{rabbit}(n) = \begin{cases} 1 & \text{if } n \text{ is 1 or 2} \\ \text{rabbit}(n - 1) + \text{rabbit}(n - 2) & \text{if } n > 2 \end{cases}$$

The diagram in Figure 3-11 illustrated the computation of  $\text{rabbit}(7)$ . Earlier, you were asked to think about what the diagram would look like for  $\text{rabbit}(10)$ . You thought about this question, you may have come to the conclusion that such a diagram would fill up most of this chapter. The diagram for  $\text{rabbit}(100)$  would fill up most of this universe!

The fundamental problem with  $\text{rabbit}$  is that it computes the same values over and over again. For example, in the diagram for  $\text{rabbit}(7)$ , you can see that  $\text{rabbit}(3)$  is computed five times. When  $n$  is moderately large, many of the values are recomputed literally trillions of times. This enormous number of computations makes the solution infeasible, even if each computation required only a trivial amount of work (for example, if you could perform 100 million of these computations per second).

However, do not conclude that the recurrence relation is of no use. One way to solve the rabbit problem is to construct an iterative solution based on this same recurrence relation. The iterative solution goes forward instead of backward and computes each value only once. You can use the following iterative method to compute  $\text{rabbit}(n)$  even for very large values of  $n$ .

The recursive version of  $\text{rabbit}$  is inherently inefficient

You can use  $\text{rabbit}$ 's recurrence relation to construct an efficient iterative solution

```
public static int iterativeRabbit(int n) {
 // Iterative solution to the rabbit problem.
 // initialize base cases:
 int previous = 1; // initially rabbit(1)
 int current = 1; // initially rabbit(2)
 int next = 1; // result when n is 1 or 2

 // compute next rabbit values when n >= 3
 for (int i = 3; i <= n; i++) {
 // current is rabbit(i-1), previous is rabbit(i-2)
 next = current + previous; // rabbit(i)

 previous = current; // get ready for
 current = next; // next iteration
 } // end for

 return next;
} // end iterativeRabbit
```

Converting recursive solutions to iterative solutions is easier to do than it is to convert iterative solutions to recursive ones. It is also more efficient to use an iterative solution.

A tail recursive method

Eliminating other forms of recursion

Thus, an iterative solution can be more efficient than a recursive solution. In certain cases, however, it may be easier to discover a recursive solution than an iterative solution. Therefore, you may need to convert a recursive solution to an iterative solution. This conversion process is easier if your recursive method calls itself once, instead of several times. Be careful when deciding whether your method calls itself more than once. Although the method `rabbits` calls itself twice, the method `binarySearch` calls itself once, even though you see two calls in the Java code. Those two calls appear within an `if` statement; only one of them will be executed.

Converting a recursive solution to an iterative solution is even easier when the solitary recursive call is the last *action* that the method takes. This situation is called **tail recursion**. For example, the method `writeBackward` exhibits tail recursion because its recursive call is the last action that the method takes. Before you conclude that this is obvious, consider the method `fact`. Although its recursive call appears last in the method definition, `fact`'s last action is the multiplication. Thus, `fact` is not tail recursive.

Recall the definition of `writeBackward` presented earlier in this section:

```
public static void writeBackward(String s, int size) {
 if (size > 0) {
 write the last character
 System.out.println(s.substring(size-1, size));
 writeBackward(s, size - 1); // write rest
 }
}
```

Because this method is tail recursive, its last recursive call simply repeats the method's action with altered arguments. You can perform this repetitive action by using an iteration that will be straightforward and often more efficient. For example, the following definition of `writeBackward` is iterative:

```
public static void writeBackward(String s, int size) {
 Iterative version.
 while (size > 0) {
 System.out.println(s.substring(size-1, size));
 --size;
 }
}
```

Because tail-recursive methods are often less efficient than their iterative counterparts and because the conversion of a tail-recursive method to an equivalent iterative method is rather mechanical, some compilers automatically replace tail recursion with iteration. Eliminating other forms of recursion is usually more complex, as you will see in Chapter 7, and is a task that you would need to undertake, if necessary.

Some recursive algorithms, such as *rabbit*, are inherently inefficient, while other recursive algorithms, such as the binary search,<sup>3</sup> are extremely efficient. You will learn how to determine the relative efficiency of a recursive algorithm in more advanced courses concerned with the analysis of algorithms. Chapter 10 introduces some of these techniques briefly.

Chapter 6 will continue the discussion of recursion by examining several difficult problems that have straightforward recursive solutions. Other chapters in this book use recursion as a matter of course.

## Summary

1. Recursion is a technique that solves a problem by solving a smaller problem of the same type.
2. When constructing a recursive solution, keep the following four questions in mind:
  - a. How can you define the problem in terms of a smaller problem of the same type?
  - b. How does each recursive call diminish the size of the problem?
  - c. What instance of the problem can serve as the base case?
  - d. As the problem size diminishes, will you reach this base case?
3. When constructing a recursive solution, you should assume that a recursive call's postcondition is true if its precondition is true.
4. You can use the box trace to trace the actions of a recursive method. These boxes resemble activation records, which many compilers use to implement recursion. (Chapter 6 discusses implementing recursion further.) Although the box trace is useful, it cannot replace an intuitive understanding of recursion.
5. Recursion allows you to solve problems—such as the Towers of Hanoi—whose iterative solutions are difficult to conceptualize. Even the most complex problems often have straightforward recursive solutions. Such solutions can be easier to understand, describe, and implement than iterative solutions.
6. Some recursive solutions are much less efficient than a corresponding iterative solution, due to their inherently inefficient algorithms and the overhead of method calls. In such cases, the iterative solution can be preferable. You can use the recursive solution, however, to derive the iterative solution.
7. If you can easily, clearly, and efficiently solve a problem by using iteration, you should do so.

## Cautions

1. A recursive algorithm must have a base case, whose solution you know directly without making any recursive calls. Without a base case, a recursive method will
3. The binary search algorithm also has an iterative formulation.

generate an infinite sequence of calls. When a recursive method contains more than one recursive call, you will often need more than one base case.

2. A recursive solution must involve one or more smaller problems that are each closer to a base case than is the original problem. You must be sure that these smaller problems eventually reach the base case. Failure to do so could result in an algorithm that does not terminate.
3. When developing a recursive solution, you must be sure that the solutions to the smaller problems really do give you a solution to the original problem. For example, *binarySearch* works because each smaller array is sorted and the value sought is between its first and last items.
4. The box trace, together with well-placed *System.out.println* statements, can be a good aid in debugging recursive methods. Such statements should report the point in the program from which each recursive call occurs as well as the values of input arguments and local variables at both entry to and exit from the methods. Be sure to remove these statements from the final version of the method.
5. A recursive solution that recomputes certain values frequently can be quite inefficient. In such cases, iteration may be preferable to recursion.

### Self-Test Exercises

1. The following method computes the sum of the first  $n \geq 1$  real numbers in an array. Show how this method satisfies the properties of a recursive method.

```
public static double sum(double anArray[], int n) {
 // Precondition: 1 <= n <= max size of anArray.
 // Postcondition: Returns the sum of the first n
 // items in anArray; anArray is unchanged.
 if (n == 1) {
 return anArray[0];
 }
 else {
 return anArray[n-1] + sum(anArray, n-1);
 } // end if
} // end product
```

2. Given an integer  $n > 0$ , write a recursive method *count* that writes the integers 1, 2, ...,  $n - 1$ ,  $n$ . *Hint:* What task can you do and what task can you ask a friend to do for you?
3. Write a recursive method that computes the product of the items in the array *anArray[first..last]*.
4. Of the following recursive methods that you saw in this chapter, identify those that exhibit tail recursion: *fact*, *writeBackward*, *writeBackward2*, *rabbit*, *c* in the Spock problem, *p* in the parade problem, *maxArray*, *binarySearch*, and *kSmallest*. Are the methods in Self-Test Exercises 1 through 3 tail recursive?
5. Compute *c(5, 1)* in the Spock problem.
6. Trace the execution of the method *solveTowers* to solve the Towers of Hanoi problem for *solveTowers(3, 'C', 'A', 'B')*. So in this case, 'C' is the original source, 'A' is the destination, and 'B' is the spare.

**Exercises**

1. The following recursive method `getNumberEqual` searches the array  $x$  of  $n$  integers for occurrences of the integer  $val$ . It returns the number of integers in  $x$  that are equal to  $val$ . For example, if  $x$  contains the 9 integers 1, 2, 4, 4, 5, 6, 7, 8, and 9, then `getNumberEqual(x, 9, 4)` returns the value 2 because 4 occurs twice in  $x$ .

```
public static int getNumberEqual(int x[], int n, int val) {
 if (n <= 0) {
 return 0;
 }
 else {
 if (x[n-1] == val) {
 return getNumberEqual(x, n-1, val) + 1;
 }
 else {
 return getNumberEqual(x, n-1, val);
 } // end if
 } // end if
} // end getNumberEqual
```

Demonstrate that this method is recursive by listing the criteria of a recursive solution and stating how the method meets each criterion.

2. Perform a box trace of the following calls to recursive methods that appear in this chapter. Clearly indicate each subsequent recursive call.

- a. `rabbit(4)`
- b. `writeBackward("loop")` (Use the version that strips the last character.)
- c. `maxArray` Find the maximum element in the array [4, 10, 12, 1, 8, 3, 6, 9]
- d. `kSmall` Search for the 3<sup>rd</sup> smallest element in the array [4, 10, 12, 1, 8, 3, 6, 9]
3. Write a Java method and an accompanying main program for the problem of *Organizing a Parade* as presented in this chapter. Test your code with  $n = 5$ . Does this return the same value as `rabbit(5)`?
- Given two integers `start` and `end`, where `end` is greater than `start`, write a recursive Java method that returns the sum of the integers from `start` through `end`, inclusive.
- Add output code to the Spock method `c(n, k)` that shows the actual sequence of calls that are made and the value that they will return when the method is executed. For example, `c(3, 2)` outputs the following:

```
c(3, 2) = c(2, 1) + c(2, 2)
c(2, 1) = c(1, 0) + c(1, 1)
c(1, 0) = 1
c(1, 1) = 1
c(2, 2) = 1
```

Use your modified version to run `c(4,2)` to show the actual order that the methods are called in Figure 3-12.

6. Given the following recursive method, answer each of the following questions.

```
public static void countDownByTwo(int n) {
 if (n != 1) {
 System.out.println(n + " ");
 countDownByTwo(n-2);
 } // end if
} // end countDownByTwo
```

- What happens when you execute the method with  $n = 7$ ?
  - What happens when you execute the method with  $n = 6$ ?
  - Answer the four questions for constructive recursive solutions to prove or disprove the correctness of this recursive solution.
  - If the answer to one or more of the questions in part c. indicates that this solution is incorrect, how would you change the method in such a way as to fix the problem?
7. The  $n^{\text{th}}$  Harmonic number is the sum of the reciprocals of the first  $n$  natural numbers:

$$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n}$$

Write a recursive method to compute the  $n^{\text{th}}$  Harmonic number.

- Write a method called `minChar` that returns the minimum character (using the ASCII collating sequence) in a given string. So for example, `minChar("hello")` returns 'e'.
- Write a recursive Java method that writes the digits of a positive decimal integer in reverse order.
- a. Write a recursive Java method `writeLine` that writes a character repeatedly to form a line of  $n$  characters. For example, `writeLine('*', 5)` produces the line \*\*\*\*\*.
- b. Now write a recursive method `writeBlock` that uses `writeLine` to write  $m$  lines of  $n$  characters each. For example, `writeBlock('*', 5, 3)` produces the output

```



```

11. What output does the following program produce?

```
import java.util.Arrays;
public class Exercisell {

 public static int guess(int[] c, int x) {
 if (c.length==1) {
 System.out.printf("z(%d) = %d\n", c.length-1, c[0]);
 return c[0];
 }
 else {
 System.out.printf("z(%d) = %d * z(%d) + %d\n",
 c.length-1, x, c.length-2, c[0]);
 }
 }
}
```

```
 return x*guess(Arrays.copyOfRange(c, 1, c.length), x) + c[0];
} // end if
} // end guess

public static void main(String[] args) {
 int[] x = {2, 4, 1};
 System.out.println(guess(x, 5));
} // end main
} // end Exercise11
```

12. What output does the following program produce? Try running it with a couple of different values for  $n$ . Can you guess what this computes?

```
public class Exercise12 {

 private static int search(int a, int b, int n) {
 int returnValue;

 int mid = (a + b)/2;
 System.out.printf("Enter: a = %d, b = %d, mid = %d\n",
 a, b, mid);
 if ((mid * mid <= n) && (n < (mid+1) * (mid+1))) {
 returnValue = mid;
 }
 else if (mid * mid > n) {
 returnValue = search(a, mid-1, n);
 }
 else {
 returnValue = search(mid+1, b, n);
 } // end if
 System.out.printf("Leave: a = %d, b = %d, mid = %d\n",
 a, b, mid);
 return returnValue;
 } // end search

 public static void main(String[] args) {
 int n = 64;
 System.out.printf("For n = %d, the result is %d\n",
 n, search(1, n, n));
 } // end main
} //end Exercise12
```

13. Consider the following method that converts a positive decimal number to base 8 and displays the result.

```
public static void displayOctal(int n) {
 if (n > 0) {
 if (n/8 > 0) {
 displayOctal(n/8);
 } // end if
 System.out.println(n%8);
 } // end if
} // end displayOctal
```

- Trace the method with  $n = 88$ .
- Describe how this method answers the four questions for constructing a recursive solution.

14. Consider the following program:

```
public class Exercise14 {
 public static int f(int n) {
 // Precondition: n >= 0.
 System.out.printf("Enter f: n = %d\n", n);
 switch (n) {
 case 1: case 2: case 3:
 return n + 1;
 default:
 return f(n-1) * f(n-3);
 } // end switch
 } // end f

 public static void main(String[] args) {
 System.out.println("f(8) is equal to " + f(8));
 } // end main
} // end Exercise14
```

Show the exact output of the program. What argument values, if any, could you pass to the method *f* to cause the program to run forever?

15. Consider the following method:

```
public static void recurse(int x, int y) {
 if (y > 0) {
 ++x;
 --y;
 System.out.println(x + " " + y);
 recurse(x, y);
 System.out.println(x + " " + y);
 } // end if
} // end recurse
```

Execute the method with  $x = 5$  and  $y = 3$ .

16. Perform a box trace of the recursive method *binarySearch*, which appears in the section “Binary Search,” with the array 2, 4, 5, 8, 9, 12, 15, 16, 20 for each of the following search values:

- 5
- 21
- 32

17. Imagine that you have 101 dalmatians; no two dalmatians have the same number of spots. Suppose that you create an array of 101 integers: The first integer is the number of spots on the first dalmatian, the second integer is the number of spots on the second dalmatian, and so on.

Your friend wants to know whether you have a dalmatian with 99 spots. Thus, you need to determine whether the array contains the integer 99.

- a. If you plan to use a binary search to look for the 99, what, if anything, would you do to the array before searching it?
- b. What is the index of the integer in the array that a binary search would examine first?
- c. If all your dalmatians have more than 99 spots, exactly how many comparisons will a binary search require to determine that 99 is not in the array?
- 18 This problem considers several ways to compute  $x^n$  for some  $n \geq 0$ .
- Write an iterative method *power1* to compute  $x^n$  for  $n \geq 0$ .
  - Write a recursive method *power2* to compute  $x^n$  by using the following recursive formulation:
- $$\begin{aligned}x^0 &= 1 \\x^n &= x \cdot x^{n-1} \text{ if } n > 0\end{aligned}$$
- Write a recursive method *power3* to compute  $x^n$  by using the following recursive formulation:
- $$\begin{aligned}x^0 &= 1 \\x^n &= (x^{n/2})^2 \text{ if } n > 0 \text{ and } n \text{ is even} \\x^n &= x \cdot (x^{n/2})^2 \text{ if } n > 0 \text{ and } n \text{ is odd}\end{aligned}$$
- How many multiplications will each of the methods *power1*, *power2*, and *power3* perform when computing  $3^{32}$ ?  $3^{19}$ ?
  - How many recursive calls will *power2* and *power3* make when computing  $3^{32}$ ?  $3^{19}$ ?
  - Modify the recursive *rabbit* method so that it is visually easy to follow the flow of execution. Instead of just adding "Enter" and "Leave" messages, indent the trace messages according to how "deep" the current recursive call is. For example, the call *rabbit(4)* should produce the output

```

Enter rabbit: n = 4
 Enter rabbit: n = 3
 Enter rabbit: n = 2
 Leave rabbit: n = 2 value = 1
 Enter rabbit: n = 1
 Leave rabbit: n = 1 value = 1
 Leave rabbit: n = 3 value = 2
 Enter rabbit: n = 2
 Leave rabbit: n = 2 value = 1
 Leave rabbit: n = 4 value = 3

```

Note how this output corresponds to figures such as Figure 3-11.

20. Consider the following recurrence relation:
- $$\begin{aligned}f(1) &= 1; f(2) = 2; f(3) = 3; f(4) = 2; f(5) = 4; \\f(n) &= 2 * f(n - 1) + f(n - 5) \text{ for all } n > 5.\end{aligned}$$
- Compute  $f(n)$  for the following values of  $n$ : 6, 7, 10, 12.

b. If you were careful, rather than computing  $f(15)$  from scratch (the way a recursive Java method would compute it), you would have computed  $f(6)$ , then  $f(7)$ , then  $f(8)$ , and so on up to  $f(15)$ , recording the values as you computed them. This ordering would have saved you the effort of ever computing the same value more than once. (Recall the nonrecursive version of the `rabbit` method discussed at the end of this chapter.)

Note that during the computation, you never need to remember all the previously computed values—only the last five. By taking advantage of these observations, write a Java method that computes  $f(n)$  for arbitrary values of  $n$ .

21. Write iterative versions of the following recursive methods: `fact`, `writeBackward`, `binarySearch`, `kSmall`.
22. Prove that the method `iterativeRabbit`, which appears in the section “Recursion and Efficiency,” is correct by using invariants.
- \* 23. Consider the problem of finding the greatest common divisor (`gcd`) of two positive integers  $a$  and  $b$ . The algorithm presented here is a variation of Euclid’s algorithm, which is based on the following theorem.<sup>4</sup>

**THEOREM.** If  $a$  and  $b$  are positive integers with  $a > b$  such that  $b$  is not a divisor of  $a$ , then  $\text{gcd}(a, b) = \text{gcd}(b, a \bmod b)$ .

This relationship between  $\text{gcd}(a, b)$  and  $\text{gcd}(b, a \bmod b)$  is the heart of the recursive solution. It specifies how you can solve the problem of computing  $\text{gcd}(a, b)$  in terms of another problem of the same type. Also, if  $b$  does divide  $a$ , then  $b = \text{gcd}(a, b)$ , so an appropriate choice for the base case is  $(a \bmod b) = 0$ .

This theorem leads to the following recursive definition:

$$\text{gcd}(a, b) = \begin{cases} b & \text{if } (a \bmod b) = 0 \\ \text{gcd}(b, a \bmod b) & \text{otherwise} \end{cases}$$

The following method implements this recursive algorithm:

```
public static int gcd(int a, int b) {
 if (a % b == 0) { // base case
 return b;
 }
 else {
 return gcd(b, a % b);
 } // end if
} // end gcd
```

- a. Prove the theorem.
- b. What happens if  $b > a$ ?
- c. How is the problem getting smaller? (That is, do you always approach a base case?) Why is the base case appropriate?
- \* 24. Let  $C(n)$  be the number of different groups of integers that can be chosen from the integers 1 through  $n - 1$  so that the integers in each group add up to  $n$ .

4. This book uses `mod` as an abbreviation for the mathematical operation modulo. In Java, the modulo operator is `%`.

example,  $4 = 1 + 1 + 1 + 1 = 1 + 1 + 2 = 2 + 2 \dots$ ). Write recursive definitions for  $C(n)$  under the following variations:

- You count permutations. For example, 1, 2, 1 and 1, 1, 2 are two groups that each add up to 4.
- You ignore permutations.
- Consider the following recursive definition:

$$\text{Acker}(m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ \text{Acker}(m - 1, 1) & \text{if } n = 0 \\ \text{Acker}(m - 1, \text{Acker}(m, n - 1)) & \text{otherwise} \end{cases}$$

This function, called **Ackermann's function**, is of interest because it grows rapidly with respect to the sizes of  $m$  and  $n$ . What is  $\text{Acker}(1, 2)$ ? Implement the function as a method in Java and do a box trace of  $\text{Acker}(1, 2)$ . (Caution: Even for modest values of  $m$  and  $n$ , Ackermann's function requires many recursive calls.)

## Programming Problems

---

Write a recursive method that counts the number of nonoverlapping occurrences of a substring in a given string.

- You have been offered a one month job that pays as follows: On the first day of the month, you are paid 1 cent. On the second day, 2 cents, on the third, 4 cents, and so forth, the amount doubles every day. Write a recursive method that, given the day number, computes the amount of money paid that day. Would you want this job?
- Implement `maxArray`, discussed in the section "Finding the Largest Item in an Array," as a Java method. What other recursive definitions of `maxArray` can you describe?
- Implement the `binarySearch` algorithm presented in this chapter for an array of strings.
- Implement `kSmallest`, discussed in the section "Finding the  $k^{\text{th}}$  Smallest Item in an Array," as a Java method. Use the first item of the array as the pivot.

## CHAPTER 4

# Data Abstraction: The Walls

This chapter elaborates on data abstraction, which was introduced in Chapter 2 as a technique for increasing the modularity of a program—for building “walls” between a program and its data structures. During the design of a solution, you will discover that you need to support several operations on the data and therefore need to define abstract data types (ADTs). This chapter introduces some simple abstract data types and uses them to demonstrate the advantages of abstract data types in general. In Part Two of this book, you will see several other important ADTs.

Only after you have clearly specified the operations of an abstract data type should you consider data structures for implementing it. This chapter explores implementation issues and introduces Java classes as a way to hide the implementation of an ADT from its users.

### 4.1 Abstract Data Types

#### 4.2 Specifying ADTs

- The ADT List
- The ADT Sorted List
- Designing an ADT
- Axioms (Optional)

#### 4.3 Implementing ADTs

- Java Classes Revisited
- Java Interfaces
- Java Packages
- An Array-Based Implementation of the ADT List

#### Summary

#### Cautions

#### Self-Test Exercises

#### Exercises

#### Programming Problems

## 4.1 Abstract Data Types

Modularity is a technique that keeps the complexity of a large program manageable by systematically controlling the interaction of its components. You can focus on one task at a time in a modular program without other distractions. Thus, a modular program is easier to write, read, and modify. Modularity also isolates errors and eliminates redundancies.

You can develop modular programs by piecing together existing software components with methods that have yet to be written. In doing so, you should focus on *what* a module does and not on *how* it does it. To use existing software, you need a clear set of specifications that details how the module behaves. To write new methods, you need to decide what you would like them to do and proceed under the assumption that they exist and work. In this way you can write the methods in relative isolation from one another, knowing what each one will do but not necessarily *how* each will eventually do it. That is, you should practice procedural abstraction.

While writing a module's specifications, you must identify details that you can hide within the module. The principle of information hiding involves not only hiding these details, but also making them *inaccessible* from outside a module. One way to understand information hiding is to imagine walls around the various tasks a program performs. These walls prevent the tasks from becoming entangled. The wall around each task *T* prevents the other tasks from "seeing" how *T* is performed. Thus, if task *Q* uses task *T*, and if the method for performing task *T* changes, task *Q* will not be affected. As Figure 4-1 illustrates, the wall prevents task *Q*'s method of solution from depending on task *T*'s method of solution.

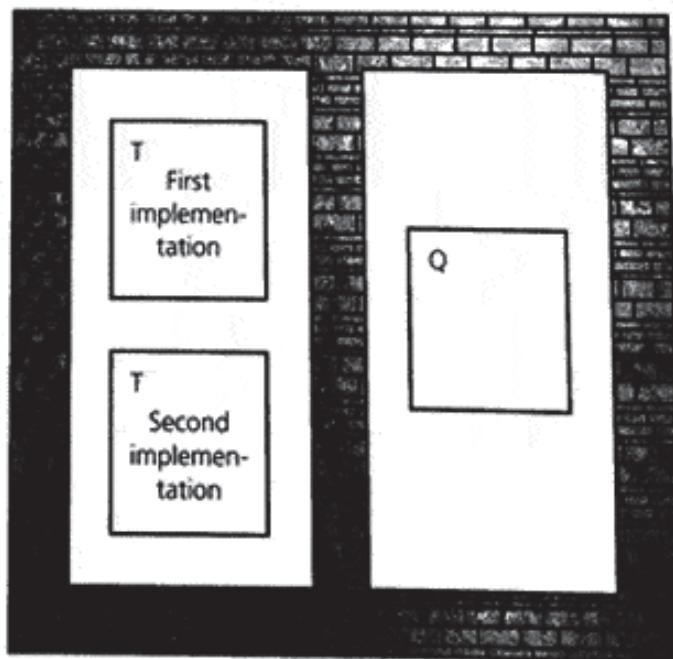


FIGURE 4-1

Isolated tasks: the implementation of task *T* does not affect task *Q*

The isolation of the modules cannot be total, however. Although task *Q* does not know *how* task *T* is performed, it must know *what* task *T* is and how to initiate it. For example, suppose that a program needs to operate on a sorted array of names. The program may, for instance, need to search the array for a given name or display the names in alphabetical order. The program thus needs a method *S* that sorts an array of names. Although the rest of the program knows that method *S* will sort an array, it should not care how *S* accomplishes its task. Thus, imagine a tiny slit in each wall, as Figure 4-2 illustrates. The slit is not large enough to allow the outside world to see the method's inner workings, but things can pass through the slit into and out of the method. For example, you can pass the array into the sort method, and the method can pass the sorted array out to you. What goes in and comes out is governed by the terms of the method's specifications, or contract: *If you use the method in this way, this is exactly what it will do for you.*

Often the solution to a problem requires operations on data. Such operations are broadly described in one of three ways:

- Add data to a data collection.
- Remove data from a data collection.
- Ask questions about the data in a data collection.

Typical operations on data

The details of the operations, of course, vary from application to application, but the overall theme is the management of data. Realize, however, that not all problems use or require these operations.

Both procedural and data abstraction ask you to think "what," not "how."

**Data abstraction** asks that you think in terms of *what* you can do to a collection of data independently of *how* you do it. Data abstraction is a technique that allows you to develop each data structure in relative isolation from the rest of the solution. The other modules of the solution will "know" what operations they can perform on the data, but they should not depend on how the data is stored or how the operations are performed. Again, the terms of the

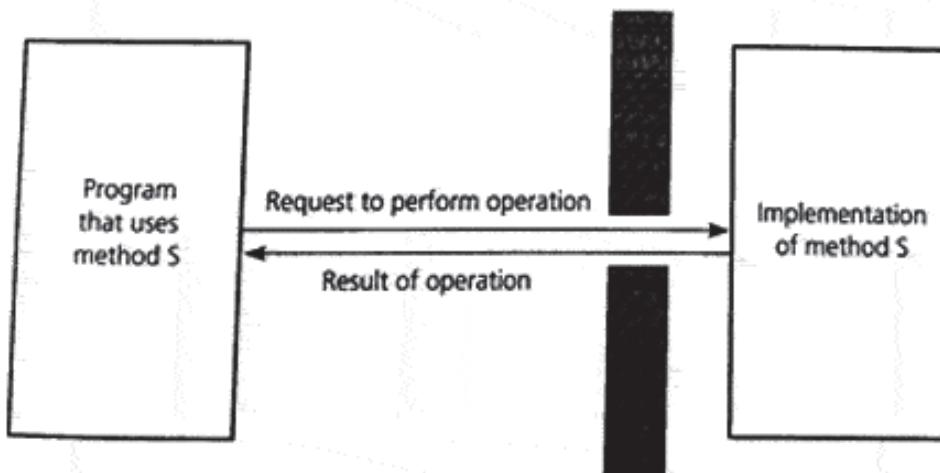


FIGURE 4-2

A slit in the wall

An ADT is a collection of data and a set of operations on that data

Specifications indicate what ADT operations do, but not how to implement them

Data structures are part of an ADT's implementation

Carefully specify an ADT's operations before you implement them

ADTs and data structures are not the same

contract are *what* and not *how*. Thus, data abstraction is a natural extension of procedural abstraction.

A collection of data together with a set of operations on that data are called an abstract data type, or ADT. For example, suppose that you need to store a collection of names in a manner that allows you to search rapidly for a given name. The binary search algorithm described in Chapter 3 enables you to search an array efficiently, if the array is sorted. Thus, one solution to this problem is to store the names sorted in an array and to use a binary search algorithm to search the array for a specified name. You can view the *sorted array together with the binary search algorithm* as an ADT that solves this problem.

The description of an ADT's operations must be rigorous enough to specify completely their effect on the data, yet it must not specify how to store the data nor how to carry out the operations. For example, the ADT operations should not specify whether to store the data in consecutive memory locations or in disjoint memory locations. You choose a particular data structure when you implement an ADT.

Recall that a data structure is a construct that you can define within a programming language to store a collection of data. For example, arrays, which are built into Java, are data structures. However, you can invent other data structures. For example, suppose that you wanted a data structure to store both the names and salaries of a group of employees. You could use the following Java statements:

```
final int MAX_NUMBER = 500;
String[] names = new String[MAX_NUMBER];
double[] salaries = new double[MAX_NUMBER];
```

Here the employee *names[i]* has a salary of *salaries[i]*. The two arrays *names* and *salaries* together form a data structure, yet Java has no single data type to describe it.

When a program must perform data operations that are not directly supported by the language, you should first design an abstract data type and carefully specify what the ADT operations are to do (the contract). Then—and only then—should you implement the operations with a data structure. If you implement the operations properly, the rest of the program will be able to assume that the operations perform as specified—that is, that the terms of the contract are honored. However, the program must not depend on a particular approach for supporting the operations.

An abstract data type is not another name for a data structure.

To give you a better idea of the conceptual difference between an ADT and a data structure, consider a refrigerator's ice dispenser, as Figure 4-3 illustrates. It has water as input and produces as output either chilled water, crushed ice, or ice cubes, according to which one of the three buttons you push. It also has an indicator that lights when no ice is presently available. The dispenser is analogous to an abstract data type. The water is analogous to data; the operations are *chill*, *crush*, *cube*, and *isEmpty*. At this level of design, you are not concerned with how the dispenser will perform its operations, only that it

**KEY CONCEPTS****ADTs Versus Data Structures**

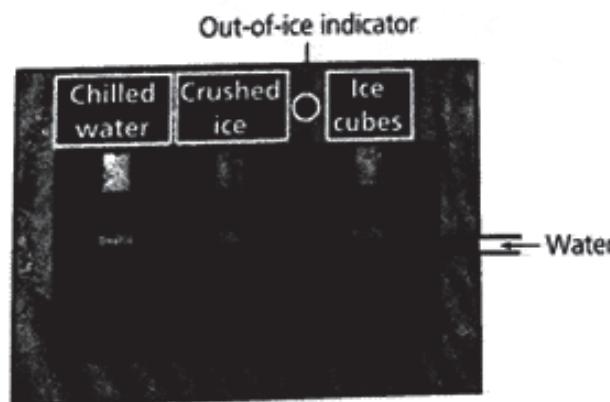
- An abstract data type is a collection of data and a set of operations on that data.
- A data structure is a construct within a programming language that stores a collection of data.

performs them. If you want crushed ice, do you really care how the dispenser accomplishes its task as long as it does so correctly? Thus, after you have specified the dispenser's methods, you can design many uses for crushed ice without knowing how the dispenser accomplishes its tasks and without the extraction of engineering details.

Eventually, however, someone must build the dispenser. Exactly how will this machine produce crushed ice, for example? It could first make ice cubes and then either crush them between two steel rollers or smash them into small pieces by using hammers. Many other techniques are possible. The internal structure of the dispenser corresponds to the implementation of the ADT in a programming language—that is, to a data structure.

Although the owner of the dispenser does not care about its inner workings, he or she does want a design that is as efficient in its operation as possible. Similarly, the dispenser's manufacturer wants a design that is as easy and cheap to build as possible. You should have these same concerns when you choose a data structure to implement an ADT in Java. Even if you do not implement the ADT yourself, but instead use an already implemented ADT, you—like the person who buys a refrigerator—should care at least about the ADT's efficiency.

Notice that the dispenser is surrounded by steel walls. The only breaks in the walls accommodate the input (water) to the machine and its output (chilled water, crushed ice, or ice cubes). Thus, the machine's interior mechanisms are not only hidden from the user but also are inaccessible. In addition, the mechanism of one operation is hidden from and inaccessible to another operation.

**FIGURE 4-3**

A dispenser of chilled water, crushed ice, and ice cubes

A program should not depend on the details of an ADT's implementation.

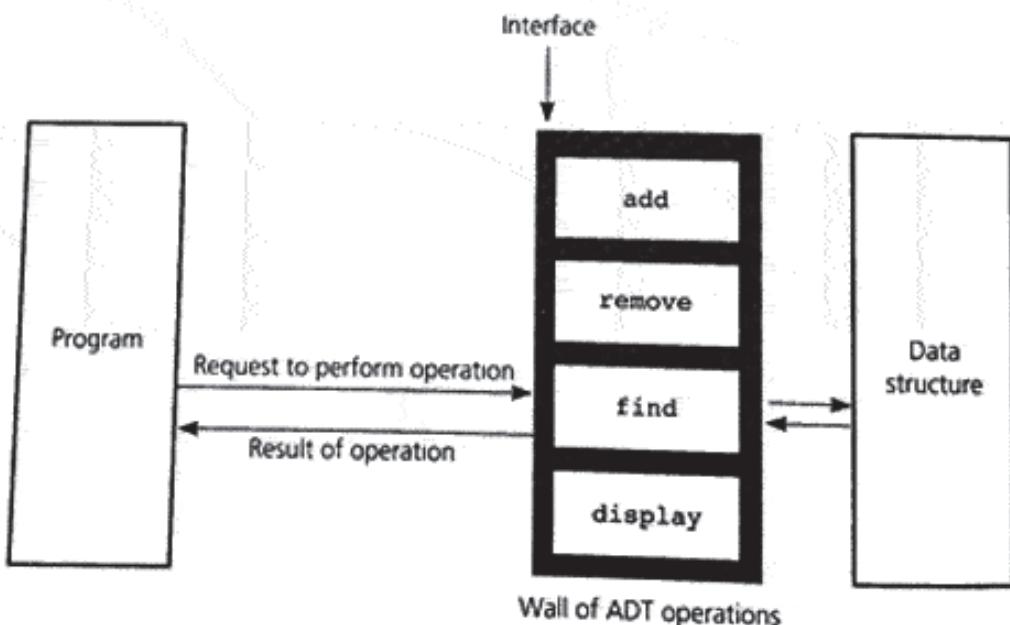
Using an ADT is like using a vending machine

This modular design has benefits. For example, you can improve the operation *crush* by modifying its module without affecting the other modules. You could also add an operation by adding another module to the machine without affecting the original three operations. Thus, both abstraction and information hiding are at work here.

To summarize, data abstraction results in a wall of ADT operations between data structures and the program that accesses the data within these data structures, as Figure 4-4 illustrates. If you are on the program's side of the wall, you will see an interface that enables you to communicate with the data structure. That is, you request the ADT operations to manipulate the data in the data structure, and they pass the results of these manipulations back to you.

This process is analogous to using a vending machine. You press buttons to communicate with the machine and obtain something in return. The machine's external design dictates how you use it, much as an ADT's specifications govern what its operations are and what they do. As long as you use a vending machine according to its design, you can ignore its inner technology. As long as you agree to access data only by using ADT operations, your program can be oblivious to any change in the data structures that implement the ADT.

The following pages describe how to use an abstract data type to realize data abstraction's goal of separating the operations on data from the implementation of these operations. In doing so, we will look at several examples of ADTs.



**FIGURE 4-4**

A wall of ADT operations isolates a data structure from the program that uses it

## 4.2 Specifying ADTs

To elaborate on the notion of an abstract data type, consider a list that you might encounter, such as a list of chores, a list of important dates, a list of addresses, or the grocery list pictured in Figure 4-5. As you write a grocery list, where do you put new items? Assuming that you write a neat one-column list, you probably add new items to the end of the list. You could just as well add items to the beginning of the list or add them so that your list is sorted alphabetically. Regardless, the items on a list appear in a sequence. The list has one first item and one last item. Except for the first and last items, each item has a unique predecessor and a unique successor. The first item—the head or front of the list—does not have a predecessor, and the last item—the tail or end of the list—does not have a successor.

Lists contain items of the same type: You can have a list of grocery items or a list of phone numbers. What can you do to the items on a list? You might count the items to determine the length of the list, add an item to the list, remove an item from the list, or look at (retrieve) an item. The items on a list, together with operations that you can perform on the items, form an abstract data type. You must specify the behavior of the ADT's operations on its data, that is, the list items. It is important that you focus only on specifying the operations and not on how you will implement them. In other words, do not bring to this discussion any preconceived notion of a data structure that the term "list" might suggest.

Where do you add a new item and which item do you want to look at? The various answers to these questions lead to several kinds of lists. You might decide to add, delete, and retrieve items only at the end of the list or only at the front of the list or at both the front and end of the list. The specifications of these lists are left as an exercise; next we will discuss a more general list.

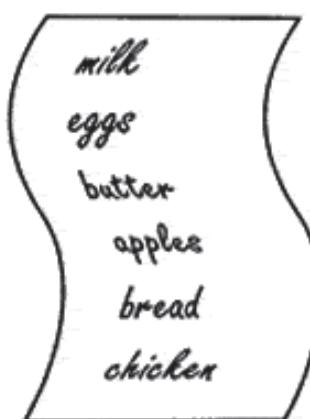


FIGURE 4-5

A grocery list

## The ADT List

Once again, consider the grocery list pictured in Figure 4-5. The previously described lists, which manipulate items at one or both ends of the list, are not really adequate for an actual grocery list. You would probably want to access items anywhere on the list. That is, you might look at the item at position  $i$ , delete the item at position  $i$ , or insert an item at position  $i$  on the list. Such operations are part of the ADT list.

Note that it is customary to include an initialization operation that creates an empty list. Other operations that determine whether the list is empty or the length of the list are also useful.

Although the six items on the list in Figure 4-5 have a sequential order, they are not necessarily sorted by name. Perhaps the items appear in the order in which they occur on the grocer's shelves, but more likely they appear in the order in which they occurred to you as you wrote the list. The ADT list is simply an ordered collection of items that you reference by position number. But to make this ADT list more like other built-in ADTs you will find in Java, the position number will start at zero. Note this is similar to the way that Java indexes an array; the subscript of the first element in an array is zero.

You reference list items by their position within the list

### KEY CONCEPTS

#### ADT List Operations

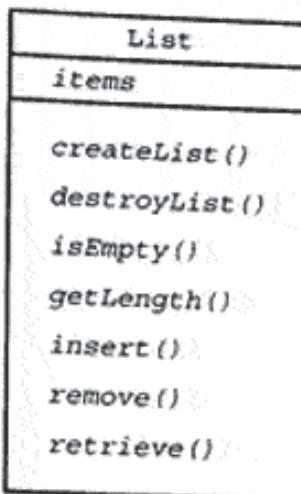
1. Create an empty list.
2. Determine whether a list is empty.
3. Determine the number of items on a list.
4. Add an item at a given position in the list.
5. Remove the item at a given position in the list.
6. Remove all the items from the list.
7. Retrieve (get) the item at a given position in the list.

The following pseudocode specifies the operations for the ADT list in more detail. Figure 4-6 shows the UML diagram for this ADT.

To get a more precise idea of how the operations work, apply them to the grocery list

*milk, eggs, butter, apples, bread, chicken*

where milk is the first item on the list and chicken is the last item. To begin, consider how you can construct this list by using the operations of the ADT.

**FIGURE 4-6**

UML diagram for ADT *List*

list. One way is first to create an empty list *aList* and then use a series of insertion operations to append successively the items to the list as follows:

```

aList.createList()
aList.add(0, milk)
aList.add(1, eggs)
aList.add(2, butter)
aList.add(3, apple)
aList.add(4, bread)
aList.add(5, chicken)

```

The notation<sup>1</sup> *aList.O* indicates that an operation *O* applies to the list *aList*.

Notice that the list's insertion operation can insert new items into any position of the list, not just at its front or end. According to *add*'s specification, if a new item is inserted into position *i*, the position of each item that was at a position of *i* or greater is increased by 1. Thus, for example, if you start with the previous grocery list and you perform the operation

```
aList.add(3, nuts)
```

the list *aList* becomes

*milk, eggs, butter, nuts, apples, bread, chicken*

1. This notation is similar to the Java implementation of the ADT.

**KEY CONCEPTS****Pseudocode for the ADT List Operations**

```
+createList()
// Creates an empty list.

+isEmpty():boolean {query}
// Determines whether a list is empty.

+size():integer {query}
// Returns the number of items that are in a list.

+add(in index:integer, in item:ListItemType)
// Inserts item at position index of a list, if
// 0 <= index <= size().
// If index < size(), items are renumbered as
// follows: The item at index becomes the item at
// index+1, the item at index+1 becomes the
// item at index+2, and so on.
// Throws an exception when index is out of range or if
// the item cannot be placed on the list (list full).

+remove(in index:integer)
// Removes the item at position index of a list, if
// 0 <= index < size(). If index < size()-1, items are
// renumbered as follows: The item at index+1 becomes
// the item at index, the item at index+2 becomes the
// item at index+1, and so on.
// Throws an exception when index is out of range or if
// the list is empty.

+removeAll()
// Removes all the items in the list.

+get(index):ListItemType {query}
// Returns the item at position index of a list
// if 0 <= index < size(). The list is
// left unchanged by this operation.
// Throws an exception if index is out of range.
```

All items that had position numbers greater than or equal to 3 before the insertion now have their position numbers increased by 1 after the insertion.

Similarly, the deletion operation specifies that if an item is deleted from position  $i$ , the position of each item that was at a position greater than  $i$  is decreased by 1. Thus, for example, if `aList` is the list

*milk, eggs, butter, nuts, apples, bread, chicken*

and you perform the operation

```
aList.remove(4)
```

the list becomes

*milk, eggs, butter, nuts, bread, chicken*

All items that had position numbers greater than 4 before the deletion now have their position numbers decreased by 1 after the deletion.

These examples illustrate that an ADT can specify the effects of its operations without having to indicate how to store the data. The specifications of the seven operations are the sole terms of the contract for the ADT list: *If you request that these operations be performed, this is what will happen.* The specifications contain no mention of how to store the list or how to perform the operations; they tell you only what you can do to the list. It is of fundamental importance that the specification of an ADT *not* include implementation issues. This restriction on the specification of an ADT is what allows you to build a wall between an implementation of an ADT and the program that uses it. (Such a program is called a *client*.) The behavior of the operations is the only thing on which a program should depend.

Note that the insertion, deletion, and retrieval operations throw an exception when the argument *index* is out of range. This technique provides the ADT with a simple mechanism to communicate operation failure to its client. For example, if you try to delete the tenth item from a five-item list, `remove` can throw an exception indicating that *index* is out of range. Exceptions enable the client to handle error situations in an implementation-independent way.

What does the specification of the ADT list tell you about its behavior? It is apparent that the list operations fall into the three broad categories presented earlier in this chapter.

- The operation `add` adds data to a data collection.
- The operations `remove` and `removeAll` remove data from a data collection.
- The operations `isEmpty`, `size`, and `get` ask questions about the data in a data collection.

An ADT specification should not include implementation issues

A program should depend only on the behavior of the ADT

Once you have satisfactorily specified the behavior of an ADT, you can design applications that access and manipulate the ADT's data solely in terms of its operations and without regard for its implementation. As a simple example, suppose that you want to display the items in a list. Even though the wall between the implementation of the ADT list and the rest of the program prevents you from knowing how

An implementation-independent application of the ADT list

the list is stored, you can write a method *displayList* in terms of the operations that define the ADT list. The pseudocode for such a method follows:<sup>2</sup>

```
displayList(in aList>List)
// Displays the items on the list aList.

for (index = 0 through aList.size()-1) {
 dataItem = aList.get(index)
 Display dataItem
} // end for
```

Notice that as long as the ADT list is implemented correctly, the *displayList* method will perform its task. In this case, *get* successfully retrieves each list item, because *index*'s value is always valid.

The method *displayList* does not depend on *how* you implement the list. That is, the method will work regardless of whether you use an array or some other data structure to store the list's data. This feature is a definite advantage of abstract data types. In addition, by thinking in terms of the available ADT operations, you will not be distracted by implementation details. Figure 4-7 illustrates the wall between *displayList* and the implementation of the ADT list.

As another application of the ADT operations, suppose that you want a method *replace* that replaces the item in position *i* with a new item. If the *i*<sup>th</sup> item exists, *replace* deletes the item and inserts the new item at position *i*, as follows:

```
replace(in aList>List, in i:integer,
 in newItem>ListItemType
// Replaces the ith item on the list aList with
// newItem.

if (i >= 0 and i < aList.size()) {
 aList.remove(i)
 aList.add(i, newItem)
} // end if
```

You can use ADT operations in an application without the distraction of implementation details

In both of the preceding examples, notice how you can focus on the task at hand without the distraction of implementation details such as arrays. With less to worry about, you are less likely to make an error in your logic when you use the ADT operations in applications such as *displayList* and *replace*. Likewise, when you finally implement the ADT operations in Java, you will not be distracted by these applications. In addition, because *displayList* and *replace* do not depend on any implementation decisions that you make for

---

2. In this example, *displayList* is not an ADT operation, so a procedural notation that specifies *aList* as a parameter is used.

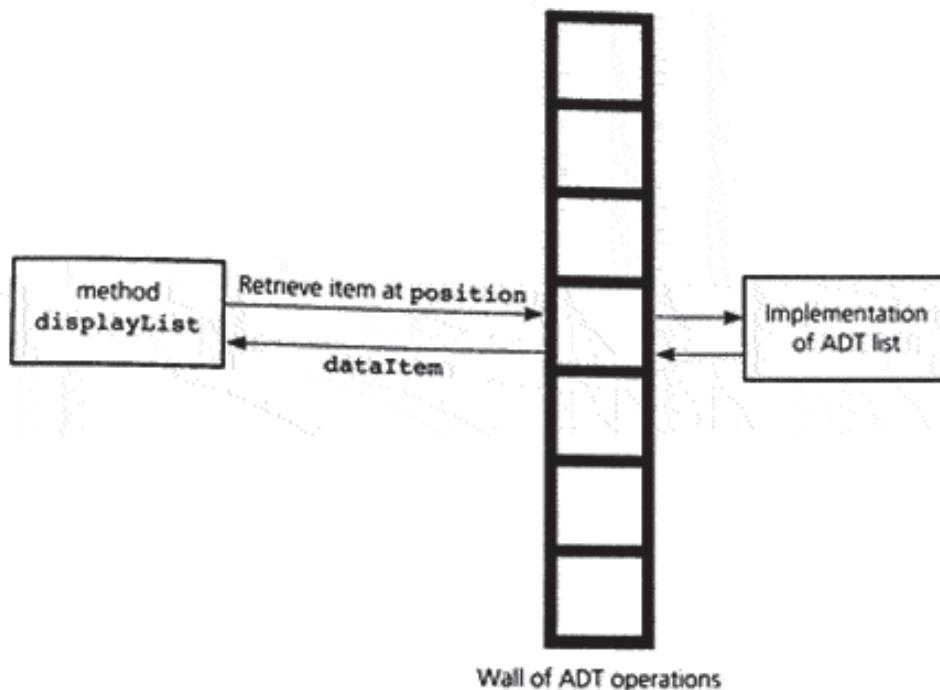


FIGURE 4-7

The wall between *displayList* and the implementation of the ADT list

the ADT list, they are not altered by your decisions. These assertions assume that you do not change the specifications of the ADT operations when you implement them. However, as Chapter 2 pointed out, developing software is not a linear process. You may realize during implementation that you need to refine your specifications. Clearly, changes to the specification of any module affect any already-designed uses of that module.

To summarize, you can specify the behavior of an ADT independently of its implementation. Given such a specification, and without any knowledge of how the ADT will be implemented, you can design applications that use the ADT's operations to access its data.

## The ADT Sorted List

One of the most frequently performed computing tasks is the maintenance, in some *specified* order, of a collection of data. Many examples immediately come to mind: students placed in order by their names, baseball players listed in order by their batting averages, and corporations listed in order by their assets. These orders are called *sorted*. In contrast, the items on a grocery list might be ordered—the order in which they appear on the grocer's shelves, for example—but they are probably not sorted by name.

The problem of *maintaining* sorted data requires more than simply sorting the data. Often you need to insert some new data item into its proper, sorted place. Similarly, you often need to delete some data item. For example,

The ADT sorted list maintains items in sorted order

suppose your university maintains an alphabetical list of the students who are currently enrolled. The registrar must insert names into and delete names from this list because students constantly enroll in and leave school. These operations should preserve the sorted order of the data.

The following specifications define the operations for the ADT sorted list.

#### KEY CONCEPTS

#### Pseudocode for the ADT Sorted List Operations

```

+createSortedList()
// Creates an empty sorted list.

+sortedIsEmpty():boolean {query}
// Determines whether a sorted list is empty.

+sortedSize():integer {query}
// Returns the number of items that are in a sorted list.

+sortedAdd(in item:ListItemType)
// Inserts item into its proper sorted position in a
// sorted list. Throws an exception if the item
// cannot be placed on the list (list full).

+sortedRemove(in item:ListItemType)
// Deletes item from a sorted list.
// Throws an exception if the item is not found.

+sortedGet(in index:integer)
// Returns the item at position index of a
// sorted list, if 0 <= index < sortedSize().
// The list is left unchanged by this operation.
// Throws an exception if the index is out of range.

+locateIndex(in item:ListItemType):integer {query}
// Returns the position where item belongs or
// exists in a sorted list; item and the list are
// unchanged.

```

The ADT sorted list differs from the ADT list in that a sorted list inserts and deletes items by their values and not by their positions. For example, `sortedAdd` determines the proper position for `item` according to its value. Also, `locateIndex`—which determines the position of any item, given its value—is a sorted list operation but not a list operation. However, `sortedGet` is like list's `get`: Both operations retrieve an item, given its position. The method `sortedGet` enables you, for example, to retrieve and then display each item in a sorted list.

## Designing an ADT

The design of an abstract data type should evolve naturally during the problem-solving process. As an example of how this process might occur, suppose that you want to determine the dates of all the holidays in a given year. One way to do this is to examine a calendar. That is, you could consider each day in the year and ascertain whether that day is a holiday. The following pseudocode is thus a possible solution to this problem:

```
listHolidays(in year:integer)
// Displays the dates of all holidays in a given year.

 date = date of first day of year
 while (date is before the first day of year+1) {
 if (date is a holiday) {
 write (date + " is a holiday")
 } // end if
 date = date of next day
 } // end while
```

What data is involved here? Clearly, this problem operates on dates, where a date consists of a month, day, and year. What operations will you need to solve the holiday problem? Your ADT must specify and restrict the legal operations on the dates just as the fundamental data type *int* restricts you to operations such as addition and comparison. You can see from the previous pseudocode that you must

What data does a problem require?

- Determine the date of the first day of a given year
- Determine whether a date is before another date
- Determine whether a date is a holiday
- Determine the date of the day that follows a given date

What operations does a problem require?

Thus, you could define the following operations for your ADT:

```
+firstDay(in year:integer):Date {query}
// Returns the date of the first day of a given year.

+isBefore(in date1:Date,
 in date2:Date) : boolean {query}
// Returns true if date1 is before date2,
// otherwise returns false.

+isHoliday(in aDate:Date) : boolean {query}
// Returns true if date is a holiday,
// otherwise returns false.
```

```
+nextDay(in aDate:Date) : Date
// Returns the date of the day after a given date.
```

The *ListHolidays* pseudocode now appears as follows:

```
listHolidays(in year:integer)
// Displays the dates of all holidays in a given year.

date = firstDay(year)
while (isBefore(date, firstDay(year+1))) {
 if (isHoliday(date)) {
 write (date + " is a holiday ")
 } // end if
 date = nextDay(date)
} // end while
```

Thus, you can design an ADT by identifying data and choosing operations that are suitable to your problem. After specifying the operations, you use them to solve your problem independently of the implementation details of the ADT.

**An appointment book.** As another example of an ADT design, imagine that you want to create a computerized appointment book that spans a one-year period. Suppose that you make appointments only on the hour and half hour between 8 A.M. and 5 P.M. For simplicity, assume that all appointments are 30 minutes in duration. You want your system to store a brief notation about the nature of each appointment along with the date and time.

To solve this problem, you can define an ADT appointment book. The data items in this ADT are the appointments, where an appointment consists of a date, time, and purpose. What are the operations? Two obvious operations are

- Make an appointment for a certain date, time, and purpose. (You will want to be careful that you do not make an appointment at an already occupied time.)
- Cancel the appointment for a certain date and time.

In addition to these operations, it is likely that you will want to

- Ask whether you have an appointment at a given time.
- Determine the nature of your appointment at a given time.

Finally, ADTs typically have initialization operations.

Thus, the ADT appointment book can have the following operations:

```
+createAppointmentBook()
// Creates an empty appointment book.

+isAppointment(in apptDate:Date,
 in apptTime:Time):boolean {query}
// Returns true if an appointment exists for the date
```

*and time specified; otherwise returns false.*

**makeAppointment(in apptDate:Date, in apptTime:Time,  
in purpose:string):boolean**  
*Creates the appointment for the date, time, and purpose  
specified as long as it does not conflict with an  
existing appointment.*  
*Returns true if successful, false otherwise.*

**cancelAppointment(in apptDate:Date,  
in apptTime:Time):boolean**  
*Deletes the appointment for the date and time specified.  
Returns true if successful, false otherwise.*

**checkAppointment(in apptDate:Date,  
in apptTime:Time):string {query}**  
*Returns the purpose of the appointment at  
the given date/time, if one exists. Otherwise, returns  
null.*

Programmers can use these ADT operations to design other operations on the appointments. For example, suppose that you want to change the date or time of a particular appointment within the existing appointment book *apptBook*. The following pseudocode indicates how to accomplish this task by using the *makeAppointment* and *cancelAppointment* ADT operations:

*change the date or time of an appointment*

```

read oldDate, oldTime, newDate, newTime)
 get purpose of appointment
purpose = apptBook.checkAppointment(oldDate, oldTime)
if 'purpose exists' {
 // see if new date/time is available
 if (apptBook.isAppointment(newDate, newTime)) {
 // new date/time is booked
 write ("You already have an appointment at " + newTime +
 " on " + newDate)

 }
else { // new date/time is available
 apptBook.cancelAppointment(oldDate, oldTime)
 if (apptBook.makeAppointment(newDate, newTime,
 purpose)){
 write ("Your appointment has been rescheduled to" +
 newTime + " on " + newDate)
 }
}
}

```

```

else {
 write ("You do not have an appointment at " + oldTime +
 " on " + oldDate)
} // end if

```

You can use an ADT without knowledge of its implementation

You can use an ADT to implement another ADT

Again notice that you can design applications of ADT operations without knowing how the ADT is implemented. The exercises at the end of the chapter provide examples of other tasks that you can perform with this ADT.

**ADTs that suggest other ADTs.** Both of the previous examples require you to represent a date; the appointment book example also requires you to represent the time. Java has a *java.util.Date* class that you can use to represent the date and time. You can also design ADTs to represent these items. It is not unusual for the design of one ADT to suggest other ADTs. In fact, you can use one ADT to implement another ADT. The programming problems at the end of this chapter ask you to design and implement the simple ADTs date and time.

This final example also describes an ADT that suggests other ADTs for its implementation. Suppose that you want to design a database of recipes. You could think of this database as an ADT: The recipes are the data items, and some typical operations on the recipes could include the following:

```

+insertRecipe(in aRecipe:Recipe)
// Inserts recipe into the database.

+deleteRecipe(in aRecipe:Recipe)
// Deletes recipe from the database.

+retrieveRecipe(in name:string):Recipe {query}
// Retrieves the named recipe from the database.

```

This level of the design does not indicate such details as where *insertRecipe* will place a recipe into the database.

Now imagine that you want to write a method that scales a recipe retrieved from the database: If the recipe is for *n* people, you want to revise it so that it will serve *m* people. Suppose that the recipe contains measurements such as  $2\frac{1}{2}$  cups, 1 tablespoon, and  $\frac{1}{4}$  teaspoon. That is, the quantities are given as mixed numbers—integers and fractions—in units of cups, tablespoons, and teaspoons.

This problem suggests another ADT—measurement—with the following operations:

```

+getMeasure():Measurement {query}
// Returns the measure.

+setMeasure(in m:Measurement)
// Sets the measure.

```

```

+scaleMeasure(in scaleFactor: float):Measurement
 // Multiplies measure by a fractional scaleFactor, which
 // has no units, and returns the result.

+convertMeasure(in oldUnits:MeasureUnit,
 in newUnits:MeasureUnit):Measurement {query}
 // Converts measure from its old units to a measure in
 // new units, and returns the result.

```

Suppose that you want the ADT measurement to perform exact fractional arithmetic. Because our planned implementation language Java does not have a data type for fractions and floating-point arithmetic is not exact, another ADT called fraction is in order. Its operations could include addition, subtraction, multiplication, and division of fractions. For example, you could specify addition as

```

addFractions(in first:Fraction, in second:Fraction):Fraction
 // Adds two fractions and returns the sum reduced to lowest
 // terms.

```

Moreover, you could include operations to convert a mixed number to a fraction and to convert a fraction to a mixed number when feasible.

When you finally implement the ADT measurement, you can use the ADT fraction. That is, you can use one ADT to implement another ADT.

## Axioms (Optional)

The previous specifications for ADT operations have been stated rather informally. For example, they rely on your intuition to know the meaning of “an item is at position  $i$ ” in an ADT list. This notion is simple, and most people will understand its intentions. However, some abstract data types are much more complex and less intuitive than a list. For such ADTs, you should use a more rigorous method of defining the behavior of their operations: You must supply a set of mathematical rules—called axioms—that precisely specify the behavior of each ADT operation.

An axiom is actually an invariant—a true statement—for an ADT operation. For example, you are familiar with axioms for algebraic operations; in particular, you know the following rules for multiplication:

$$(a \times b) \times c = a \times (b \times c)$$

$$a \times b = b \times a$$

$$a \times 1 = a$$

$$a \times 0 = 0$$

An axiom is a mathematical rule

Axioms for multiplication

These rules, or axioms, are true for any numeric values of  $a$ ,  $b$ , and  $c$ , and they describe the behavior of the multiplication operator  $\times$ .

Axioms specify the behavior of an ADT

In a similar fashion, you can write a set of axioms that completely describes the behavior of the operations for the ADT list. For example,

*A newly created list is empty*

is an axiom since it is true for all newly created lists. You can state this axiom succinctly in terms of the operations of the ADT list as follows:

`(aList.createList()).isEmpty() is true`

That is, the list `aList` is empty.

The statement

*If you insert an item  $x$  into the  $i^{\text{th}}$  position of an ADT list, retrieving the  $i^{\text{th}}$  item will result in  $x$*

is true for all lists, and so it is an axiom. You can state this axiom in terms of the operations of the ADT list as follows:<sup>3</sup>

`(aList.add(i, x)).get(i) = x`

That is, `get` retrieves from position  $i$  of list `aList` the item  $x$  that `add` has put there.

The following axioms formally define the ADT list:

#### KEY CONCEPTS

#### Axioms for the ADT List

1. `(aList.createList()).size() = 0`
2. `(aList.add(i, x)).size() = aList.size() + 1`
3. `(aList.remove(i)).size() = aList.size() - 1`
4. `(aList.createList()).isEmpty() = true`
5. `(aList.add(i, item)).isEmpty() = false`
6. `(aList.createList()).remove(i) = error`
7. `(aList.add(i, x)).remove(i) = aList`
8. `(aList.createList()).get(i) = error`
9. `(aList.add(i, x)).get(i) = x`
10. `aList.get(i) = (aList.add(i, x)).get(i+1)`
11. `aList.get(i+1) = (aList.remove(i)).get(i)`

3. The `=` notation within these axioms denotes algebraic equality.

A set of axioms does not make the pre- and postconditions for an ADT's operations unnecessary. For example, the previous axioms do not describe *add*'s behavior when you try to insert an item into position 50 of a list of 2 items. One way to handle this situation is to include the restriction

```
0 <= index <= size()
```

in *add*'s precondition. Another way—which you will see when we implement the ADT list later in this chapter—does not restrict *index*, but rather throws an exception if *index* is outside the previous range. Thus, you need both a set of axioms and a set of pre- and postconditions to define the behavior of an ADT's operations completely.

You can use axioms to determine the outcome of a sequence of ADT operations. For example, if *aList* is a list of characters, how does the sequence of operations

```
aList.add(0, b)
aList.add(0, a)
```

Use axioms to determine the effect of a sequence of ADT operations

affect *aList*? We will show that *a* is the first item in this list and that *b* is the second item by using *get* to retrieve these items.

You can write the previous sequence of operations in another way as

```
(aList.add(0, b)).add(0, a)
```

or

```
tempList.add(0, a)
```

where *tempList* represents *aList.add(0, b)*. Now retrieve the first and second items in the list *tempList.add(0, a)*, as follows:

```
(tempList.add(0, a)).get(0) = a by axiom 9
```

and

|                                    |                                  |
|------------------------------------|----------------------------------|
| <i>(tempList.add(0, a)).get(1)</i> |                                  |
| = <i>tempList.get(0)</i>           | by axiom 10                      |
| = <i>(aList.add(0, b)).get(0)</i>  | by definition of <i>tempList</i> |
| = <i>b</i>                         | by axiom 9                       |

Thus, *a* is the first item in the list and *b* is the second item.

Axioms are treated further in exercises in the rest of the book.

### 4.3 Implementing ADTs

The previous sections emphasized the specification of an abstract data type. When you design an ADT, you concentrate on what its operations do, but you ignore how you will implement them. The result should be a set of clearly specified ADT operations.

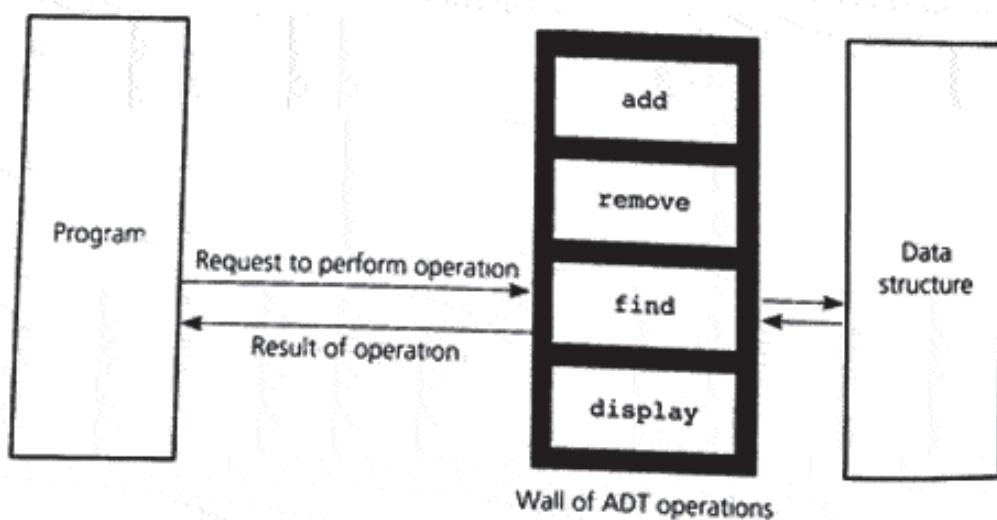
Data structures are part of an ADT's implementation

How do you implement an ADT once its operations are clearly specified? That is, how do you store the ADT's data and carry out its operations? Earlier in this chapter you learned that when implementing an ADT, you choose data structures to represent the ADT's data. Thus, your first reaction to the implementation question might be to choose a data structure and then to write methods that access it in accordance with the ADT's operations. Although this point of view is not incorrect, hopefully you have learned not to jump right into code. In general, you should refine an ADT through successive levels of abstraction. That is, you should use a top-down approach to designing an algorithm for each of the ADT operations. You can view each of the successively more concrete descriptions of the ADT as implementing its more abstract predecessors. The refinement process stops when you reach data structures that are available in your programming language. The more primitive your language, the more levels of implementation you will require.

The choices that you make at each level of the implementation can affect its efficiency. For now, our analyses will be intuitive, but Chapter 10 will introduce you to quantitative techniques that you can use to weigh the trade-offs involved.

Recall that the program that uses the ADT should see only a wall of available operations that act on data. Figure 4-8 illustrates this wall once again. Both the data structure that you choose to contain the data and the implementations of the ADT operations are hidden behind the wall. By now, you should realize the advantage of this wall.

In a non-object-oriented implementation, both the data structure and the ADT operations are distinct pieces. The client agrees to honor the wall by



**FIGURE 4-8**

ADT operations provide access to a data structure

using only the ADT operations to access the data structure. Unfortunately, the data structure is hidden only if the client does not look over the wall! Thus, the client can violate the wall—either intentionally or accidentally—by accessing the data structure directly, as Figure 4-9 illustrates. Why is such an action undesirable? Later, this chapter will use an array `items` to store an ADT list's items. In a program that uses such a list, you might, for example, accidentally access the first item in the list by writing

```
firstItem = items[0];
```

instead of by invoking the list operation `get`. If you changed to another implementation of the list, your program would be incorrect. To correct your program, you would need to locate and change all occurrences of `items[0]`—but first you would have to realize that `items[0]` is in error!

Object-oriented languages such as Java provide a way for you to enforce the wall of an ADT, thereby preventing access of the data structure in any way other than by using the ADT operations. We will spend some time now exploring this aspect of Java by discussing classes, interfaces, and exceptions.

## Java Classes Revisited

Recall from Chapter 2 that object-oriented programming, or OOP, views a program not as a sequence of actions but as a collection of components called objects. Encapsulation—one of OOP's three fundamental principles<sup>4</sup>—enables

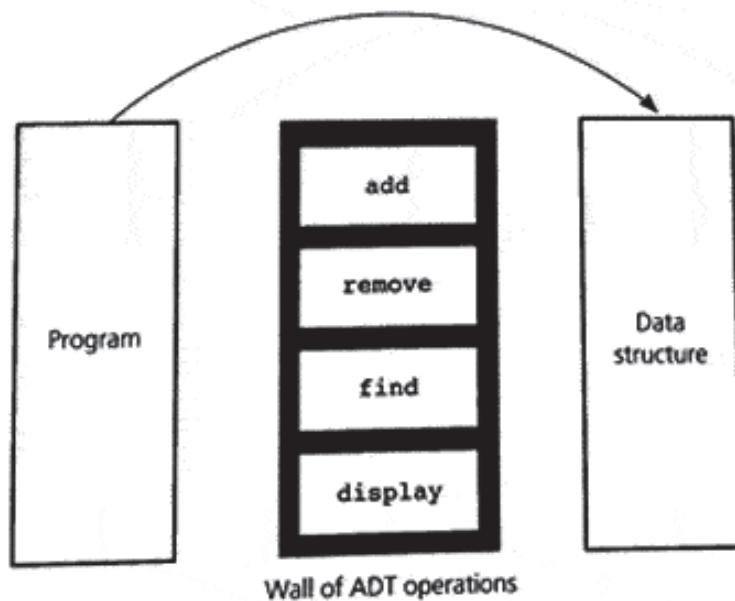


FIGURE 4-9

Violating the wall of ADT operations

4. The other principles are inheritance and polymorphism, which Chapter 9 will discuss.

Encapsulation hides implementation details

A Java class defines a new data type

An object is an instance of a class

A class's data fields should be private

you to enforce the walls of an ADT. It is, therefore, essential to an ADT's implementation and our main focus here.

Encapsulation combines an ADT's data with its operations—called **methods**—to form an **object**. Rather than thinking of the many components of the ADT in Figure 4-8, you can think at a higher level of abstraction when you consider the object in Figure 4-10 because it is a single entity. The object hides its inner detail from the programmer who uses it. Thus, an ADT's operations become an object's behaviors.

We could use a ball as an example of an object. Because thinking of a basketball, volleyball, tennis ball, or soccer ball probably suggests images of the game rather than the object itself, let's abstract the notion of a ball by picturing a sphere. A sphere of a given radius has attributes such as volume and surface area. A sphere as an object should be able to report its radius, volume, surface area, and so on. That is, the sphere object has methods that return such values. This section will develop the notion of a sphere as an object. Later, in Chapter 9, you will see how to derive a ball from a sphere.

In Java, a class is a new data type whose instances are objects. A class contains **data fields** and **methods**, collectively known as **class members**. Methods typically act on the data fields. By default, all members in a class are **private**—they are not directly accessible by your program—unless you designate them as **public**. The implementations of a class's methods, however, can use any private members of that class.

You should almost always declare a class's data fields as private. Typically, as was mentioned in Chapter 2, you provide methods—such as `setDataField` and `getDataField`—to access the data fields. In this way, you control how and whether the rest of the program can access the data fields. This design principle should lead to programs that not only are easier to debug, but also have fewer logical errors from the beginning.

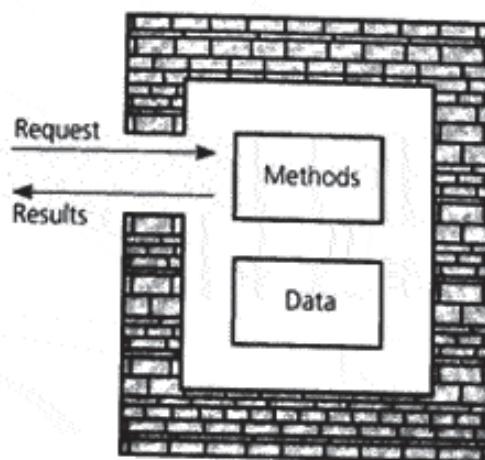


FIGURE 4-10

An object's data and methods are encapsulated

You should also distinguish between a class's data fields and any local variables that the implementation of a method requires. In Java, data fields are variables that are shared by all of the methods in the class. Data fields have initial default values, based on their type, and thus do not need to be explicitly initialized. But it is considered to be good programming practice to explicitly initialize data fields in the constructors when necessary. Local variables are used only within a single method and must be initialized explicitly before they are used.

The ADTs that you saw earlier had an operation for their creation. Classes have such methods, called **constructors**. A constructor creates and initializes new instances of a class. A typical class has several constructors. A constructor has the same name as the class. Constructors have no return type—not even `void`—and cannot use `return` to return a value. They can, however, have parameters. We will discuss constructors in more detail shortly, after we look at an example of a class definition.

Java has a garbage collection mechanism to destroy objects that a program no longer needs. When a program no longer references an object, the Java runtime environment marks it for garbage collection. Periodically, the Java runtime environment executes a method that returns the memory used by these marked objects to the system for future use. Sometimes when an object is destroyed, other tasks beyond memory deallocation are necessary. In these cases, you define a `finalize` method for the object.

A constructor creates and initializes an object

Java destroys objects that a program no longer references

## Java Interfaces

Often it is convenient to be able to specify a set of methods that you might want to provide in many different classes. One way to do this is to define a superclass that contains these methods and then use inheritance to create the different classes that need to provide those methods. This could pose a problem, however, if the subclass also needs to extend another superclass. Java allows only one class to appear in the `extends` clause.

To address this situation, Java provides interfaces. An interface provides a way to specify methods and constants, but supplies no implementation details for the methods. Interfaces enable you to specify some desired common behavior that may be useful over many different types of objects. You can then design a method to work with a variety of object types that exhibit this common behavior by specifying the interface as the parameter type for the method, instead of a class. This allows the method to use the common behavior in its implementation, as long as the arguments to the method have implemented the interface.

The Java API has many predefined interfaces. For example, `java.util.Collection` is an interface that provides methods for managing a collection of objects. Here are two of the methods specified in the `Collection` interface:

```
public boolean add(Object o);
public boolean contains(Object o);
```

An interface specifies methods and constants but supplies no implementations

If you want to have your class provide the methods in this interface, you must indicate your intent to implement the interface by including an *implements* clause in your class definition and provide implementations of the methods:

A class that implements an interface

```
public class CardCollection
 implements java.util.Collection {
 ...
 public boolean add(Object o) {
 // implementation of add method
 } // end add

 public boolean contains(Object o) {
 // implementation of contains method
 } // end contains

 // and so on...
} // end CardCollection
```

Suppose there is a *print(Collection c)* method. Instances of *CardCollection* are now eligible to be used as arguments to this method since *CardCollection* implements the interface *Collection*.

To define your own interface, you use the keyword *interface* instead of *class*, and you provide only method specifications and constants in the interface definition. For example,

An example of an interface

```
public interface MyInterface {
 public final int f1 = 0;
 public void method1();
 public int method2(int a, int b);
} // end MyInterface
```

This defines an interface *MyInterface* that has one constant *f1* and two methods *method1* and *method2*. Note that the name of the interface ends with *Interface*. This is another coding convention that we will use throughout this text.

Interfaces will be used to specify the ADTs that are developed in this text. The implementation of the ADT *list* presented later in this chapter will provide a more complete example of a user-defined interface and how it can be used.

**Object Comparison.** Earlier we saw the use of the *equals* method to determine the equality of two objects. Sometimes it is useful to also be able to determine not only the equality of objects, but if one object is greater or less than another object.

When comparing objects in this way, the determination of what makes one object "less" than another object can be specified by implementing the

`java.lang.Comparable` interface. This interface contains one method, `compareTo`, that returns a negative integer, zero, or a positive integer if the current object is less than, equal to, or greater than the specified object. The following example showing the `SimpleSphere` class implementing the `Comparable` interface:

```
class SimpleSphere implements java.lang.Comparable<Object> {

 ... methods as before

 public int compareTo(Object rhs) {
 //Compares rhs object with this object
 Precondition: The object rhs should be a Sphere object
 Postcondition: If this sphere has the same radius as the
 rhs sphere, returns zero. If this sphere has a larger
 radius than the rhs sphere, a positive integer is
 returned. If this sphere has a smaller radius than the
 rhs sphere, a negative integer is returned.
 Throws: ClassCastException if the rhs object is not a
 Sphere object.

 throws ClassCastException if rhs cannot be cast to
 Sphere
 Sphere other = (Sphere)rhs;

 if (radius == other.radius) {
 return 0; //Equal
 else if (radius < other.radius) {
 return -1;
 else // radius > other.radius
 return 1;
 }
 }

 end compareTo
}

end SimpleSphere class
```

In this example, the criterion for comparison is based solely in the radius of the spheres. Spheres with a smaller radius value are considered “less than” spheres with larger radii. Sometimes, the criterion used to compare objects depends on more than one value. For example, suppose you want to compare the names of people, each consisting of a first name and a last name. Simply examining the last name might not work unless you have two people with the same last name, then you would need to compare the two first names. The following example defines a `FullName` class, and demonstrates how such a comparison could be defined:

```
public class FullName implements java.lang.Comparable<Object> {
 private String firstName;
 private String lastName;
```

```

public FullName(String first, String last) {
 firstName = first;
 lastName = last;
} // end constructor

public int compareTo(Object rhs) {
 // Precondition: The object rhs should be a FullName object
 // Postcondition: Returns 0 if all fields match
 // if lastName equals rhs.lastName and
 // firstName is greater than rhs.firstName.
 // Returns -1 if lastName is less than rhs.lastName or
 // if lastName equals rhs.lastName and
 // firstName is less than rhs.firstName
 // Throws: ClassCastException if the rhs object is not a Full
 // object.

 // Throws ClassCastException if rhs cannot be cast to Full
 FullName other = (FullName)rhs;

 if (lastName.compareTo(((FullName)other).lastName)==0){
 return firstName.compareTo(((FullName)other).firstName);
 }
 else {
 return lastName.compareTo(((FullName)other).lastName);
 } // end if
} // end compareTo
} // end class FullName

```

## Java Packages

Java packages provide a way to group related classes together. To create a package, you place a *package* statement at the top of each class file that is part of the package. For example, Java source files that are part of a drawing package would include the following line at the top of the file (Java package names usually begin with a lowercase letter):

To include a class in a package, begin the class's source file with a *package* statement

Place the files that contain a package's classes in the same directory

```
package drawingPackage;
```

Just as you must use the same name for both a Java class and the file that contains the class, you must use the same name for a package and the directory that contains all the classes in the package. Thus, the source files for the drawing package must be contained in a directory called *drawingPackage*.

When declaring classes within a package, the keyword *public* must appear in front of the *class* keyword to make the class available to clients of the package.

the drawing package contains a class `Palette`, the source file for `Palette` begins as follows:

```
package drawingPackage;
public class Palette {
 ...
```

Omitting the keyword `public` will make the class available only to other classes within the package. Sometimes, such restricted access is desirable.

When a class is publicly available within a package, it can also be used as a superclass for any new class, even those appearing in other packages. This is actually done often within the Java API. For example, the exception class `java.lang.RuntimeException` is the superclass for the class `java.util.NoSuchElementException`.

Access to a package's classes can be public or restricted

A package can contain other packages as well. If `shapePackage` is a package in `drawingPackage`, the directory for `shapePackage` must be a subdirectory of the `drawingPackage` directory. The package name consists of the hierarchy of package names, separated by periods. For example, if the Java source file for the class `Sphere` is part of `shapePackage`, the following line must appear at the top of the file `Sphere.java`:

```
package drawingPackage.shapePackage;
```

You already use packages in your programs when you place an `import` statement in your code. When you write a statement such as

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

Using a package

you indicate to the compiler that you want to use classes in the package `java.io`. The `*` is a way to indicate that you might use any class in `java.io`, but if you know the specific class from the `java.io` package that you plan to use, you replace the `*` with the name of that class. For example, the statement

```
import java.io.DataStream;
```

indicates that you will use the class `DataStream` from the package `java.io`.

If you omit the package declaration from the source file for a class, the class is added to a default, unnamed package. If all the classes in a group are declared this way, they are all considered to be within this same unnamed package and hence do not require an `import` statement. But if you are developing a package, and you want to use a class that is contained in the unnamed package, you will need to import the class. In this case, since the package has no name, the class name itself is sufficient in the `import` statement.

## An Array-Based Implementation of the ADT List

We will now implement the ADT list as a class. Recall that the ADT list operations are

```
+createList()
+isEmpty():boolean
+size():integer
+add(in index:integer, in newItem:ListItemType)
+remove(in index:integer)
+removeAll()
+get(in index:integer):ListItemType
```

You need to represent the items in the ADT list and its length. Your first thought is probably to store the list's items in an array *items*. In fact, you might believe that the list is simply a fancy name for an array. This belief is not quite true, however. An **array-based implementation** is a natural choice because both an array and a list identify their items by number. However, the ADT list has operations such as *removeAll* that an array does not. In the next chapter you will see another implementation of the ADT list that does not use an array.

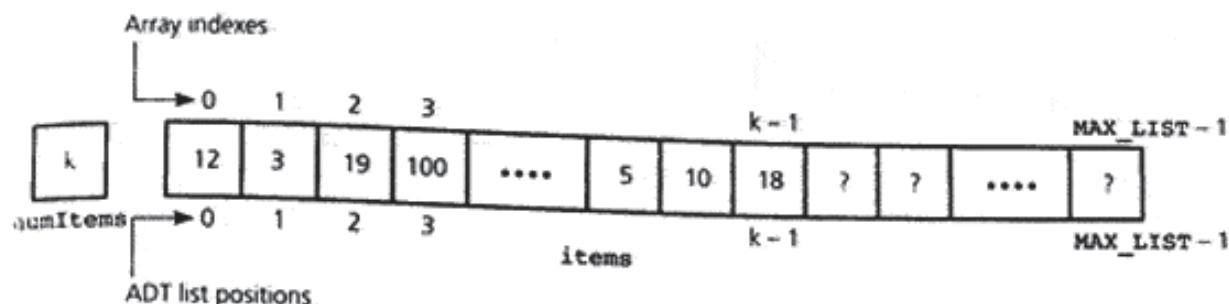
In any case, you can store a list's  $k^{\text{th}}$  item in *items*[*k*]. How much of the array will the list occupy? Possibly all of it, but probably not. That is, you need to keep track of the array elements that you have assigned to the list and those that are available for use in the future. The maximum length of the array—**physical size**—is a known, fixed value such as *MAX\_LIST*. You can keep track of the current number of items in the list—that is, the list's length or **logical size**—in a variable *numItems*. An obvious benefit of this approach is that implementing the operation *size* will be easy. Thus, we could use the following statements to implement a list of integers:

```
private final int MAX_LIST = 100; // max length of list
private int items[MAX_LIST]; // array of list items
private int numItems; // length of list
```

Shift array elements  
to insert an item

Figure 4-11 illustrates the data fields for an array-based implementation of an ADT list of integers. To insert a new item at a given position in the array of list items, you must shift to the right the items from this position on, and insert the new item in the newly created opening. Figure 4-12 depicts this insertion.

Now consider how to delete an item from the list. You could blank it out, but this strategy can lead to gaps in the array, as Figure 4-13a illustrates. An array that is full of gaps has three significant problems:

**FIGURE 4-11**

An array-based implementation of the ADT list

- `numItems - 1` is no longer the index of the last item in the array. You need another variable, `lastPosition`, to contain this index.
- Because the items are spread out, the method `get` might have to look at every cell of the array even when only a few items are present.
- When `items[MAX_LIST - 1]` is occupied, the list could appear full, even when fewer than `MAX_LIST` items are present.

Thus, what you really need to do is shift the elements of the array to fill the gap left by the deleted item, as shown in Figure 4-13b.

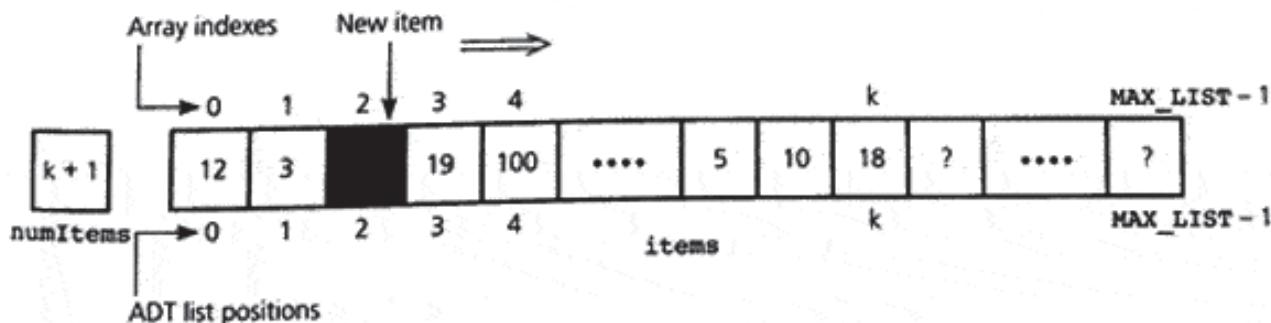
You should implement each ADT operation as a method of a class. Each operation will require access to both the array `items` and the list's length `numItems`, so make `items` and `numItems` data fields of the class. To hide `items` and `numItems` from the clients of the class, make these data fields private.

If one of the operations is provided an index value that is out of range, an exception should be thrown. Here is a definition of an exception that can be used for an out-of-bounds list index called `ListIndexOutOfBoundsException`. It is based upon the more general `IndexOutOfBoundsException` from the Java API:

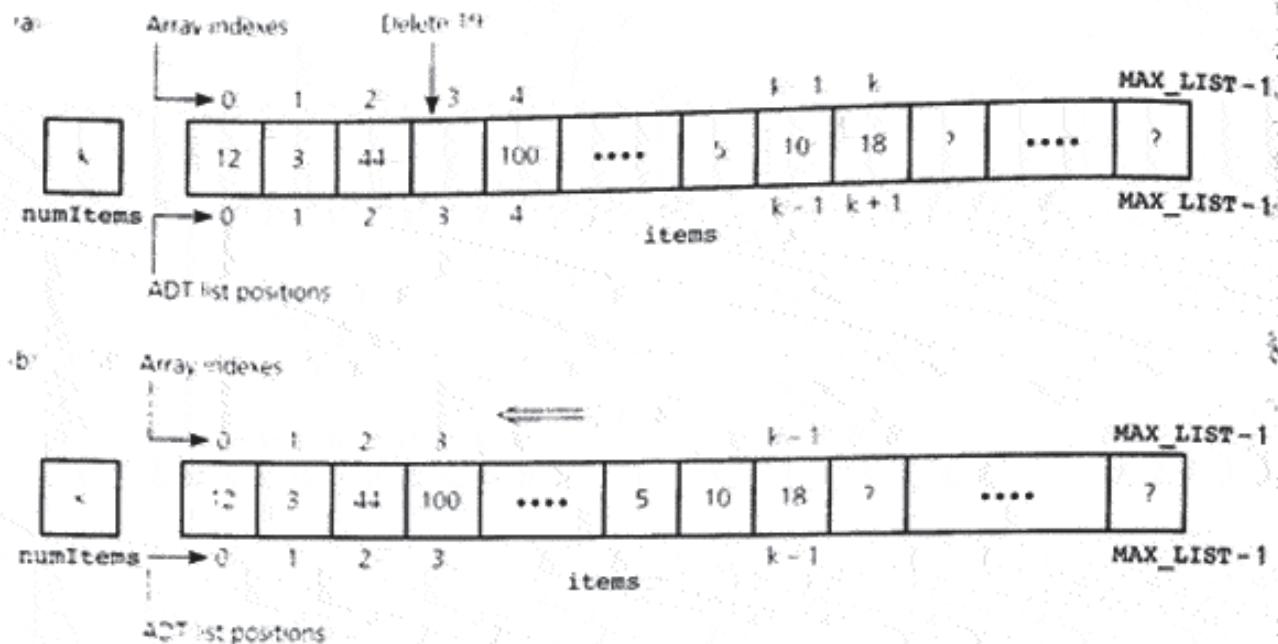
Shift array elements to delete an item

Implement the ADT list as a class

`items` and `numItems` are private data fields

**FIGURE 4-12**

Shifting items for insertion at position 3

**FIGURE 4-13**

(a) Deletion causes a gap. (b) fill gap by shifting

```
public class ListIndexOutOfBoundsException
 extends IndexOutOfBoundsException {
 public ListIndexOutOfBoundsException(String s) {
 super(s);
 } // end constructor
} // end ListIndexOutOfBoundsException
```

Also, the exception *ListException* is needed when the array storing the list becomes full. Here is the exception *ListException*:

```
public class ListException extends RuntimeException {
 public ListException(String s) {
 super(s);
 } // end constructor
} // end ListException
```

The following interface *IntegerListInterface* provides the specifications for the list operations. The ADT operation *createList* does not appear in the interface because it will be implemented as a constructor.

Interface for a list  
of integers

```
// ****
// Interface IntegerListInterface for the ADT list.
// ****
```

```
public interface IntegerListInterface {
 public boolean isEmpty();
 // Determines whether a list is empty.
 // Precondition: None.
 // Postcondition: Returns true if the list is empty,
 // otherwise returns false.
 // Throws: None.

 public int size();
 // Determines the length of a list.
 // Precondition: None.
 // Postcondition: Returns the number of items that are
 // currently in the list.
 // Throws: None.

 public void removeAll();
 // Deletes all the items from the list.
 // Precondition: None.
 // Postcondition: The list is empty.
 // Throws: None.

 public void add(int index, int item)
 throws ListIndexOutOfBoundsException,
 ListException;
 // Adds an item to the list at position index.
 // Precondition: index indicates the position at which
 // the item should be inserted in the list.
 // Postcondition: If insertion is successful, item is
 // at position index in the list, and other items are
 // renumbered accordingly.
 // Throws: ListIndexOutOfBoundsException if index < 0 or
 // index > size().
 // Throws: ListException if item cannot be placed on
 // the list.

 public int get(int index) throws
 ListIndexOutOfBoundsException;
 // Retrieves a list item by position.
 // Precondition: index is the number of the item to be
 // retrieved.
 // Postcondition: If 0 <= index < size(), the item at
 // position index in the list is returned.
 // Throws: ListIndexOutOfBoundsException if index < 0 or
 // index > size()-1.

 public void remove(int index)
 throws ListIndexOutOfBoundsException;
```

```

 // Deletes an item from the list at a given position.
 // Precondition: index indicates where the deletion
 // should occur.
 // Postcondition: If 0 <= index < size(), the item at
 // position index in the list is deleted, and other items
 // are renumbered accordingly.
 // Throws: ListIndexOutOfBoundsException if index < 0 or
 // index > size()-1.

} // end IntegerListInterface

```

**Use *Object* as the type of list elements**

The notion of a list and of the operations that you perform on the list are really independent of the type of items that are stored in the list. The definition just given is very limiting in that it will support only a list of integers. If you use the class *Object* as the type for the list's elements, the specification will be far more flexible. Every class in Java is ultimately derived from the class *Object* through inheritance. This means that any class created in Java could be used as an item in the list class that implements the new list interface.

What happens if you want to have a list of integers? If the list item type is *Object*, you can no longer use the primitive type *int* as the item type, since *int* is not derived from the class *Object*. In cases where you want a list to contain items of a primitive type, you will need to use a corresponding wrapper class from the Java API. For example, instead of using items of type *int* in the list, the items would be of the type *java.lang.Integer*, a subclass of *Object*.

Here is a revised version, called *ListInterface*, that uses the class *Object* for the list elements. The comments, which are the same as those in *IntegerListInterface*, are left out to save space:

```

// ****
// Interface ListInterface for the ADT list.
// ****
public interface ListInterface {
 public boolean isEmpty();
 public int size();
 public void add(int index, Object item)
 throws ListIndexOutOfBoundsException,
 ListException;
 public Object get(int index)
 throws ListIndexOutOfBoundsException;
 public void remove(int index)
 throws ListIndexOutOfBoundsException;
 public void removeAll();
} // end ListInterface

```

**Interface for a list of objects**

The following class implements the interface *ListInterface*, using arr

```
// ****
// Array-based implementation of the ADT list.
// ****
public class ListArrayBased implements ListInterface { Implementation file
 private static final int MAX_LIST = 50;
 private Object items[]; // an array of list items
 private int numItems; // number of items in list

 public ListArrayBased() {
 items = new Object[MAX_LIST];
 numItems = 0;
 } // end default constructor

 public boolean isEmpty() {
 return (numItems == 0);
 } // end isEmpty

 public int size() {
 return numItems;
 } // end size

 public void removeAll() {
 // Creates a new array; marks old array for
 // garbage collection.
 items = new Object[MAX_LIST];
 numItems = 0;
 } // end removeAll

 public void add(int index, Object item)
 throws ListIndexOutOfBoundsException {
 if (numItems > MAX_LIST) {
 throw new ListException("ListException on add");
 } // end if
 if (index >= 0 && index <= numItems) {
 // make room for new element by shifting all items at
 // positions >= index toward the end of the
 // list (no shift if index == numItems+1)
 for (int pos = numItems; pos >= index; pos--) {
 items[pos+1] = items[pos];
 } // end for
 // insert new item
 items[index] = item;
 numItems++;
 }
 else { // index out of range
 throw new ListIndexOutOfBoundsException(
 "ListIndexOutOfBoundsException on add");
 }
 }
}
```

```

 } // end if
 } //end add

 public Object get(int index)
 throws ListIndexOutOfBoundsException {
 if (index >= 0 && index < numItems) {
 return items[index];
 }
 else { // index out of range
 throw new ListIndexOutOfBoundsException(
 "ListIndexOutOfBoundsException on get");
 } // end if
 } // end get

 public void remove(int index)
 throws ListIndexOutOfBoundsException {
 if (index >= 0 && index < numItems) {
 // delete item by shifting all items at
 // positions > index toward the beginning of the list
 // (no shift if index == size)
 for (int pos = index+1; pos <= size(); pos++) {
 items[pos-1] = items[pos];
 } // end for
 numItems--;
 }
 else { // index out of range
 throw new ListIndexOutOfBoundsException(
 "ListIndexOutOfBoundsException on remove");
 } // end if
 } // end remove

} // end ListArrayBased

```

The following program segment demonstrates the use of `ListArrayBased`:

```

static public void main(String args[]) {
 ...
 ListArrayBased alist = new ListArrayBased();
 String dataItem;
 int size;
 size = 5;
 dataItem = "Gathgym";
 ...
}

```

Note that references within this program such as `aList.numItems` and `aList.items[4]` would be illegal because `numItems` and `items` are private members of the class.

In summary, to implement an ADT, given implementation-independent specifications of the ADT operations, you first must choose a data structure to contain the data. Next, you define and implement a class within a Java source file. The ADT operations are public methods within the class, and the ADT data is represented as data fields that are typically private. You then implement the class's methods within an implementation file. The program that uses the class will be able to access the data only by using the ADT operations.

A client of the class cannot access the class's private members directly

## Summary

1. Data abstraction is a technique for controlling the interaction between a program and its data structures. It builds walls around a program's data structures, just as other aspects of modularity build walls around a program's algorithms. Such walls make programs easier to design, implement, read, and modify.
2. The specification of a set of data-management operations together with the data values upon which they operate define an abstract data type (ADT).
3. The formal mathematical study of ADTs uses systems of axioms to specify the behavior of ADT operations.
4. Only after you have fully defined an ADT should you think about how to implement it. The proper choice of a data structure to implement an ADT depends both on the details of the ADT operations and on the context in which you will use the operations.
5. Even after you have selected a data structure as an implementation for an ADT, the remainder of the program should not depend on your particular choice. That is, you should access the data structure by using only the ADT operations. Thus, you hide the implementation behind a wall of ADT operations. To enforce the wall within Java, you define the ADT as a class, thus hiding the ADT's implementation from the program that uses the ADT.
6. An object encapsulates both data and operations on that data. In Java, objects are instances of a class, which is a programmer-defined data type.

## Cautions

1. After you design a class, try writing some code that uses your class before you commit to your design. Not only will you see whether your design works for the problem at hand, but you will also test your understanding of your own design and check the comments that document your specifications.
2. When you implement a class, you might discover problems with either your class design or your specifications. If these problems occur, change your design and

- specifications, try using the class again, and continue implementing. These comments are consistent with the discussion of software life cycle in Chapter 1.
3. A program should not depend upon the particular implementations of its ADTs. By using a class to implement an ADT, you encapsulate the ADT's data and operations. In this way, you can hide implementation details from the program that uses the ADT. In particular, by making the class's data fields private, you can change the class's implementation without affecting the client.
  4. By making a class's data fields private, you make it easier to locate errors in a program's logic. An ADT—and hence a class—is responsible for maintaining its data. If an error occurs, you look at the class's implementation for the source of the error. If the client could manipulate this data directly because the data was public, you would not know where to look for errors.
  5. Variables that are local to a method's implementation should not be data fields of the class.
  6. An array-based implementation of an ADT restricts the number of items that you can store. Chapter 5 will discuss a way to avoid this problem.

### Self-Test Exercises

1. What is the significance of "wall" and "contract"? Why do these notions help you to become a better problem solver?
2. Write a pseudocode method `swap(aList, i, j)` that interchanges the items currently in positions *i* and *j* of a list. Define the method in terms of the operations of the ADT list, so that it is independent of any particular implementation of the list. Assume that the list, in fact, has items at positions *i* and *j*. What impact does this assumption have on your solution? (See Exercise 2.)
3. What grocery list results from the following sequence of ADT list operations?

```
aList.createList()
aList.add(0, butter)
aList.add(1, eggs)
aList.add(0, cereal)
aList.add(1, milk)
aList.add(0, coffee)
aList.add(1, bread)
```

4. Write specifications for a list whose insertion, deletion, and retrieval operations are at the beginning of the list.
5. In mathematics, a set is a group of distinct items. Specify an ADT *set* that includes operations such as equality, subset, union, and intersection. Can you think of any other operations?
6. Write a pseudocode method that creates a sorted list `sortedList` from the list `aList` by using the operations of the ADTs list and sorted list.

7. The specifications of the ADTs list and sorted list do not mention the case in which two or more items have the same value. Are these specifications sufficient to cover this case, or must they be revised?
8. Specify operations that are a part of the ADT character string. Include typical operations such as length computation and concatenation (appending one string to another).

## Exercises

1. Consider an ADT list of integers. Write a method that computes the maximum of the integers in the list *aList*. The definition of your method should be independent of the list's implementation.
2. Implement the method *swap*, as described in Self-Test Exercise 2, but remove the assumption that the *i*<sup>th</sup> and *j*<sup>th</sup> items in the list exist. Throw an exception *ListIndexOutOfBoundsException* if *i* or *j* is out of range.
3. Use the method *swap* that you wrote in Exercise 2 to write a method that reverses the order of the items in a list *aList*.
4. The section "The ADT List" describes the methods *displayList* and *replace*. As given in this chapter, these operations exist outside of the ADT; that is, they are not operations of the ADT list. Instead, their implementations are written in terms of the ADT list's operations.
  - a. What is an advantage and a disadvantage of the way that *displayList* and *replace* are implemented?
  - b. What is an advantage and a disadvantage of adding the operations *displayList* and *replace* to the ADT list?
5. The ADT *Bag* is a group of items, much like what you might have with a bag of groceries. Note that the items in the bag are in no particular order and that the bag may contain duplicate items. Specify operations to put an item in the bag, remove the last item put in the bag, remove a random item from the bag, check how many items are in the bag, check to see if the bag is full or empty, and completely empty the bag.
6. Design and implement an ADT that represents a credit card. The data of the ADT should include the customer name, the account number, the next due date, the reward points, and the account balance. The initialization operation should set the data to client-supplied values. Include operations for a credit card charge, a cash advance, a payment, the addition of interest to the balance, and the display of the statistics of the account.
7. Specify operations that are a part of the ADT fraction. Include typical operations such as addition, subtraction, and reduce (reduce fraction to lowest terms).
8. Suppose you want to write a program to play the card game War. Create an ADT for a card, a second ADT for a deck of cards, and a third ADT for a hand. What operations will you need on each of these ADTs to play the game of War? Note that in the game of War, you must be able to determine the higher card (Ace is high), and the winner wins all of the cards in that round and places those cards at the bottom of his or her hand. When there is a tie, a "war" is dealt with three cards face down, then the fourth face up, and again the winner wins all the cards. If there is a tie again, "war" is played again until the

tie is broken. If a player runs out of cards in his or her hand, that last card is always played face up, even if it is in the middle of a "war."

9. Write pseudocode implementations of the operations of an ADT that represents a trapezoid. Include typical operations, such as setting and retrieving the dimensions of the trapezoid, finding the area and the perimeter of the trapezoid, and displaying the statistics of the trapezoid.
10. Implement in Java the pseudocode for the ADT trapezoid that you created in Exercise 9.
11. Write a pseudocode method in terms of the ADT appointment book, described in the section "Designing an ADT," for each of the following tasks:
  - a. Change the purpose of the appointment at a given date and time.
  - b. Display all the appointments for a given date.
12. Consider the ADT polynomial—in a single variable  $x$ —whose operations include the following:

```
+degree():integer {query}
// Returns the degree of a polynomial.
+getCoefficient(in power:integer):integer
// Returns the coefficient of the x^{power} term.
+changeCoefficient(in newCoef:integer,
 in power:integer)
// Replaces the coefficient of the x^{power} term
// with newCoef.
```

For this problem, consider only polynomials whose exponents are nonnegative integers. For example,

$$p = 4x^5 + 7x^3 - x^2 + 9$$

The following examples demonstrate the ADT operations on this polynomial.

`p.degree()` is 5 (the highest power of a term with a nonzero coefficient)

`p.getCoefficient(3)` is 7 (the coefficient of the  $x^3$  term)

`p.getCoefficient(4)` is 0 (the coefficient of a missing term is implicitly 0)

`p.changeCoefficient(-3, 7)` produces the polynomial

$$p = -3x^7 + 4x^5 + 7x^3 - x^2 + 9$$

Using only the ADT operations provided, write statements to perform the following tasks:

- a. Display the constant term (the coefficient for the  $x^0$  term).
- b. Change each coefficient in the polynomial by multiplying them by 5.
- c. For a given polynomial such as  $p = -3x^7 + 4x^5 + 7x^3 - x^2 + 9$ , convert it to its derivative— $-21x^6 + 20x^4 + 21x^2 - 2x^1$ .

- 7 Design and implement an ADT for a *Playing Card* and a *Deck* of playing cards. Each *Playing Card* must keep track of its suit (heart, diamond, club, spade), rank (2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace), and value (2 through 10 are face value, Jack is 11, Queen is 12, King is 13, and Ace is 14). The *Deck* of cards is all 52 cards—assume no Jokers will be used. The *Deck* ADT should initialize the 52 cards by suit (hearts, diamonds, clubs, spades) and within each suit is 2 through Ace. Operations for the *Deck* ADT should minimally include shuffling the deck and dealing a card from the deck.
- 8 Imagine an unknown implementation of an ADT sorted list of integers. This ADT organizes its items into ascending order. Suppose that you have just read  $N$  integers into a one-dimensional array of integers called *data*. Write some Java statements that use the operations of the ADT sorted list to sort the array into ascending order.
- 9 Use the axioms for the ADT list, as given in this chapter in the section “Axioms,” to prove that the sequence of operations

```
Insert A into position 2
Insert B into position 3
Insert C into position 2
```

has the same effect on a nonempty list of characters as the sequence

```
Insert B into position 2
Insert A into position 2
Insert C into position 2
```

- 10 Define a set of axioms for the ADT sorted list and use them to prove that the sorted list of characters, which is defined by the sequence of operations

```
Create an empty sorted list
Insert S
Insert T
Insert R
Delete T
```

is exactly the same as the sorted list defined by the sequence

```
Create an empty sorted list
Insert T
Insert R
Delete T
Insert S
```

- 11 Repeat Exercise 20 in Chapter 3, using a variation of the ADT list to implement a nonrecursive version of the method  $f(n)$ .
- 12 Write pseudocode that merges two sorted lists into a new third sorted list by using only operations of the ADT sorted list.

## Programming Problems

1. Design and implement an ADT that represents a triangle. The data for the ADT should include the three sides of the triangle but could also include the triangle's three angles. This data should be declared private in the class that implements the ADT.

Include at least two initialization operations: One that provides default values for the ADT's data (in this case a 3, 4, 5 right triangle), and another that sets this data to client-supplied values. These operations are the class's constructors.

The ADT also should include operations that look at the values of the ADT's data; change the values of the ADT's data; compute the triangle's area; and determine whether the triangle is a right triangle, an equilateral triangle, or an isosceles triangle.

2. Design and implement an ADT that represents the time of day. Represent the time as hours, minutes, and seconds on a 24-hour clock. The hours, minutes, and seconds are the private data fields of the class that implements the ADT.

Include at least two initialization operations: One that provides a default value for the time (midnight, all fields zero), and another that sets the time to a client-supplied value. These operations are the class's constructors.

Include operations that set the time, increase the time by 1 second, return the number of seconds between the time and a given time, increase the present time by a number of minutes, and two operations to display the time in 12-hour and 24-hour notations.

3. Design and implement an ADT that represents a calendar date. You can represent a date's month, day, and year as integers (for example, 5/15/2011). Include operations that advance the date by one day and provide two operations to display the date by using either numbers (05/16/2011) or words for the months (May 16, 2011). As an enhancement, include the name of the day.

4. Design and implement an ADT that represents a price in U.S. currency as dollars and cents. After you complete the implementation, write a client method that computes the change due a customer who pays  $x$  for an item whose price is  $y$ .

5. Define a class for an array-based implementation of the ADT sorted list. Consider a recursive implementation for `locateIndex`. Should `sortedAdd` and `sortedRemove` call `locateIndex`?

6. Write recursive array-based implementations of the insertion, deletion, and retrieval operations for the ADTs list and sorted list.

7. Implement the ADT bag that you specified in Exercise 5 by using only arrays and simple variables.

8. Implement the ADT character string that you specified in Self-Test Exercise 8.

9. Write a program to play the card game *War*. Use the Playing Card and Game ADTs developed in Exercise 18. You will also need to create a `Player` class that stores a player's cards that keeps track of the cards in the player's hand. The game begins by evenly dividing the deck of cards between the two players, which they hold face down. The player with the higher point value wins.

cards face down, with a fourth card face up. In this case, the player whose card has a higher point value wins all 10 cards (the 2 original cards in the tie, plus the 6 face down cards, and the last 2 face up cards), and again, adds them to the bottom of his or her hand. Should a player "run out" of cards in the midst of a "war," his or her last card is the face up card. If there is a tie in a "war," then another "war" is played, with the winner taking all of the cards and adding them to the bottom of his or her hand. Play continues as long as a player has cards in his or her hand—once they run out, the other player is declared the winner.

10. Implement the ADT appointment book, described in the section "Designing an ADT." Add operations as necessary. For example, you should add operations to read and write appointments.
11. a. Implement the ADT fraction that you specified in Exercise 7. Provide operations that read, write, add, subtract, multiply, and divide fractions. The results of all arithmetic operations should be in lowest terms, so include a private method *reduceToLowestTerms*. Exercise 23 in Chapter 3 will help you with the details of this method. (Should your read and write operations call *reduceToLowestTerms*?) To simplify the determination of a fraction's sign, you should maintain the denominator of the fraction as a positive value, and keep the sign on the numerator.  
b. Specify and implement an ADT for mixed numbers, each of which contains an integer portion and a fractional portion in lowest terms. Assume the existence of the ADT fraction (see part a). Provide operations that read, write, add, subtract, multiply, and divide mixed numbers. The results of all arithmetic operations should have fractional portions that are in lowest terms. Also include an operation that converts a fraction to a mixed number.
12. Implement the recipe database as described in the section "Designing an ADT" and, in doing so, also implement the ADT's recipe and measurement. A recipe has a title, a list of ingredients with measurements, and a list of directions. Add operations as necessary. For example, you should add operations to the ADT recipe database to read, write, and scale recipes.
13. Implement a program based on the UML specification in Programming Problem 1 of Chapter 2.
14. In mathematics, a set is a group of distinct items. Design and implement (using an array) an ADT *Set* that supports the following operations:

```
+createSet()
// creates an empty set

+isEmpty():boolean {query}
// Determines whether a set is empty

+size():integer {query}
// Returns the number of elements in this set (its
// cardinality)

+add(in item:integer)
// Adds the specified element to this set if it is not already
// present

+contains(in item:integer):boolean {query}
```

```
// Determines if this set contains the specified item

+union(in other:Set):Set
// Creates a new set containing all of the elements of this
// set and the other set (no duplicates) and returns the
// resulting set

+intersection(in other:Set):Set
// Creates a new set of elements that appear in both this set
// and the other set and returns the resulting set

+removeAll()
// Removes all of the items in the set
```

## CHAPTER 5

# Linked Lists

This chapter reviews Java references and introduces you to the data structure linked list. You will see algorithms for fundamental linked list operations such as insertion and deletion. The chapter also describes several variations of the basic linked list. As you will see, you can use a linked list and its variations when implementing many of the ADTs that appear throughout the remainder of this book. The material in this chapter is thus essential to much of the presentation in the following chapters.

### 5.1 Preliminaries

- Object References
- Resizeable Arrays
- Reference-Based Linked Lists

### 5.2 Programming with Linked Lists

- Displaying the Contents of a Linked List
- Deleting a Specified Node from a Linked List
- Inserting a Node into a Specified Position of a Linked List
- A Reference-Based Implementation of the ADT List
- Comparing Array-Based and Reference-Based Implementations
- Passing a Linked List to a Method
- Processing Linked Lists Recursively

### 5.3 Variations of the Linked List

- Tail References
- Circular Linked Lists
- Dummy Head Nodes
- Doubly Linked Lists

### 5.4 Application: Maintaining an Inventory

### 5.5 The Java Collections Framework

- Genetics
- Iterators
- The Java Collection's Framework
- List Interface

- Summary
- Cautions
- Self-Test Exercises
- Exercises
- Programming Problems

## 5.1 Preliminaries

The ADT list, as described in the previous chapter, has operations to insert, delete, and retrieve items, given their positions within the list. A close examination of the array-based implementation of the ADT list reveals that an array is not always the best data structure to use to maintain a collection of data. An array has a **fixed size**—at least in most commonly used programming languages—but the ADT list can have an arbitrary length. Thus, in the strict sense, you cannot use an array to implement a list because it is certainly possible for the number of items in the list to exceed the fixed size of the array. When developing implementations for ADTs, you often are confronted with this fixed-size problem. In many contexts, you must reject an implementation that has a fixed size in favor of one that can grow dynamically.

In addition, although the most intuitive means of imposing an order on data is to sequence it physically, this approach has its disadvantages. In a physical ordering, the successor of an item  $x$  is the next data item in sequence after  $x$ , that is, the item “to the right” of  $x$ . An array orders its items physically and, as you saw in the previous chapter, when you use an array to implement a list, you must shift data when you insert or delete an item at a specified position. Shifting data can be a time-consuming process that you should avoid, if possible. What alternatives to shifting data are available?

To get a conceptual notion of a list implementation that does not involve shifting, consider Figure 5-1. This figure should help free you from the notion that the only way to maintain a given order of data is to store the data in that order. In these diagrams, each item of the list is actually *linked* to the next item. Thus, if you know where an item is, you can determine its successor, which can be anywhere physically. This flexibility not only allows you to insert and delete data items without shifting data, but it also allows you to increase the size of the list easily. If you need to insert a new item, you simply find its place in the list and set two links. Similarly, to delete an item, you find the item and change a link to bypass the item.

Because the items in this data structure are *linked* to one another, it is called a **linked list**. As you will see shortly, a *linked list is able to grow as needed*, whereas an array can hold only a fixed number of data items. In many applications, this flexibility gives a linked list a significant advantage.

Before we examine linked lists and their use in the implementation of an ADT, we will examine how Java references can be used to implement a linked list. Like many programming languages, Java allows one object to reference another, and you can use this ability to build a linked list. The next section reviews the mechanics of these references.

An item in a linked list references its successor

### Java References

Java uses references to store a variable that refers to an object.

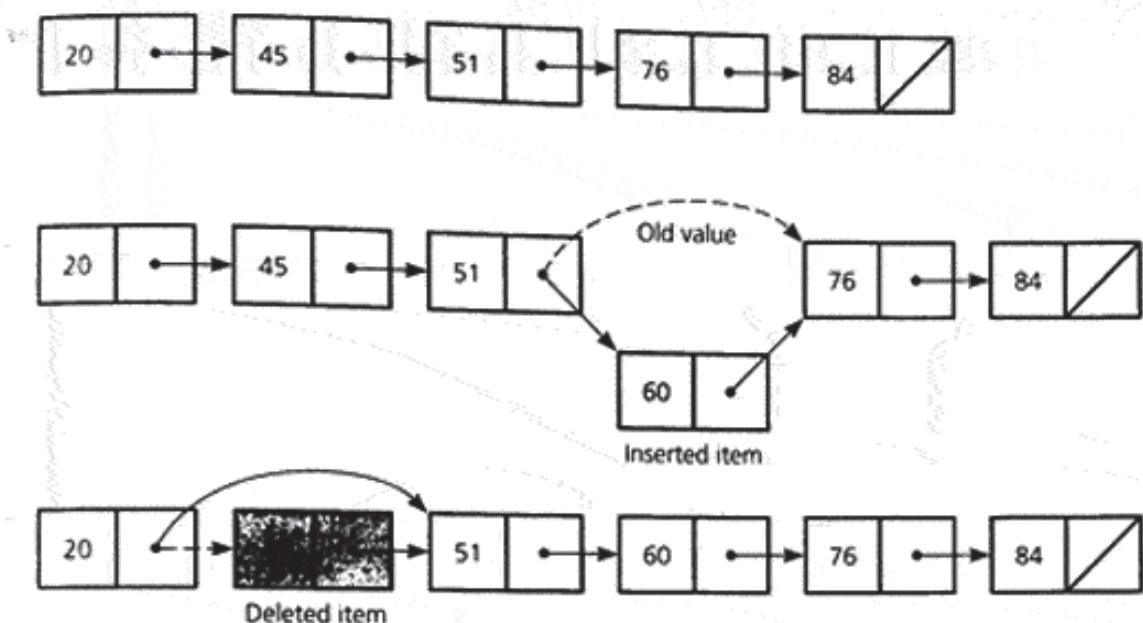


FIGURE 5-1

a) A linked list of integers; (b) insertion; (c) deletion

By using a reference to a particular object, you can locate the object and, for example, access the object's public members.

Let's look at an example using the class `java.lang.Integer` to help us visualize this scenario:

```
Integer intRef;
intRef = new Integer(5);
```

The first line declares a reference variable `intRef` that can be used to locate an `Integer` object. The second line actually instantiates an `Integer` object and assigns its location to `intRef`. Figure 5-2 illustrates that there are now two separate entities in our program: an `Integer` object and a reference variable `intRef` that provides the location of that `Integer` object.

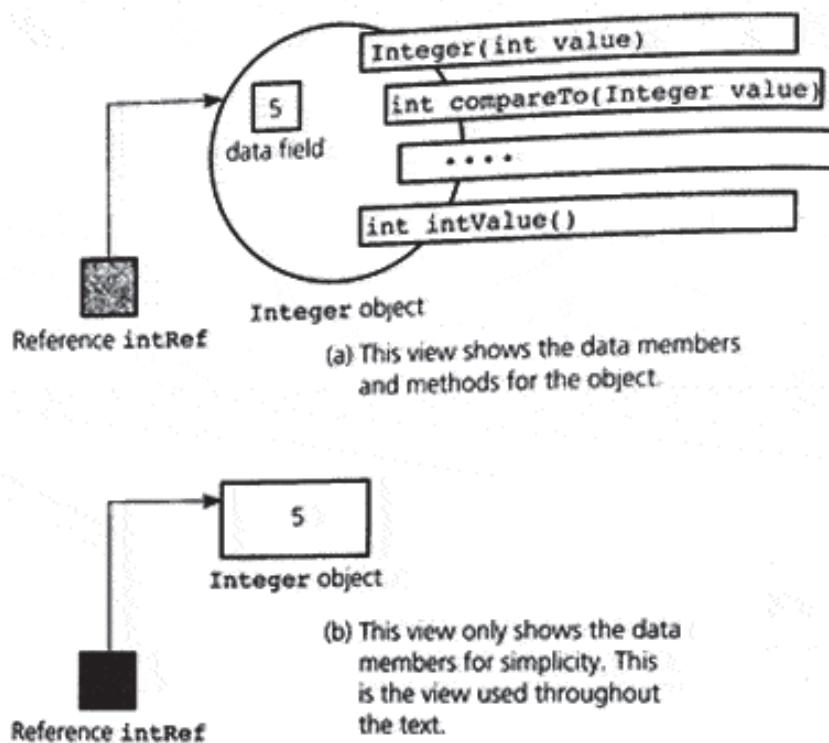
`intRef` references the newly instantiated `Integer` object

When you declare a reference variable as a data field within a class but do not instantiate an object for it in a constructor, it is initialized to `null`. You can use this constant `null` as the value of a reference to any type of object. This use indicates that the reference variable does not currently reference any object. For example, if `intRef` is declared as a data field, and you attempt to use it before you instantiate an `Integer` object for it, the exception `java.lang.NullPointerException` will be thrown at runtime. This exception indicates that you attempted to access an object by using a reference variable that contains a `null` value.

A reference variable as a data field of a class has the default value `null`

When you declare a reference variable to be local to a method, no default value is provided. For example, let `p` be declared as a reference to an `Integer` object. If you attempt to use `p` to access an object before `p` is initialized, the compiler will give you the error message "Variable `p` may not have been initialized."

A local reference variable has no default value

**FIGURE 5-2**

A reference to an `Integer` object

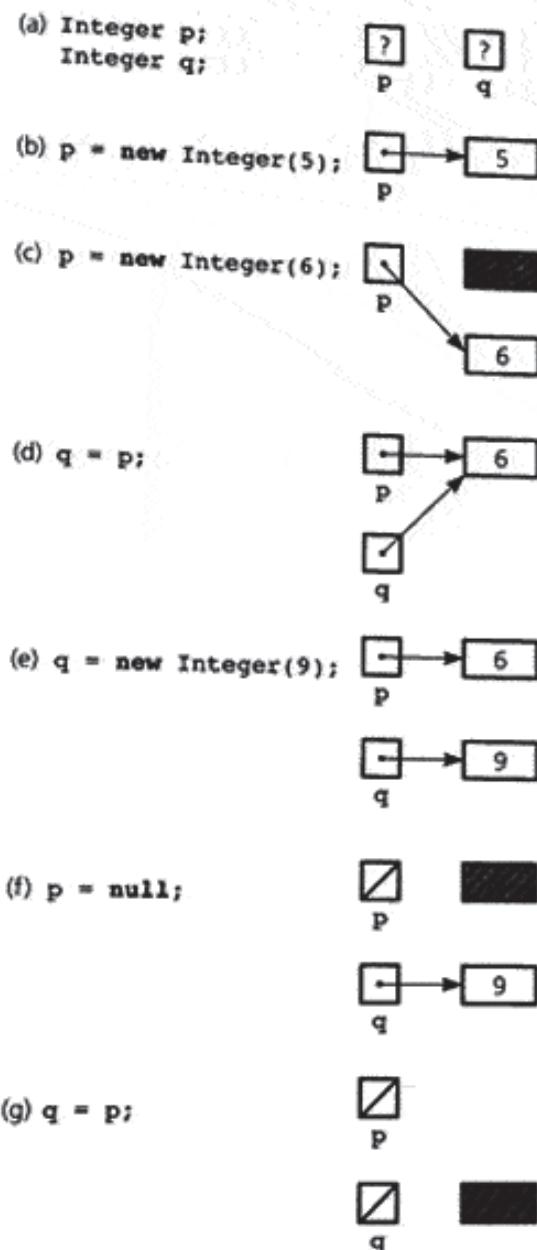
When one reference variable is assigned to another reference variable, both references then refer to the same object. For example,

```
Integer p, q;
p = new Integer(6);
q = p;
```

Figure 5-3d illustrates the result of this assignment. Now the `Integer` object has two references to it, `p` and `q`. The effect of the assignment operator is to cause the reference variable on the left side of the assignment to reference the same object as referenced by the right side of the assignment operator. Alternatively, you could let `q` reference a new object, as Figure 5-3c shows.

Suppose that you no longer need the value in a reference variable. If you do not want the reference variable to locate any particular object, explicitly assign the constant value `null`, discussed earlier, to a reference variable to indicate that it no longer references any object. When a reference to an object, the system frees the memory used by the object.

Figure 5-3e and 5-3f illustrate the effect of the `null` assignment method that is shown in Figure 5-3c.

**FIGURE 5-3**

(a) Declaring reference variables; (b) allocating an object; (c) allocating another object, with the dereferenced object marked for garbage collection; (d) assigning a reference; (e) allocating an object; (f) assigning `null` to a reference variable; (g) assigning a reference with a `null` value

When you declare an array of objects and apply the `new` operator, an array of references is actually created, not an array of objects. For example,

```
Integer[] scores = new Integer[30];
```

An array of objects is actually an array of references to the objects

creates an array of 30 references for *Integer* objects. You must instantiate actual *Integer* objects for each of the array references. For example, you might instantiate objects for the array just created as follows:

```
scores[0] = new Integer(7);
scores[1] = new Integer(9); // and so on ...
```

**Equality operators compare values of reference variables, not the objects that they reference**

When you use the equality operators (`==` and `!=`), you are actually comparing the values of the reference variables, not the objects that they reference. Suppose that you have the following class definition:

```
public class MyNumber {
 private int num;

 public MyNumber(int n) {
 num = n;
 } // end constructor

 public String toString() {
 return "My number is " + num;
 } // end toString
} // end class MyNumber
```

and you declare the following:

```
MyNumber x = new MyNumber(9);
MyNumber y = new MyNumber(9);
MyNumber z = x;
```

Although objects *x* and *y* contain the same data, the `==` operator returns *false*, since *x* and *y* refer to different objects. The expression *x == z* is *true* because the assignment statement *z = x* causes *z* to refer to the same object that *x* references. If you need to be able to compare objects field by field, you must redefine the *equals* method for the class, as discussed in Chapter 1.

Parameter passing in Java can also be discussed in terms of reference variables. When a method is called and has parameters that are objects, the reference value of the actual argument is copied to a formal parameter reference variable. During the execution of the method, the object is accessed through the formal parameter reference variable. This provides the same result as if the original reference was used. Upon completion of the method, the references stored in these formal parameters are discarded, although the objects that the parameters reference may be retained.

This method of parameter passing helps to explain why the use of the `new` operator with a formal parameter in a method can produce unexpected results. For example, suppose you have the following method *changeNumber*, which uses the *MyNumber* class:

**When you pass an object to a method as an argument, the reference to the object is copied to the method's formal parameter**

```
public void changeNumber(MyNumber n) {
 n = new MyNumber(5);
} // end changeNumber
```

and the following Java statements:

```
MyNumber x = new MyNumber(9);
changeNumber(x); // attempts to assign 5 to x
System.out.println(x);
```

The output is "My number is 9". Figure 5-4 demonstrates why this is the case. When the *changeNumber* method is invoked, the reference to object *x* is copied to the formal parameter reference *n* of *changeNumber*. During the execution of *changeNumber*, a new object is created for *n* to reference. But when the *changeNumber* method completes execution, the reference variable *n* is discarded. This causes the newly created object containing 5 to be marked for garbage collection. The value of the reference variable *x* remains unchanged; it still references the same object (containing 9) that it did before the *changeNumber* method was executed.

Note that ADT implementations and data structures that use Java references are said to be **reference based**.

---

**KEY CONCEPTS**

---

## Java Reference Variables

### 1. The declaration

```
Integer intRef;
```

statically allocates a reference variable *intRef* whose value is *null*. When a reference variable contains *null*, it does not reference anything.

### 2. *intRef* can reference an *Integer* object. The statement

```
intRef = new Integer(5);
```

dynamically allocates an *Integer* object referenced by *intRef*. (However, see item 3 on this list.)

### 3. If, for some reason, *new* cannot instantiate an object of the class represented, it may throw a *java.lang.InstantiationException* or a *java.lang.IllegalAccessException*. Thus, you can place the following statement within a *try* block to test whether memory was successfully allocated:

```
intRef = new Integer(5);
```

### 4. When the last reference to an object is removed, the object is marked for garbage collection.

---

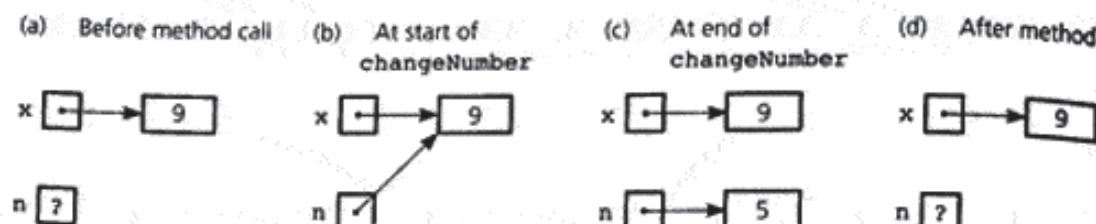


FIGURE 5-4

The value of a parameter does not affect the argument's value

## Resizable Arrays

When you declare an array in Java by using statements such as

```
final int MAX_SIZE = 50;
double[] myArray = new double[MAX_SIZE];
```

The number of references in a Java array is of a fixed size  
the Java runtime environment reserves a specific number—*MAX\_SIZE*, in this case—of references for the array. Once the array has been instantiated, it has a fixed size for the remainder of its lifetime. We have already discussed the problem this fixed-size data structure causes when your program has more than *MAX\_SIZE* items to place into the array.

You can create the illusion of a resizable array—an array that grows and shrinks as the program executes—by using an allocate and copy strategy with fixed-size arrays. If the array that you are currently using in the program reaches its capacity, you allocate a larger array and copy the references stored in the original array to the larger array. How much larger should the new array be? The increment size can be a fixed number of elements or a multiple of the current array size. The following statements demonstrate how you could accomplish this task for an array *myArray*:

```
if (capacityIncrement == 0) {
 capacity *= 2;
}
else {
 capacity += capacityIncrement;
}
// now create a new array using the updated
// capacity value
double [] newArray = new double[capacity];
// copy the contents of the original array
// to the new array
for (int i = 0; i < myArray.length; i++) {
 newArray[i] = myArray[i];
} // end for
```

Allocate a larger array

Copy the original array to the new larger array

```

 now change the reference to the original array
 to the new array
myArray = newArray;

```

In this example, *capacity* and *capacityIncrement* represent the capacity of the array and the size of the increment, respectively. Once you exceed the capacity of *myArray*, you allocate a larger array *newArray* according to the value of *capacityIncrement*. Note that if the *capacityIncrement* is zero, the array capacity doubles instead of increasing by a fixed amount. You must copy the values from the original array to the new array and then change the original array reference to reference the new array.

The classes *java.util.Vector* and *java.util.ArrayList* use a similar technique to implement a growable array of objects. The underlying implementation of *java.util.Vector* uses a fixed array of size *capacity* and has a *capacityIncrement* that you can change to suit your needs. Exercise 19 asks you to explore the *java.util.Vector* class to determine when resizing the underlying array will occur.

Subsequent discussion in this book will refer to both fixed-sized and resizable arrays. Our array-based ADT implementations will use fixed-sized arrays for simplicity. The programming problems will ask you to create array-based implementations that use resizable arrays.

## Reference-Based Linked Lists

A linked list, such as the one in Figure 5-1, contains components that are linked to one another. Each component—usually called a **node**—contains both data and a “link” to the next item. Typically, such links are Java reference variables; another possibility is mentioned at the end of this section. Although you have seen most of the mechanics of references, using references to implement a linked list is probably not yet completely clear to you. Consider now how you can set up such a linked list.

Each node of the list can be implemented as an object. For example, if you want to create a linked list of integers, you could use the following class definition, as Figure 5-5 illustrates:

```

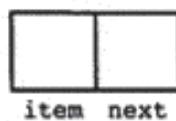
public class IntegerNode {
 public int item;
 public IntegerNode next;
} // end class IntegerNode

```

However, this type of definition violates our rule that data fields must be declared *private*, especially in public classes. But what if the *IntegerNode* class is not declared *public*, and is only used as a building block in the actual list implementation? In Java, classes can be declared as *public* using the access modifier *public*, or as package-private if no access modifier is used. Hence, it is possible to make a class available only for other classes within the same package by declaring it package-private. This effectively prevents the user of the package from gaining access to the underlying implementation. Should the

A node in a linked list is an object

A node definition that is not desirable because its data fields are public



**FIGURE 5-5**

A node

underlying implementation be changed, no code outside of the package will be affected. So, we could rewrite the *IntegerNode* class as follows:

A node for a linked list of integers

```
package IntegerList;
class IntegerNode {
 int item;
 IntegerNode next;
} // end class IntegerNode
```

and use this class as follows:

Defining a reference to a node

```
IntegerNode n1 = new IntegerNode();
IntegerNode n2 = new IntegerNode();
n1.item = 5; // set item in first node
n2.item = 9; // set item in second node
n1.next = n2; // link the nodes
```

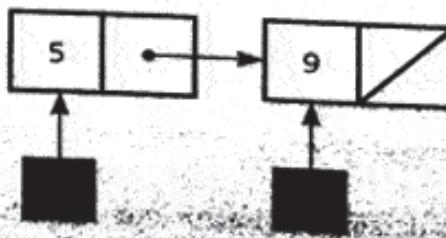
This scenario is depicted in Figure 5-6. The *item* field is initialized for each node. The *next* field for the node *n1* is then set to *n2*, which in effect links the nodes by making the first node reference the second. Exercise 8 will ask you to explore the declaration of a node class with private data fields.

We could improve this class by adding constructors, as follows:

Constructors

```
package IntegerList;
class IntegerNode {
 int item;
 IntegerNode next;

 IntegerNode(int newItem) {
 item = newItem;
 next = null;
 } // end constructor
```



```
IntegerNode(int newItem, IntegerNode nextNode) {
 item = newItem;
 next = nextNode;
} // end constructor
} // end class IntegerNode
```

This definition of a node restricts the data to a single integer field. Since we would like to have this class be as reusable as possible, it would be better to change the data field to be of type *Object*. Recall that every class in Java is ultimately derived from the class *Object* through inheritance. This means that any class created in Java could use this node definition for storing objects. Let's first examine the revised class, using objects for data:

```
package List;

class Node {
 Object item;
 Node next;

 Node(Object newItem) {
 item = newItem;
 next = null;
 } // end constructor

 Node(Object newItem, Node nextNode) {
 item = newItem;
 next = nextNode;
 } // end constructor
} // end class Node
```

A node for a linked list of objects

You can use this class as follows:

```
Node n = new Node(new Integer(6));
Node first = new Node(new Integer(9), n);
```

Figure 5-7 illustrates this scenario. The constructors are used to initialize the data field and a link value that is either *null* or provided as an argument. Although the data portion of each node in a linked list can reference an instance of any class, the figure illustrates data items that are instances of the class *java.lang.Integer*.

To complete our general description of the linked list, we must consider two other issues. First, what is the value of the data field *next* in the last node in the list? By setting this field to *null*, you can easily detect when you are at the end of the linked list.

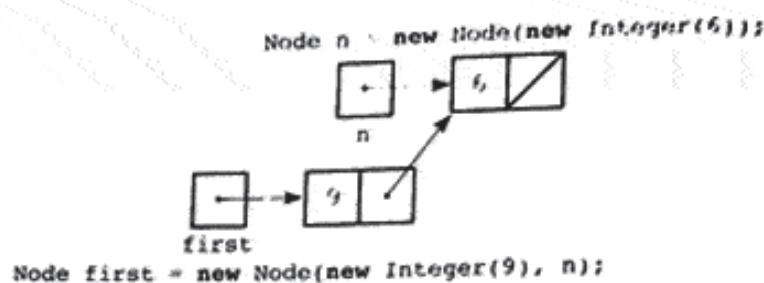


FIGURE 5-7

Using the `Node` constructor to initialize a data field and a link value.

Second, nothing so far references the beginning of the linked list. If you cannot get to the beginning of the list, you cannot get to the second node in the list, and if you cannot get to the second node in the list, you cannot get to the third node in the list, and so on. The solution is to have an additional reference variable whose sole purpose is to locate the first node in the linked list. Such a variable is often called `head`.

Observe in Figure 5-8 that the reference variable `head` is different from the other reference variables in the diagram in that it is not within one of the nodes. Rather, it is a simple reference variable that is external to the linked list, whereas the `next` data fields are internal reference variables within the nodes of the list. The variable `head` simply enables you to access the list's beginning. Also, note that `head` always exists, even at times when there are no nodes in the linked list. The statement

```
Node head = null;
```

creates the variable `head`, whose value is initially `null`. This indicates that `head` does not reference anything, and therefore that this list is empty.

It is a common mistake to think that before you can assign `head` a value, you must execute the statement `head = new Node()`. This misconception is rooted in the belief that the variable `head` does not exist unless you create a new node. This is not at all true; `head` is a reference variable waiting to be

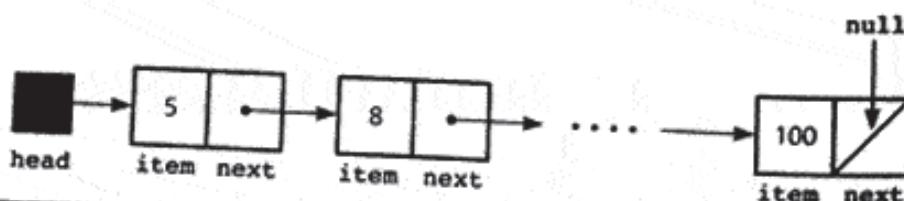


FIGURE 5-8

A `head` reference to a linked list.

assigned a value. Thus, for example, you can assign `null` to `head` without first using `new`. In fact, the sequence

```
head = new Node(); // Don't really need to use new here
head = null; // since we lose the new Node object here
```

A common misconception

destroys the contents of the only reference—`head`—to the newly created node, as Figure 5-9 illustrates. Thus, you have needlessly created a new node and then made it inaccessible. Remember that when you remove the last reference from a node, the system marks it for garbage collection.

As was mentioned earlier, you do not need references to implement a linked list. Programming Problem 10 at the end of this chapter discusses an implementation that uses an array to represent the items in a linked list. Although sometimes useful, such implementations are unusual.

## 5.2 Programming with Linked Lists

The previous section illustrated how you can use reference variables to implement a linked list. This section begins by developing algorithms for displaying the data portions of such a linked list and for inserting items into and deleting items from a linked list. These linked list operations are the basis of many of the data structures that appear throughout the remainder of the book. Thus, the material in this section is essential to much of the discussion in the following chapters.

### Displaying the Contents of a Linked List

Suppose now that you have a linked list, as was pictured in Figure 5-8, and that you want to display the data in the list. A high-level pseudocode solution is

```
Let a variable curr reference the first node in
the linked list
while (the curr reference is not null) {
 Display the data portion of the current node
 Set the curr reference to the next field of the
 current node
} // end while
```



```
head = new Node(new Integer(5)); head = null;
```

**FIGURE 5-9**

A lost node

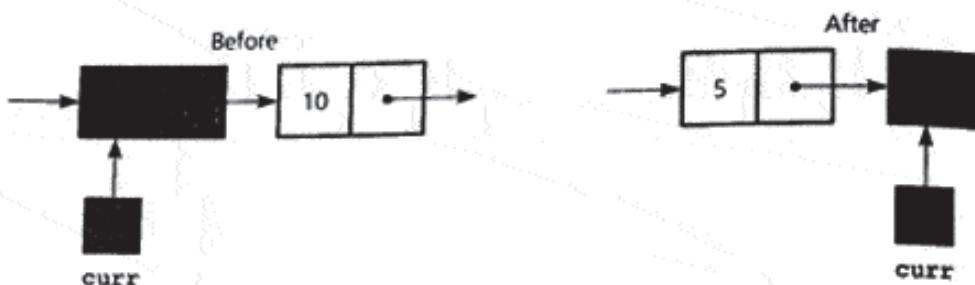


FIGURE 5-10

The effect of the assignment `curr = curr.next`

This solution requires that you keep track of the current position within the linked list. Thus, you need a reference variable `curr` that references the current node. Initially, `curr` must reference the first node. Since `head` references the first node, simply copy `head` into `curr` by writing

```
Node curr = head;
```

To display the data portion of the current node, you can use the statement<sup>1</sup>

```
System.out.println(curr.item);
```

Finally, to advance the current position to the next node, you write

```
curr = curr.next;
```

Figure 5-10 illustrates this action. If the previous assignment statement is not clear, consider

```
temp = curr.next;
curr = temp;
```

and then convince yourself that the intermediate variable `temp` is not necessary. These ideas lead to the following loop in Java:

```
// Display the data in a linked list that head
// references.
// Loop invariant: curr references the next node to be
// displayed
```

1. See Chapter 1 for a discussion of `System.out.println` with object parameters.

```

for (Node curr = head; curr != null; curr = curr.next) {
 System.out.println(curr.item);
} // end for

```

The variable *curr* references each node in a nonempty linked list during the course of the *for* loop's execution, and so the data portion of each node is displayed. After the last node is displayed, *curr* becomes *null* and the *for* loop terminates. When the list is empty—that is, when *head* is *null*—the *for* loop is correctly skipped.

A common error in the *for* statement is to compare *curr.next* instead of *curr* with *null*. When *curr* references the last node of a nonempty linked list, *curr.next* is *null*, and so the *for* loop would terminate before displaying the data in the last node. In addition, when the list is empty—that is, when *head* and therefore *curr* are *null*—*curr.next* will throw a *NullPointerException*. Such references are incorrect and should be avoided.

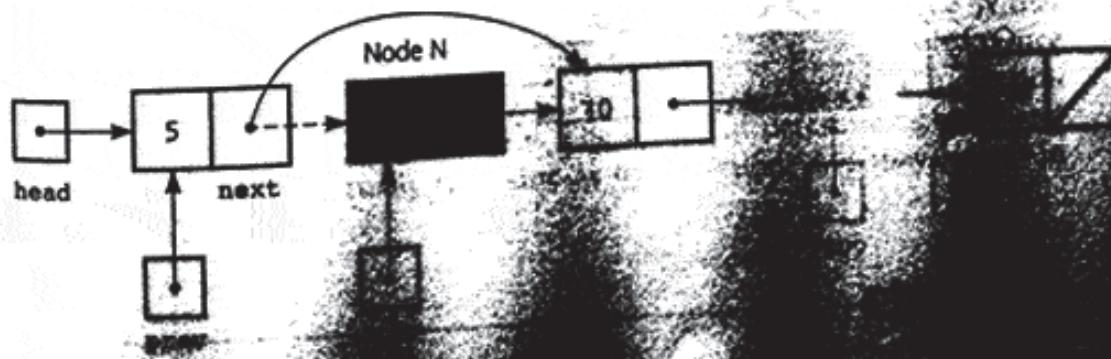
Displaying a linked list is an example of a common operation, list traversal. A traversal sequentially visits each node in the list until it reaches the end of the list. Our example displays the data portion of each node when it visits the node. Later in this book, you will see that you can do other useful things to a node during a visit.

Displaying a linked list does not alter it; you will now see operations that modify a linked list by deleting and inserting nodes. These operations assume that the linked list has already been created. Ultimately, you will see how to build a linked list by inserting nodes into an initially empty list.

A traverse operation visits each node in the linked list

## Deleting a Specified Node from a Linked List

So that you can focus on how to delete a particular node from a linked list, assume that the linked list shown in Figure 5-11 already exists. Notice that, in addition to *head*, the diagram includes two external reference variables, *curr* and *prev*. The task is to delete the node that *curr* references. As you soon will see, you also need *prev* to complete the deletion. For the moment, do not worry about how to establish *curr* and *prev*.



FIGURE

Deleting

As Figure 5-11 indicates, you can delete a node  $N$ , which  $curr$  references, by altering the value of the reference  $next$  in the node that precedes  $N$ . You need to set this data field to reference the node that follows  $N$ , thus bypassing  $N$  on the chain. (The dashed line indicates the old reference value.) Notice that this reference change does not directly affect node  $N$ . Since  $curr$  still references node  $N$ , the node remains in existence, and it references the same node that it referenced before the deletion. However, the node has effectively been deleted from the linked list. For example, the method `displayList` from the previous section would not display the contents of node  $N$ . If  $curr$  is the only reference to node  $N$ , when we change  $curr$  to reference another node or set it equal to `null`, node  $N$  is marked for garbage collection.

To accomplish this deletion, notice first that if only the reference  $curr$  points to  $N$ , you would have no direct way to access the node that precedes  $N$ . After all, you cannot follow the links in the list backward. However, notice that the reference variable  $prev$  in Figure 5-11 references the node that precedes  $N$  and makes it possible for you to alter that node's  $next$  data field. Doing so deletes node  $N$  from the linked list. The following assignment statement is all that you need to delete the node that  $curr$  references:

#### Deleting an interior node

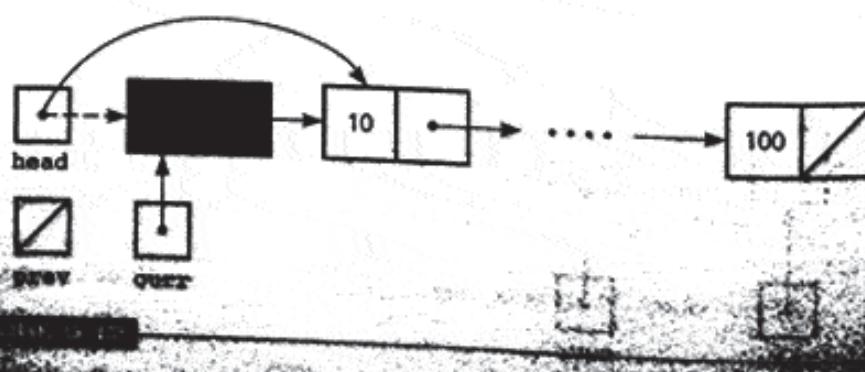
```
prev.next = curr.next;
```

A question comes to mind at this point:

- Does the previous method work for any node  $N$ , regardless of where in the linked list it appears?

No, the method does not work if the node to be deleted is the *first* node in the list, because it certainly does not make sense to assert that  $prev$  references the node that precedes this node! Thus, *deletion of the first node in a linked list is a special case*, as Figure 5-12 depicts. In this case,  $curr$  references the first node and  $prev$  is `null`.

When you delete the first node of the list, you must change the value of `head` to reflect the fact that, after the deletion, the list has a new first node.



That is, the node that was second prior to the deletion is now first. You make this change to *head* by using the assignment statement

```
head = head.next;
```

As was the case for the deletion of an interior node, the *head* reference now bypasses the old first node. Notice also that if the node to be deleted is the *only* node in the list—and thus it is both the first node and the last node—the previous assignment statement assigns the value *null* to the variable *head*. Recall that the value *null* in *head* indicates an empty list, and so this assignment statement handles the deletion of the only node in a list correctly.

If the node *N* is no longer needed, you should change the *next* data field of the node *N* to *null* and also the value of *curr* to *null*, as the following statements show:

```
curr.next = null;
curr = null;
```

This serves two purposes. First, the reference variables *curr* and *next* (in node *N*) can't be inadvertently followed, leading to subtle errors later in the program. Second, the system can now use this returned memory and possibly even reallocate it to your program as a result of the *new* operator.

So far, we have deleted the node *N* that *curr* references, given a reference variable *prev* to the node that precedes *N*. However, another question remains:

- How did the variables *curr* and *prev* come to reference the appropriate nodes?

To answer this question, consider the context in which you might expect to delete a node. In one common situation, you need to delete a node that you specify by position. Such is the case if you use a linked list to implement an ADT list. In another situation, you need to delete a node that contains a particular data value. Such is the case if you use a linked list to implement an ADT sorted list. In both of these situations, you do not pass the values of *curr* and *prev* to the deletion method, but instead the method establishes these values as its first step by searching the linked list for the node *N* that either is at a specified position or contains the data value to be deleted. Once the method finds the node *N*—and the node that precedes *N*—the deletion of *N* proceeds as described previously. The details of determining *curr* and *prev* for deletion are actually the same as for insertion, and they appear in the next section.

Deleting the first node is a special case

Three steps to  
delete a node from  
a linked list

To summarize, the deletion process has three high-level steps:

1. Locate the node that you want to delete.
2. Disconnect this node from the linked list by changing references.
3. Return the node to the system.

Later in this chapter, we will incorporate this deletion process into the implementation of the ADT list.

### Inserting a Node into a Specified Position of a Linked List

Figure 5-13 illustrates the technique of inserting a new node into a specified position of a linked list. You insert the new node, which the reference variable *newNode* references, between the two nodes that *prev* and *curr* reference. As the diagram suggests, you can accomplish the insertion by using the following pair of assignment statements:

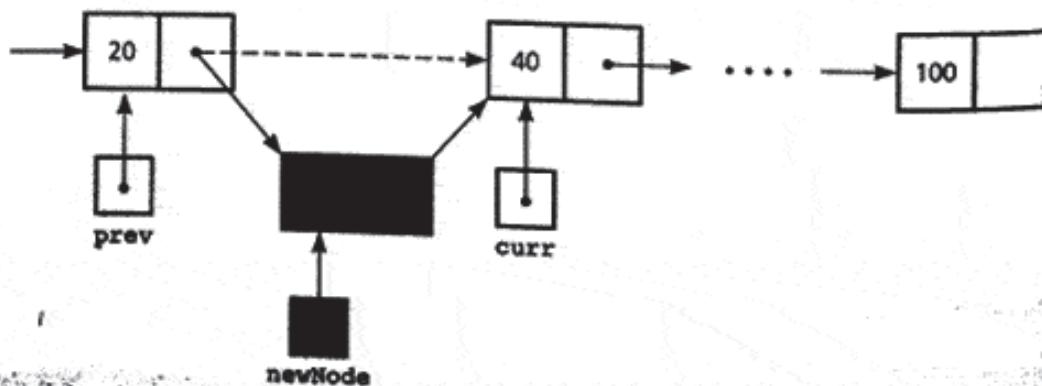
Inserting a node  
between nodes

```
newNode.next = curr;
prev.next = newNode;
```

The following two questions are analogous to those previously asked about the deletion of a node:

- How did the variables *newNode*, *curr*, and *prev* come to reference the appropriate nodes?
- Does the method work for inserting a node into any position of a linked list?

The answer to the first question, like the answer to the analogous question for deletion, is found by considering the context in which you will use the insertion operation. You establish the values of *curr* and *prev* by traversing the linked list until you find the proper position for the new item.



can then use the `new` operator to create a new node that references the item as follows:

```
newNode = new Node(item);
```

Creating a node for the new item

You can now insert the node into the list, as was just described.

The answer to the second question is that *insertion, like deletion, must account for special cases*. First, consider the insertion of a node at the beginning of the linked list, as shown in Figure 5-14. You must make `head` reference to `newNode`, and the new node must reference the node that had been at the beginning of the list. You accomplish this by using these statements:

```
newNode.next = head;
head = newNode;
```

Inserting a node at the beginning of a linked list

Observe that if the list is empty before the insertion, `head` is `null`, so the `next` reference of the new item is set to `null`. This step is correct because the new item is the last item—as well as the first item—on the list.

Figure 5-15 shows the insertion of a new node at the end of a linked list. Insertion is potentially a special case because the intention of the pair of assignment statements

```
curr.next = curr;
curr.next = newNode;
```

If `curr` is `null`, inserting at the end of a linked list is not a special case

is to insert the new node *between* the node that `curr` references and the node that `prev` references. If you are to insert the new node at the end of the list, what node should `curr` reference? In this situation, it makes sense to view the value of `curr` as `null` because, as you traverse the list, `curr` becomes `null` as it moves past the end of the list. Observe that if `curr` has the value `null` and

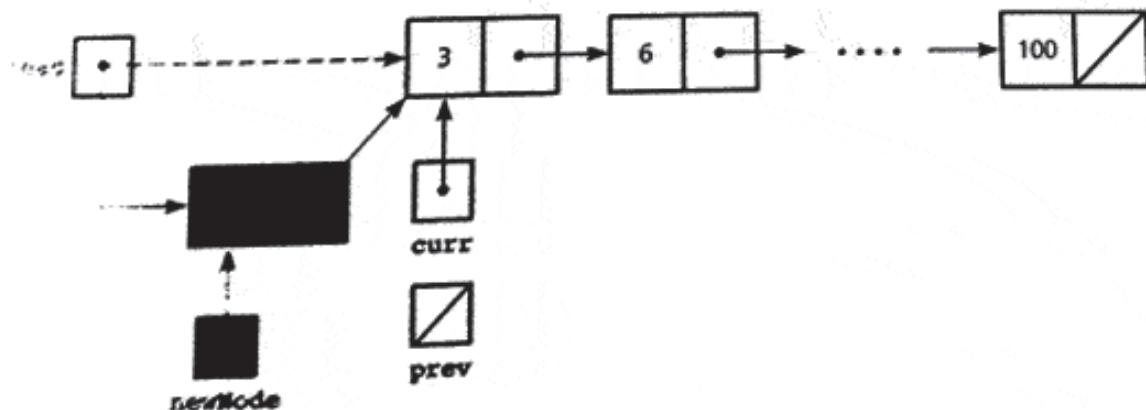


FIGURE 5-14

Inserting at the beginning of a linked list

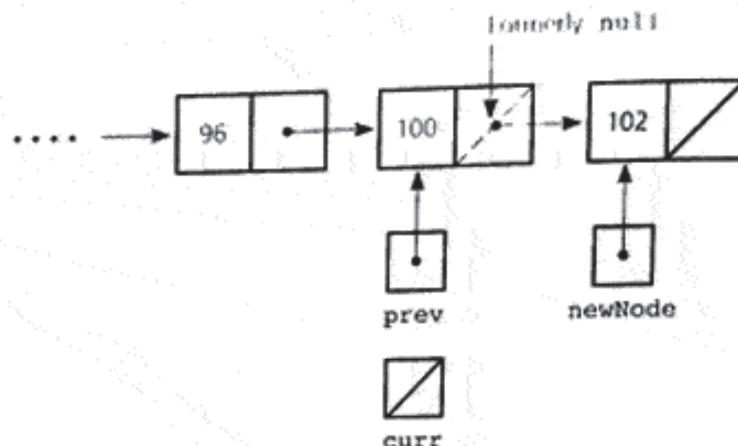


FIGURE 5-15

Inserting at the end of a linked list

`prev` references the last node on the list, the previous pair of assignment statements will indeed insert the new node at the end of the list. Thus, insertion at the end of a linked list is not a special case.

To summarize, the insertion process requires three high-level steps:

1. Determine the point of insertion.
2. Create a new node and store the new data in it.
3. Connect the new node to the linked list by changing references.

**Determining `curr` and `prev`.** Let us now examine in more detail how to determine the references `curr` and `prev` for the insertion operation just described. As was mentioned, this determination depends on the context in which you will insert a node. As an example, consider a linked list of integers that are sorted into ascending order using the `IntegerNode` class in the previous section. To simplify the discussion, assume that the integers are distinct; that is, no duplicates are present in the list.

To determine the point at which the value `newValue` should be inserted into a sorted linked list, you must traverse the list from its beginning until you find the appropriate place for `newValue`. This appropriate place is just before the node that contains the first data item greater than `newValue`. You know that you will need a reference `curr` to the node that is to follow the new node; that is, `curr` references the node that contains the first data item greater than `newValue`. You also need a reference `prev` to the node that is to precede the new node; that is, `prev` references the node that contains the last data item smaller than `newValue`. Thus, as you traverse the linked list, you keep both a current reference `curr` and a trailing reference `prev`. When you reach the node that contains the first value larger than `newValue`, the trailing reference `prev` references the previous node. At this time, the new node is positioned between the two nodes that `prev` and `curr` reference.

Three steps to insert a new node into a linked list

A first attempt at some pseudocode follows:

```

determine the point of insertion into a sorted
linked list
initialize prev and curr to start the traversal
from the beginning of the list
prev = null
curr = head

advance prev and curr as long as
newValue > the current data item
loop invariant: newValue > data items in all
nodes at and before the node that prev references
while (newValue > curr.item) { // causes a problem!
 prev = curr
 curr = curr.next
} // end while

```

A first attempt at a solution

Unfortunately, the **while** loop causes a problem when the new value is less than all the values in the list, that is, when the insertion will be at the *end* of the linked list (or when the linked list is empty). Eventually, the **while** statement compares *newValue* to the value in the last node. During that execution of the loop, *curr* is assigned the value *null*. After this iteration, *newValue* is again compared to *curr.item*, which, when *curr* is *null*, will cause the exception *NullPointerException*.

To solve this problem, you need another test in the termination condition of the **while** statement so that the loop exits when *curr* becomes *null*. Thus, replace the **while** statement with

```
while (curr != null && newValue > curr.item)
```

The revised pseudocode is

```

determine the point of insertion into a sorted
linked list
initialize prev and curr to start the traversal
from the beginning of the list
prev = null
curr = head

// advance prev and curr as long as newValue > the
// current data item; do not go beyond end of list
// Loop invariant: newValue > data items in all
// nodes at and before the node that prev references
while (curr != null && newValue > curr.item) {
 prev = curr
 curr = curr.next
} // end while

```

The correct solution

Notice how the `while` statement also solves the problem of inserting a node at the end of the linked list. In the case where `newValue` is greater than all the values in the list, `prev` references the last node in the list and `curr` becomes `null`, thus terminating the `while` loop. (See Figure 5-16.) Therefore, as you saw earlier, you can insert the new node at the end of the list by using the standard pair of assignment statements

Insertion at the end of a linked list is not a special case

```
newNode.next = curr;
prev.next = newNode;
```

Now consider the insertion of a node at the beginning of the linked list. This situation arises when the value to be inserted is *smaller* than all the values currently in the list. In this case, the `while` loop in the previous pseudocode is never entered, so `prev` and `curr` maintain their original values, as Figure 5-17 illustrates. In particular, `prev` maintains its original value of `null`. This is the only situation in which the value of `prev` is equal to `null` after execution of the `while` loop ends. Thus, you can detect an insertion at the beginning of the list by comparing `prev` to `null`.

Observe that the solution also correctly handles insertion into an empty linked list as an insertion at the beginning of the list. When the list is empty, the statement `curr = head` assigns `curr` an initial value of `null`, and thus the `while` loop is never entered. Therefore, `prev` maintains its original value of `null`, indicating an insertion at the beginning of the list.

A little thought should convince you that the solution that determines the point of insertion also works for deletion. If you want to delete a given integer from a linked list of sorted integers, you obviously want to traverse the list until you find the node that contains the value sought. The previous pseudocode will do just that: `curr` will reference the desired node and `prev` either will reference the preceding node or, if the desired node is first on the list, will be `null`, as shown in Figure 5-17.

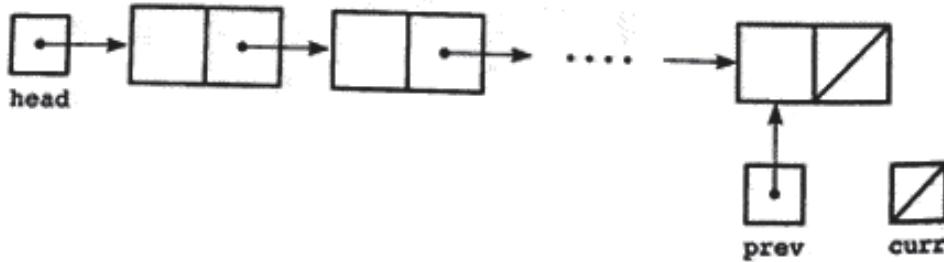
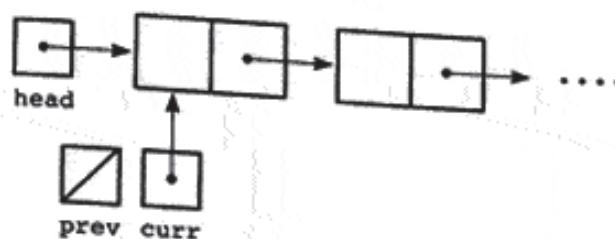


FIGURE 5-16

When `prev` references the last node and `curr` is `null`, insertion will be at the end of the linked list

**FIGURE 5-17**

When *prev* is *null* and *curr* references the first node, insertion or deletion will be at the beginning of the linked list

The following Java statements implement the previous pseudocode:

```
determine the point of insertion or deletion
for a sorted linked list
Loop invariant: newValue > data items in all
nodes at and before the node that prev references
for (prev = null, curr = head;
 (curr != null) && (newValue > curr.item);
 prev = curr, curr = curr.next) {
 // no statements in loop body
} // end for
```

A Java solution for  
the point of insertion  
or deletion

Recall that the `&&` (*and*) operator in Java does not evaluate its second operand if its first operand is *false*. Thus, when *curr* becomes *null*, the loop exits without attempting to evaluate *curr.item*. It is, therefore, essential that *curr != null* be first in the logical expression.

Notice that this implementation relies on the ability to compare one value to another by using the built-in greater than (`>`) operation for the primitive data type `int`. Suppose instead that the items in the list are of data type `Object`. As mentioned in Chapter 4, you can compare objects if they implement the interface `java.lang.Comparable` and have an implementation of the method `compareTo`.

You can then use the following code to find the location of the item *newValue* of type `Comparable` within the list:

```
// determine the point of insertion or deletion
// for a sorted linked list of objects
// Loop invariant: newValue > data items (using
// compareTo method) in all nodes at and before
// the node that prev references
for (prev = null, curr = head;
 (curr != null) &&
 (newValue.compareTo(curr.item) > 0);
 prev = curr, curr = curr.next) {
} // end for
```

Determining the  
point of insertion or  
deletion for a sorted  
linked list of objects

Use `compareTo` to compare objects

`head` and  
`numItems` are  
private data fields

`curr` and `prev`  
should not be data  
fields of the class

The `compareTo` method defines the criteria to decide when objects are equal or when one object is less than or greater than another. This in turn can be used to create a sorted list of objects based upon the criteria defined by the new comparison method.

Determining the values of `curr` and `prev` is simpler when you insert or delete a node by position instead of by its value. This determination is necessary when you use a linked list to implement the ADT list, as you will see next.

## A Reference-Based Implementation of the ADT List

This section considers how you can use Java references instead of an array to implement the ADT list. Unlike the array-based implementation, a reference-based implementation does not shift items during insertion and deletion operations. It also does not impose a fixed maximum length on the list—except, of course, as imposed by the storage limits of the system.

As in Chapter 4, and as we will do in the rest of the book, we will implement this ADT as a Java class. For the array-based implementation, we wrote declarations for public methods corresponding to the operation of the ADT list. These declarations will appear unchanged in the reference-based implementation.

You need to represent the items in the ADT list and its length. Figure 5-18 indicates one possible way to represent this data by using references. The variable `head` references a linked list of the items in the ADT list, where the first node in the linked list contains the first item in the ADT list and so on. The variable `numItems` is an integer that is the current number of items in the list. Both `head` and `numItems` will be private data fields of our class.

As you saw previously, you use two references—`curr` and `prev`—to manipulate a linked list. These reference variables will be local to the methods that need them; they are not appropriate data fields of the class.

Recall that the ADT list operations for insertion, deletion, and retrieval specify the position number  $i$  of the relevant item. Assume that position number 0 is the first node in the list, referenced by `head`. In an attempt to obtain values for `curr` and `prev` from  $i$ , suppose that you define a method `find( $i$ )` that returns a reference to the  $i^{\text{th}}$  node in the linked list. If `find` provides a reference `curr` to the  $i^{\text{th}}$  node, how will you get a reference `prev` to the previous node, that is, to the  $(i - 1)^{\text{th}}$  node? You can get the value of `prev` by invoking `find( $i - 1$ )`. Instead of calling `find` twice, however, note that once you have `prev`, `curr` is simply `prev.next`. The only exception to using `find` in this way is for the first node, but you know immediately from  $i$  whether the

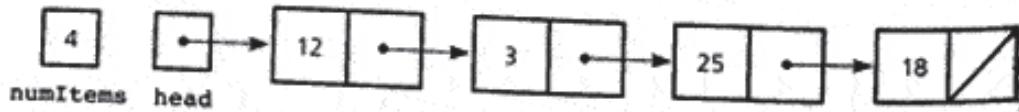


FIGURE 5-18

A reference-based implementation of the ADT list

A client involves the first node. If it does, you know the reference to the first node, namely `head`, without invoking `find`.

The method `find` is not an ADT operation. Because `find` returns a reference to a node, you would not want any client to call it. Such clients should be able to use the ADT without knowledge of the references that the implementation uses. It is perfectly reasonable for the implementation of an ADT to define variables and methods that the rest of the program should not access. Therefore, `find` is a private method that only the implementations of the ADT operations call.

The following interface specification developed in Chapter 4 will be used for the reference-based implementation of the ADT list. The pre- and postconditions for the ADT list operations are the same as for the array-based implementation that you saw in Chapter 4; they are omitted here to save space.

`find` is a private method

```
package List;

Interface for the ADT list

public interface ListInterface {
 list operations:
 public boolean isEmpty();
 public int size();
 public void add(int index, Object item)
 throws ListIndexOutOfBoundsException;
 public void remove(int index)
 throws ListIndexOutOfBoundsException;
 public Object get(int index)
 throws ListIndexOutOfBoundsException;
 public void removeAll();
 end ListInterface
```

The implementation of the list begins as follows:

```
package List;

Reference-based implementation of ADT list.

public class ListReferenceBased implements ListInterface {
 reference to linked list of items
 private Node head;
 private int numItems; // number of items in list

 definitions of constructors and methods
 ...
}
```

You include the implementations of the class's methods at this point in the class, as well as any private methods that may be needed. We now examine each of these implementations.

**Default constructor.** The default constructor simply initializes the data fields *numItems* and *head*:

```
Default constructor: public ListReferenceBased() {
 numItems = 0;
 head = null;
} // end default constructor
```

Since the variables *numItems* and *head* are initialized to these same values by default, this constructor is really not necessary. But if you have other constructors defined and you want to allow for a constructor without parameters, it must be defined explicitly. In general, it's a good idea to define all constructors explicitly.

**List operations.** The methods *isEmpty* and *size* have straightforward implementations:

```
public boolean isEmpty() {
 return numItems == 0;
} // end isEmpty

public int size() {
 return numItems;
} // end size
```

Because a linked list does not provide direct access to a specified position, the retrieval, insertion, and deletion operations must all traverse the list from its beginning until the specified point is reached. The method *find* performs this traversal and has the following implementation:

Private method to  
locate a particular  
node

```
private Node find(int index) {
// -----
// Locates a specified node in a linked list.
// Precondition: index is the number of the desired
// node. Assumes that 1 <= index <= numItems+1
// Postcondition: Returns a reference to the desired
// node.
// -----
 Node curr = head;
 for (int skip = 0; skip < index; skip++) {
 curr = curr.next;
 } // end for
 return curr;
} // end find
```

The precondition for *find* requires the *index* to be in the proper range. The *get* operation calls *find* to locate the desired node:

```
public Object get(int index)
 throws ListIndexOutOfBoundsException {
 if (index >= 0 && index < numItems) {
 // get reference to node, then data in node
 Node curr = find(index);
 Object dataItem = curr.item;
 return dataItem;
 }
 else {
 throw new ListIndexOutOfBoundsException(
 "List index out of bounds on get");
 } // end if
} // end get
```

Retrieved by position

The reference-based implementations of the insertion and deletion operations use the linked list processing techniques developed earlier in this chapter. To insert an item after the first item of a list, you must first obtain a reference to the preceding item. Insertion into the first position of a list is a special case.

```
public void add(int index, Object item)
 throws ListIndexOutOfBoundsException {
 if (index >= 0 && index < numItems+1) {
 if (index == 0) {
 // insert the new node containing item at
 // beginning of list
 Node newNode = new Node(item, head);
 head = newNode;
 }
 else {
 Node prev = find(index-1);

 // insert the new node containing item after
 // the node that prev references
 Node newNode = new Node(item, prev.next);
 prev.next = newNode;
 } // end if
 numItems++;
 }
 else {
 throw new ListIndexOutOfBoundsException(
 "List index out of bounds on add");
 } // end if
} // end add
```

Insertion at a given position

The `remove` operation is analogous to insertion. To delete an item that occurs after the first item in a list, you must first obtain a reference to the item that precedes it. Removal from the first position of a list is a special case.

*Deletion from a  
List Class*

```
public void remove int index,
 throws ListIndexOutOfBoundsException {
 if (index < 0 || index > numItems) {
 if (index == 0) {
 // delete the first node from the list
 head = head.next;
 }
 else {
 Node prev = find(index-1);
 // delete the node after the node that prev
 // references, save reference to node
 Node curr = prev.next;
 prev.next = curr.next;
 }
 end if
 numItems--;
 }
 end if
 else {
 throw new ListIndexOutOfBoundsException(
 "List index out of bounds on remove");
 }
}
end remove
```

The `removeAll` operation simply sets the `head` reference to `null`, making the nodes in the list unreachable and thus marking them for garbage collection.

```
public void removeAll() {
 // setting head to null causes list to be
 // unreachable and thus marked for garbage
 // collection
 head = null;
 numItems = 0;
}
end removeAll
```

## Comparing Array-Based and Reference-Based Implementations

Typically, the various implementations that a programmer contemplates for a particular ADT have advantages and disadvantages. When you must select an implementation, you should weigh these advantages and disadvantages before you make your choice. As you will see, the decision among possible implementations of an ADT is one that you must make time and time again. This section

spares the two implementations of the ADT list that you have seen as an example of how you should proceed in general.

The array-based implementation that you saw in Chapter 4 appears to be a sensible approach. An array behaves like a list, and arrays are easy to use. However, as was already mentioned, an array has a fixed size; it is possible for the number of items in the list to exceed this fixed size. In practice, when choosing among implementations of an ADT, you must ask the question, does the fixed-size restriction of an array-based implementation present a problem in the context of a particular application? The answer to this question depends on two factors. The obvious factor is whether or not, for a given application, you can predict in advance the maximum number of items in the ADT at any time. If you cannot, it is quite possible that an operation—and hence the program—will fail because the ADT in the context of a particular application requires more storage than the array can provide.

On the other hand, if, for a given application, you can predict in advance the maximum number of items in the ADT list at any one time, you must ignore a more subtle factor: Would you waste storage by declaring an array too large enough to accommodate this maximum number of items? Consider a case in which the maximum number of items is large, but you suspect that this never rarely will be reached. For example, suppose that your list could contain as many as 10,000 items, but the actual number of items in the list never exceeds 50. If you declare 10,000 array locations at compilation time, at least 9,950 array locations will be wasted most of the time. In both of the previous cases, the array-based implementation given in Chapter 4 is not preferable.

What if you used a resizable array? Because you would use the *new* operator to allocate a larger array dynamically, you would be able to provide as much storage as the list needs (within the bounds of the particular computer, of course). Thus, you would not have to predict the maximum size of the list. However, if you doubled the size of the array each time you reached the end of the array—which is a reasonable approach to enlarging the array—you still might have many unused array locations. In the example just given, you could allocate an array of 50 locations initially. If you actually have 10,000 items in your list, array doubling will eventually give you an array of 12,800 locations, 2,800 more than you need. Remember also that you waste time by copying the array each time you need more space.

Now suppose that your list will never contain more than 25 items. You could allocate enough storage in the array for the list and know that you would waste little storage when the list contained only a few items. With respect to its size, an array-based implementation is perfectly acceptable in this case.

A reference-based implementation can solve any difficulties related to the fixed size of an array-based implementation. You use the *new* operator to allocate storage dynamically, so you do not need to predict the maximum size of the list. Because you allocate memory one item at a time, the list will be allocated only as much storage as it needs. Thus, you will not waste storage.

There are other differences between the array-based and reference-based implementations. These differences affect both the time and memory

Arrays are easy to use, but they have a fixed size

Can you predict the maximum number of items in the ADT?

Will an array waste storage?

Increasing the size of a resizable array can waste storage and time

An array-based implementation is a good choice for a small list

Linked lists do not have a fixed size

The item after an array item is implied; in a linked list, an item explicitly references the next item

An array-based implementation requires less memory than a reference-based implementation

You can access array items directly with equal access time

You must traverse a linked list to access its  $i^{\text{th}}$  node

The time to access the  $i^{\text{th}}$  node in a linked list depends on  $i$

Insertion into and deletion from a linked list do not require you to shift data

Insertion into and deletion from a linked list require a list traversal

requirements of the implementations. Any time you store a collection of data in an array or a linked list, the data items become ordered; that is, there is a first item, a second item, and so on. This order implies that a typical item has a predecessor and a successor. In an array `anArray`, the location of the next item after the item in `anArray[i]` is *implicit*—it is in `anArray[i+1]`. In a linked list, however, you *explicitly* determine the location of the next item by using the reference in the current node. This notion of an implicit versus explicit next item is one of the primary differences between an array and a linked list. Therefore, an advantage of an array-based implementation is that it does not have to store explicit information about where to find the next data item, thus requiring less memory than a reference-based implementation.

Another, more important advantage of an array-based implementation is that it can provide direct access to a specified item. For example, if you use the array `items` to implement the ADT list, you know that the item associated with list position  $i$  is stored in `items[i-1]`. Accessing either `items[0]` or `items[49]` takes the same amount of time. That is, the access time is constant for an array.

On the other hand, if you use a linked list to implement the ADT list, you have no way of immediately accessing the node that contains the  $i^{\text{th}}$  item. To get to the appropriate node, you use the `next` data fields to traverse the linked list from its beginning until you reach the  $i^{\text{th}}$  node. That is, you access the first node and get the reference to the second node, access the second node and get the reference to the third node, and so on until you finally access the  $i^{\text{th}}$  node. Clearly, the time it takes you to access the first node is less than the time it takes to access the 50th node. The access time for the  $i^{\text{th}}$  node depends on  $i$ .

The type of implementation chosen will affect the efficiency of the operations of the ADT list. An array-based `get` is almost instantaneous, regardless of which list item you access. A reference-based retrieval operation like `get`, however, requires  $i$  steps to access the  $i^{\text{th}}$  item in the list.

You already know that the array-based implementation of the ADT `List` requires you to shift the data when you insert items into or delete items from the list. For example, if you delete the first item of a 20-item list, you must shift 19 items. In general, deleting the  $i^{\text{th}}$  item of a list of  $n$  items requires  $n - i$  shifts. Thus, `remove` requires  $n - 1$  shifts to delete the first item, but zero shifts to delete the last item. The list insertion operation `add` has similar requirements.

In contrast, you do not need to shift the data when you insert items into or delete items from the linked list of a reference-based implementation. The methods `add` and `remove` require essentially the same effort, regardless of the length of the list or the position of the operation within the list, once you know the point of insertion or deletion. Finding this point, however, requires list traversal, the time for which will vary depending on where in the list the operation will occur. Recall that the private method `find` performs this traversal. If you examine the definition of `find`, you will see that `find` requires  $i$  assignment operations. Thus, `find`'s effort increases with  $i$ .

We will continue to compare various solutions to a problem throughout this book. Chapter 10 will introduce a more formal way to discuss the efficiency of algorithms. Until then, our discussions will be informal.

## Passing a Linked List to a Method

How can a method access a linked list? It is sufficient for the method to have access to the list's head reference. From this variable alone, the method can access the entire linked list. In the reference-based implementation of the ADT list that you saw earlier in this chapter, the head reference `head` to the linked list that contains the ADT's items is a private data field of the class `ListReferenceBased`. The methods of this class use `head` directly to manipulate the linked list.

Would you ever want `head` to be an argument of a method? Certainly not for methods outside of the class, because such methods should not have access to the class's underlying data structure. Although on the surface, it would seem that you would never need to pass the head reference to a method, that is not the case. Recursive methods, for example, might need the head reference as an argument. You will see examples of such methods in the next section. Realize that these methods must not be public members of their class. If they were, clients could access the linked list directly, thereby violating the ADT's wall.

As Figure 5-19 illustrates, when `head` is an actual argument to a method, its value is copied into the corresponding formal parameter. The method then can access and alter the nodes in the list. However, the method cannot modify `head`'s value (recall our earlier example in Figure 5-4). This is fine for situations, such as a search method, that do not modify the list. But what should you do if you want to write a method that may need to modify the `head` reference? For example, a method that inserts a node at the beginning of the list will need to modify the `head` reference. One solution is for the return value of the method to be the new value for the `head` reference. Such an example will appear in the next section.

A method with access to a linked list's `head` reference has access to the entire list

## Processing Linked Lists Recursively

It is possible, and sometimes desirable, to process linked lists recursively. This section examines recursive traversal and insertion operations on a linked list. If

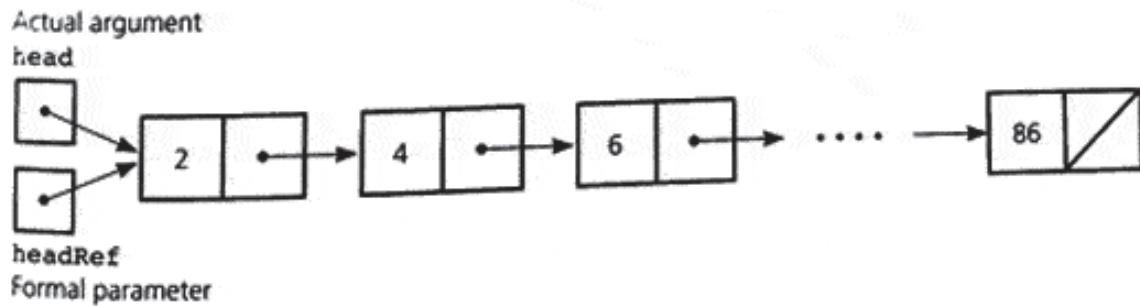


FIGURE 5-19

A head reference as an argument

the recursive methods in this section are members of a class, they should not be public because they require the linked list's head reference as an argument.

**Traversal.** Suppose that you want to display the elements in a list referenced by *head*. That is, you want to write the objects in the order in which they appear in the linked list. The recursive strategy is simply

*Write the first node of the list  
Write the list minus its first node*

The following Java method implements this strategy:

```
private static void writeList(Node nextNode) {
 // -----
 // Writes a list of objects.
 // Precondition: The linked list is referenced by nextNode.
 // Postcondition: The list is displayed. The linked list
 // and nextNode are unchanged.
 // -----
 if (nextNode != null) {
 // write the first data object
 System.out.println(nextNode.item);
 // write the list minus its first node
 writeList(nextNode.next);
 } // end if
} // end writeList
```

This method is uncomplicated. It requires that you have direct access only to the first node of the list. The linked list provides this direct access because the list's first node, referenced by *head*, contains the list's first data item. Furthermore, you can easily pass the list minus its first node to *writeList*. If *head* references the beginning of the list, *head.next* references the list minus its first node. You should compare *writeList* to the iterative technique that we used earlier in this chapter to display a linked list.

Now suppose that you want to display the list backward. Chapter 3 already developed two recursive strategies for writing a string *s* backward. Recall that the strategy of the method *writeBackward* is

*Write the last character of string s  
Write string s minus its last character backward*

The strategy of the method *writeBackward2* is

*Write string s minus its first character backward  
Write the first character of string s*

A recursive traversal method

Compare the recursive *writeList* to the iterative technique on page 253

*writeBackward* strategy

*writeBackward2* strategy

You can easily translate these strategies to linked lists. The method `writeBackward` translates to:

- the last node of the list
- the list minus its last node backward

`writeListBackwardStrategy`

The strategy of the method `writeBackward2` translates to:

- the list minus its first node backward
- the first node of the list

`writeListBackward2Strategy`

You saw that these two strategies work equally well when an array is used. However, when a linked list is used, the first strategy is very difficult to implement. If `nextNode` references the node that contains the first node of the list, how do you get to the last node? Even if you had some way to get to the last node in the list quickly, it would be very difficult for you to move toward the front of the list at each recursive call. That is, it would be difficult for you to keep track of the ends of the successively shorter lists that the recursive calls generate. (Later you will see a doubly linked list, which would solve this problem.)

This discussion illustrates one of the primary disadvantages of linked lists: Whereas an array provides direct access to any of its items, a linked list does not. Fortunately, however, the strategy of method `writeBackward2` requires the same access that `writeListBackward2` requires: The list's head reference `nextNode` locates the first node in the list, and `nextNode.next` references the list minus the first node.

The following Java method implements the `writeListBackward2` strategy for a linked list:

`writeListBackward2` is much easier to implement recursively than `writeListBackward`

```
private static void writeListBackward2(Node nextNode) {
 // -----
 // Writes a list of objects backwards.
 // Precondition: The linked list is referenced by
 // nextNode.
 // Postcondition: The list is displayed backwards. The
 // linked list and nextNode are unchanged.
 // -----
 if (nextNode != null) {
 // write the list minus its first node backward
 writeListBackward2(nextNode.next);
 // write the data object in the first node
 System.out.println(nextNode.item);
 } // end if
} // end writeListBackward2
```

**Self-Test Exercise 3** Write the `insert` method. This trace will be similar to the box trace in Figure 5.9. Exercise 5 asks you to write an iterative version of this method. Which version is more efficient?

**Insertion.** Now view the insertion of a node into a sorted linked list from a new perspective—that is, recursive. Later in this book you will need a recursive algorithm to perform an insertion into a linked structure. Interestingly, recursive insertion requires two base cases: a trailing reference and a special case for inserting into the beginning of the list.

Consider the following recursive view of a sorted linked list: A linked list is sorted if its first data item is less than its second data item and the list that begins with the second data item is sorted. More formally, you can state the definition as follows:

The linked list that head references is a sorted linked list if

head is null—the empty list is a sorted linked list;

or

head.next is null (a list with a single node is a sorted linked list);

or

head.item < head.next.item and head.next references a sorted linked list

You can base a recursive insertion on this definition. Notice that the following method inserts the node at one of the base cases—either when the list is empty or when the new data item is smaller than all the data items in the list. In both cases, you need to insert the new data item at the beginning of the list.

```
private static Node insertRecursive(Node headNode,
 java.lang.Comparable newItem) {
 if (headNode == null) ||

 (newItem.compareTo(headNode.item) < 0) {

 // base case: insert newItem at the beginning of the

 // linked list that nextNode references

 Node newNode = new Node(newItem, headNode);

 headNode = newNode;

 }

 else { //insert into rest of linked list

 Node nextNode = insertRecursive(headNode.next,newItem);

 headNode.next = nextNode;

 } // end if

 return headNode;

} // end insertRecursive
```

First, consider the context for `insertRecursive`. Recall from Chapter 5 that the ADT operation `sortedAdd(newItem)` inserts `newItem` into its pre-

acter in the sorted list. As a public method of the class, `sortedAdd` would call `insertRecursive` to do the insertion recursively. However, `insertRecursive` requires the linked list's head reference as an argument. Since the reference `head` is private and hidden from the client, you would not want `insertRecursive` to be an ADT operation. Thus, you would make it private.

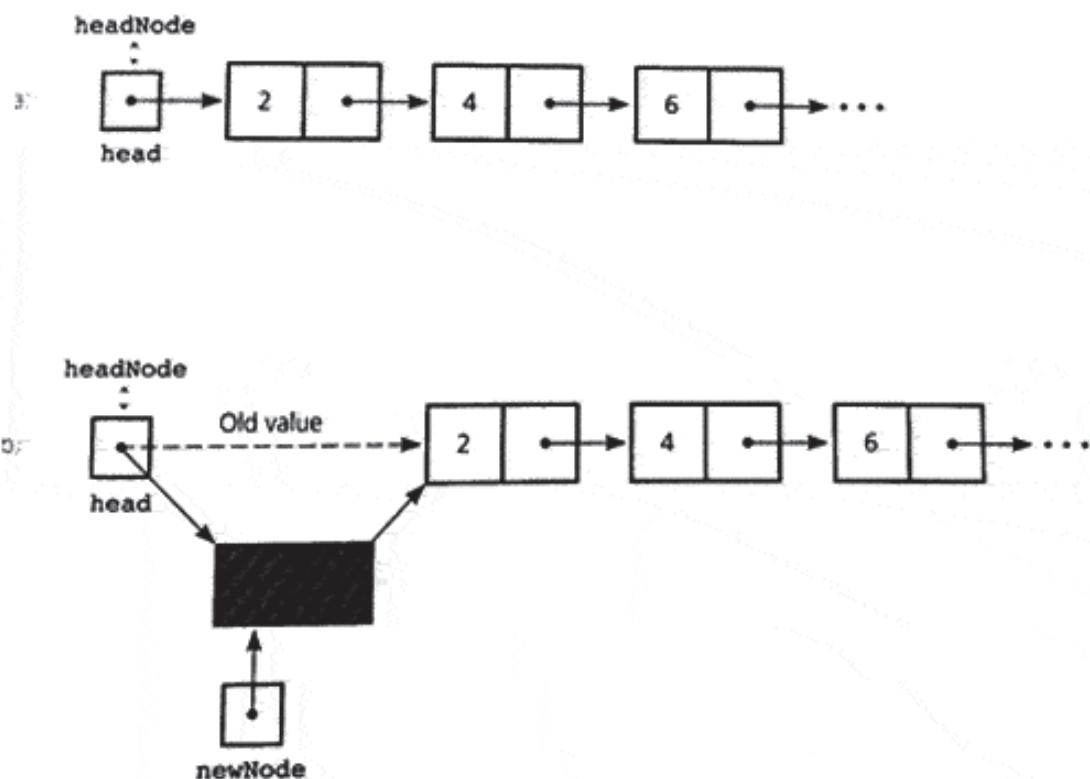
To see how `insertRecursive` works, consider that `sortedAdd` will invoke `insertRecursive` by using the statement

```
head = insertRecursive(head, newItem);
```

Although `insertRecursive` does not maintain a trailing reference, inserting the new node is easy when the base case is reached. Note that within `insertRecursive`, `headNode` references the beginning of the sorted linked list. You use `headNode` to make the new node reference the first node in the original list and then change `headNode` so that it references the new node. Since `insertRecursive` returns `headNode`, `sortedAdd`'s assignment to `head` makes `head` reference the new node as required.

To understand the previous remarks, consider the case in which the new item is to be inserted at the beginning of the original list that has the external reference `head`. In this case, no recursive calls are made, and consequently when the base case is reached—that is, when `newItem.compareTo(headNode.item) < 0`—the actual argument that corresponds to `headNode` is `head`, as Figure 5-20a illustrates.

Insertion occurs at the base case



**FIGURE 5-20**

(a) A sorted linked list; (b) the assignment made for insertion at the beginning of the list

The assignment `headNode = newNode` then sets the method's return value to reference the new node. Upon return of `insertRecursive`, the statements

```
head = insertRecursive(head, newItem);
```

assigns the return value to `head`, as Figure 5-20b shows.

The general case in which the new item is inserted into the interior of a list that `head` references is similar. When `insertRecursive` is first called, the `else` clause of the `if` statement executes, making a recursive call to `insertRecursive`. When the base case is reached, what is the actual argument that corresponds to `headNode`? It is the `next` reference of the node that should precede the new node, as Figure 5-21 illustrates. Therefore, the base case returns a reference to the new node. The `else` clause assigns this reference to `nextNode`, and sets the next reference of the appropriate node to reference the new node.

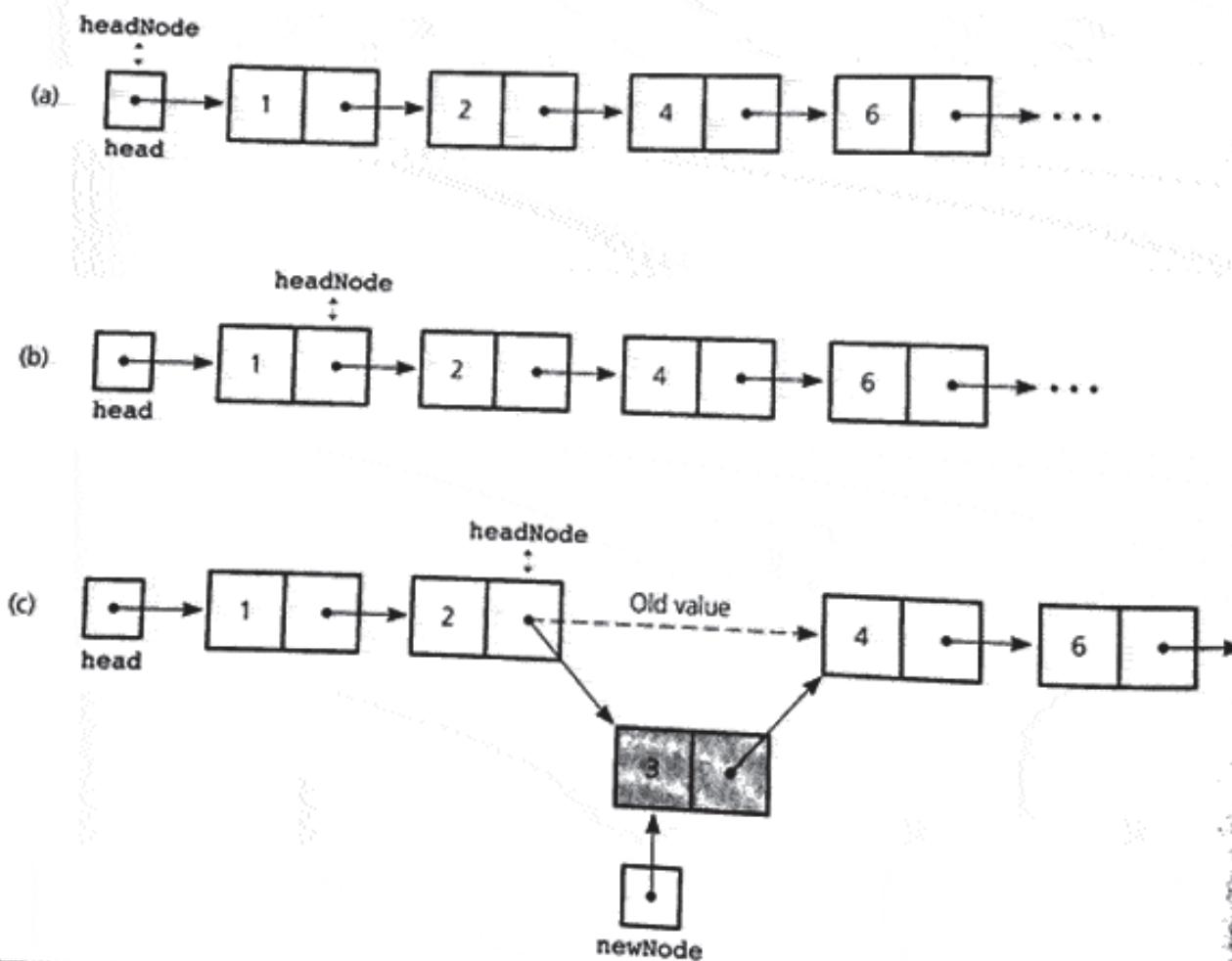


FIGURE 5-21

- (a) The initial call `insertRecursive(head, newItem)`; (b) the first recursive call; (c) the second recursive call inserts at the beginning of the list that `headNode` references

When the original call to `insertRecursive` returns `headNode`, its value is changed from its original value of `head`. Thus, the assignment

```
 head = insertRecursive(head, newItem);
```

leaves the value of `head` unchanged.

Although it could be argued that you should perform the operations on a linked list recursively (after all, recursion does eliminate special cases and need for a trailing reference), the primary purpose in presenting the recursive `insertRecursive` is to prepare you for the binary search tree algorithms presented in Chapter 11.

### 3 Variations of the Linked List

This section briefly introduces several variations of the linked list that you have seen. These variations are often useful, and you will encounter them later in this text. Many of the implementation details are left as exercises. Note that in addition to the data structures discussed in this section, it is possible to have other data structures, such as arrays of references to linked lists and linked lists of linked lists. These data structures are also left as exercises.

#### Tail References

In many situations, you simply want to add an item to the end of a list. For example, maintaining a list of requests for a popular book at the local library would require that new requests for the book be placed at the end of a waiting list. You could use an ADT list called `waitingList` as follows:

```
waitingList.add(request, waitingList.size() + 1);
```

This statement adds `request` to the end of `waitingList`. Recall that in implementing `add` to insert an item at the position indicated, we used the method `find` to traverse the list to that position. Note that this statement actually performs these four steps:

1. Allocate a new node for the linked list.
2. Set the reference in the last node in the list to reference the new node.
3. Put the new `request` in the new node.
4. Set the reference in the new node to `null`.

Each time you add a new request, you must get to the last node in the linked list. One way to accomplish this is to traverse the list each time you add a new request. A much more efficient method uses a tail reference `tail` to remember where the end of the linked list is—just as `head` remembers where the beginning of the list is. Like `head`, `tail` is external to the list. Figure 5-22 illustrates a linked list of integers that has both `head` and `tail` references.

Use a `tail` reference to facilitate adding nodes to the end of a linked list

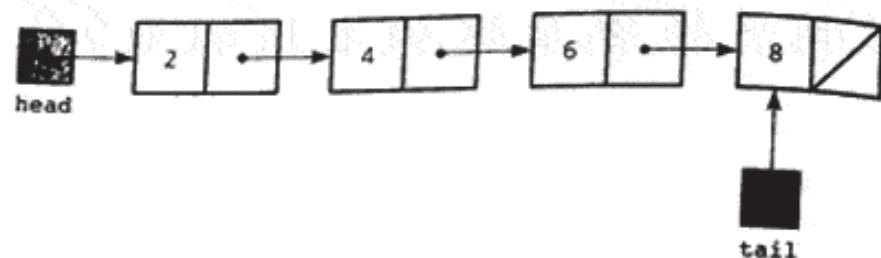


FIGURE 5-22

A linked list with *head* and *tail* references

With *tail* pointing to the end of the linked list, you can perform Step through 4 by using the single statement

```
tail.next = new Node(request, null);
```

This statement sets the *next* reference in the last node in the list to point to the newly allocated node. You then update *tail* so that it references the new last node by writing *tail = tail.next;*. You thus have an easy method for adding a new item to the end of the list. Initially, however, when you insert the first item into an empty linked list, *tail*—like *head*—is *null*. We leave the details of a solution as an exercise.

Treat the first insertion as a special case

## Circular Linked Lists

When you use a computer that is part of a network, you share the services of another computer—called a *server*—with many other users. A similar sharing of resources occurs when you access a central computer by using a remote terminal. The system must organize the users so that only one user at a time has access to the shared computer. By ordering the users, the system can give each user a turn. Because users regularly enter and exit the system (by logging on and logging off), a linked list of user names allows the system to maintain order without shifting names when it makes insertions to and deletions from the list. Thus, the system can traverse the linked list from the beginning and give each user on the list a turn on the shared computer. What must the system do when it reaches the end of the list? It must return to the beginning of the list. However, the fact that the last node of a linked list does not reference another node can be an inconvenience.

If you want to access the first node of a linked list after accessing the last node, you must resort to the *head* reference. Suppose that you change the right portion of the list's last node so that, instead of containing *null*, it references the first node. The result is a **circular linked list**, as illustrated in Figure 5-23. In contrast, the linked list you saw earlier is said to be a **linear linked list**.

Every node in a circular linked list references a successor, so you can start at any node and traverse the entire list. Although you could think of a circu-

Every node in a circular linked list has a successor

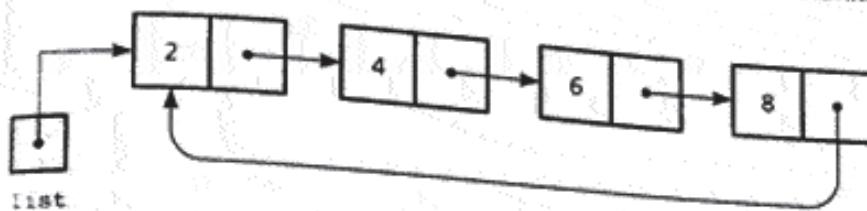


FIGURE 5-23

Circular linked list

not having either a beginning or an end, you still would have an external reference to one of the nodes in the list. Thus, it remains natural to think of a first and a last node in a circular list. If the external reference locates the "first" node, you still would have to traverse the list to get to the last node. However, if the external reference—call it *list*—references the "last" node, as it does in Figure 5-24, you can access both the first and last nodes without a traversal, because *list.next* references the first node.

A *null* value in the external reference indicates an empty list, as it did for a linear list. However, no node in a circular list contains *null* in its *next* reference. Thus, you must alter the algorithm for detecting when you have traversed an entire list. By simply comparing the current reference *curr* to the external reference *list*, you can determine when you have traversed the entire circular list. For example, the following Java statements display the data portions of every node in a circular list, assuming that *list* references the "last" node:

```
display the data in a circular linked list;
list references its last node
if (list != null) {
 // list is not empty
 Node first = list.next; // reference first node

 Node curr = first; // start at first node
```

No node in a circular linked list contains *null*

Write the data in a circular linked list

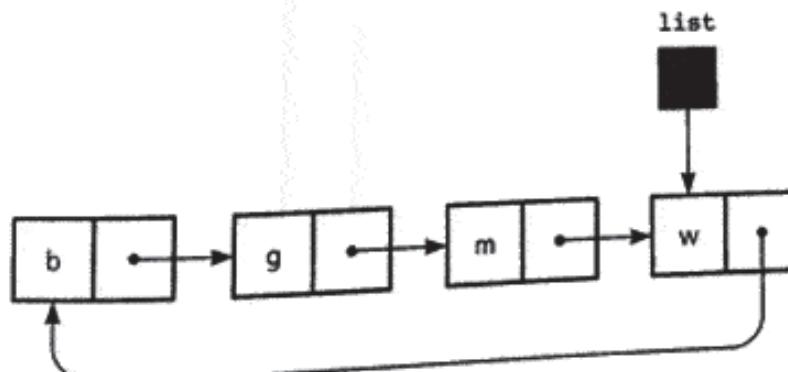


FIGURE 5-24

A circular linked list with an external reference to the last node

```

// Loop invariant: curr references next node to display
do {
 // write data portion
 System.out.println(curr.item);
 curr = curr.next; // reference next node
} while (curr != first); // list traversed?
} // end if

```

Operations such as insertion into and deletion from a circular linked list are left as exercises.

### Dummy Head Nodes

Both the insertion and deletion algorithms presented earlier for linear linked lists require a special case to handle action at the first position of a list. Many people prefer a method that eliminates the need for the special case. One such method is to add a **dummy head node**—as Figure 5-25 depicts—that is always present, even when the linked list is empty. In this way, the item at the first position of the list is actually in the second node. Also, the insertion and deletion algorithms initialize *prev* to reference the dummy head node, rather than *null*. Thus, for example, in the deletion algorithm, the statement

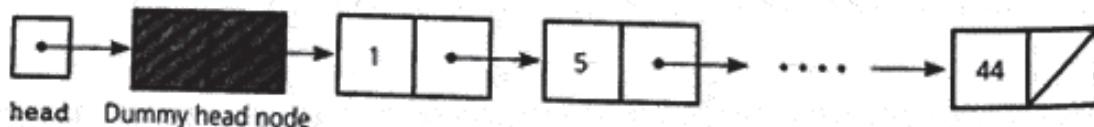
```
prev.next = curr.next;
```

deletes from the list the node that *curr* references, regardless of whether or not this node is the first element in the list.

Despite the fact that a dummy head node eliminates the need for a special case, handling the first list position separately can, in general, be less distracting than altering the list's structure by adding a dummy head node. However, dummy head nodes are useful with doubly linked lists, as you will see in the next section.

### Doubly Linked Lists

Suppose that you wanted to delete a particular node from a linked list. If you were able to locate the node directly without a traversal, you would not have established a trailing reference to the node that precedes it in the list. Without a trailing reference, you would be unable to delete the node. You could overcome this



**FIGURE 5-25**

A dummy head node

problem if you had a way to back up from the node that you wished to delete to the node that precedes it. A **doubly linked list** solves this problem because each of its nodes has references to both the next node and the previous node.

Consider a sorted linked list of customer names such that each node contains, in addition to its data field, two reference variables, *preceding* and *next*. As usual, the *next* reference of node *N* references the node that follows *N* in the list. The *preceding* data field references the node that precedes *N* in the list. Figure 5-26 shows the form of this sorted linked list of customers.

Notice that if *curr* references a node *N*, you can get a reference to the node that precedes *N* in the list by using the assignment statement

```
prev = curr.preceding;
```

A doubly linked list thus allows you to delete a node without traversing the list to establish a trailing reference.

Because there are more references to set, the mechanics of inserting into and deleting from a doubly linked list are a bit more involved than for a singly linked list. In addition, the special cases at the beginning or the end of the list are more complicated. It is common to eliminate the special cases by using a dummy head node. Although dummy head nodes may not be worthwhile for singly linked lists, the more complicated special cases for doubly linked lists make them very attractive.

As Figure 5-27a shows, the external reference *listHead* always references the dummy head node. Notice that the dummy head node has the same data type as the other nodes in the list; thus it also contains *preceding* and *next* references. You can link the list so that it becomes a **circular doubly linked list**. The *next* reference of the dummy head node then references the first "real node"—for example, the first customer name—in the list, and the *preceding* reference of the first real node refers back to the dummy head node. Similarly, the *preceding* reference of the dummy head node references the last node in the list, and the *next* reference of the last node references the dummy head node. Note that the dummy head node is present even when the list is empty. In this case, both reference variables of the dummy head node reference the head node itself, as Figure 5-27b illustrates.

Each node in a doubly linked list references both its predecessor and its successor

Dummy head nodes are useful in doubly linked lists

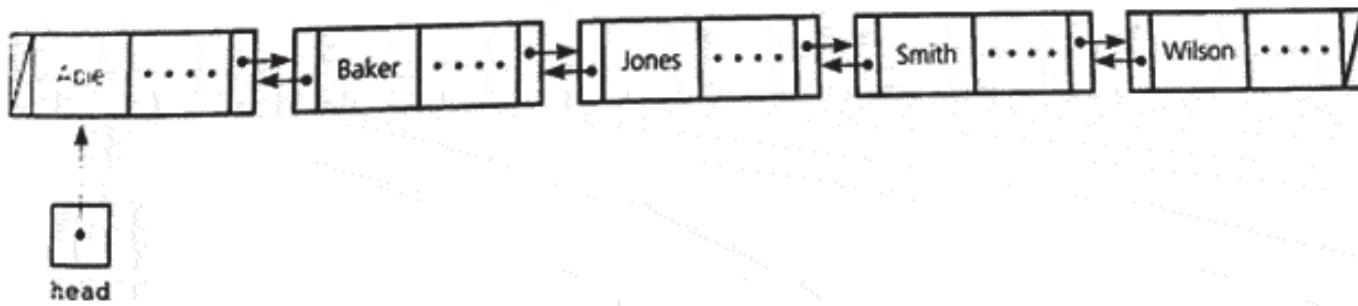


FIGURE 5-26

A doubly linked list

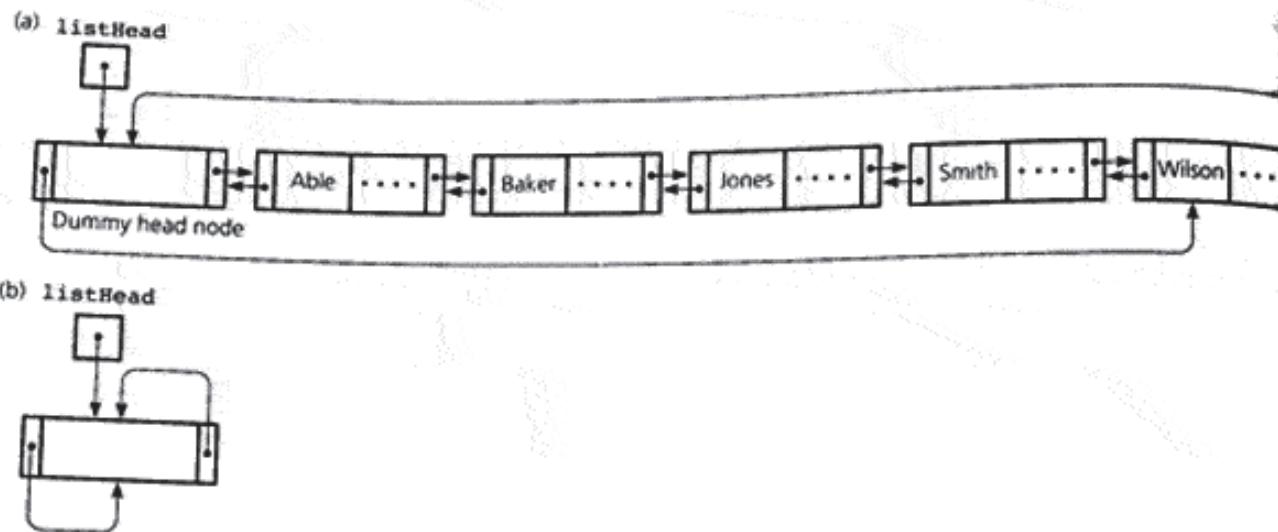


FIGURE 5-27

(a) A circular doubly linked list with a dummy head node; (b) an empty list with a dummy head node

A circular doubly linked list eliminates special cases for insertion and deletion

By using a circular doubly linked list, you can perform insertions and deletions without special cases: Inserting into and deleting from the first or last position is the same as for any other position. Consider, for example, how to delete the node  $N$  that  $curr$  references. As Figure 5-28 illustrates, you need to

1. Change the *next* reference of the node that precedes  $N$  so that it references the node that follows  $N$ .
2. Change the *preceding* reference of the node that follows  $N$  so that it references the node that precedes  $N$ .

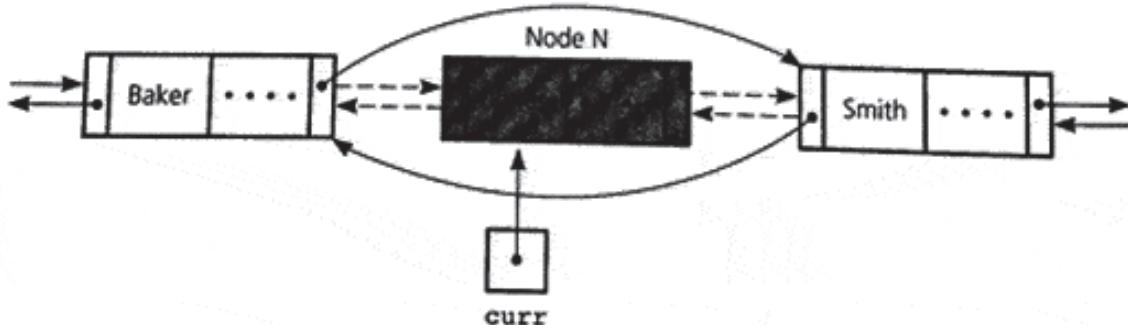


FIGURE 5-28

Reference changes for deletion

The following Java statements accomplish these two steps:

```
// delete the node that curr references
curr.preceding.next = curr.next;
curr.next.preceding = curr.preceding;
```

Deleting a node

You should convince yourself that these statements work even when the node to be deleted is the first, last, or only data (nonhead) node in the list.

Now consider how to insert a node into a circular doubly linked list. In general, the fact that the list is doubly linked does not mean that you avoid traversing the list to find the proper place for the new item. For example, if you insert a new customer name, you must find the proper place within the sorted linked list for the new node. The following pseudocode sets *curr* to reference the node that contains the first name greater than *newName*. Thus, *curr* will reference the node that is to follow the new node on the list:

```
/ find the insertion point
curr = listHead.next // reference first node, if any
while (curr != listHead and newName > curr.item) {
 curr = curr.next
} // end while
```

Traverse the list  
to locate the  
insertion point

Notice that if you want to insert the new node either at the end of the list or into an empty list, the loop will set *curr* to reference the dummy head node.

As Figure 5-29 illustrates, once *curr* references the node that is to follow the new node, you need to

1. Set the *next* reference in the new node to reference the node that is to follow it.
2. Set the *preceding* reference in the new node to reference the node that is to precede it.

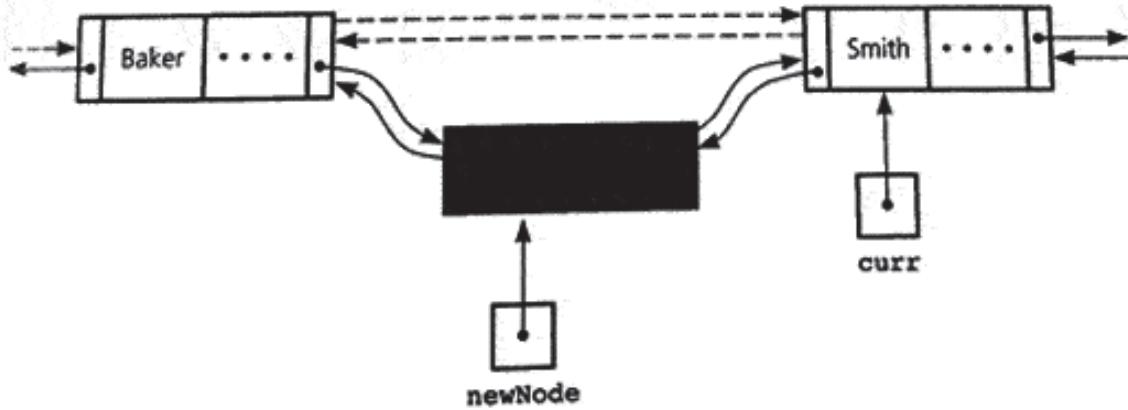


FIGURE 5-29

Reference changes for insertion

3. Set the *preceding* reference in the node that is to follow the new node so that it references the new node.
4. Set the *next* reference in the node that is to precede the new node so that it references the new node.

The following Java statements accomplish these four steps, assuming that *newNode* references the new node:

#### Inserting a node

```
// insert the new node that newNode references before
// the node referenced by curr
newNode.next = curr;
newNode.preceding = curr.preceding;
curr.preceding = newNode;
newNode.preceding.next = newNode;
```

You should convince yourself that these statements work even when you insert the node into the beginning of a list; at the end of a list, in which case *curr* references the head node; or into an empty list, in which case *curr* also references the head node.

## 5.4 Application: Maintaining an Inventory

Imagine that you have a part-time job at the local movie rental store. Realizing that you know a good deal about computers, the store owner asks you to write an interactive program that will maintain the store's inventory of DVDs that are for sale. The inventory consists of a list of movie titles and the following information associated with each title:

- **Have value:** number of DVDs currently in stock.
- **Want value:** number of DVDs that should be in stock. (When the have value is less than the want value, more DVDs are ordered.)
- **Wait list:** list of names of people waiting for the title if it is sold out.

Because the owner plans to turn off the power to the computer when the store is closed, your inventory program will not be running at all times. Therefore, the program must save the inventory in a file before execution terminates and later restore the inventory when it is run again.

Program input and output are as follows:

#### Input

- A file that contains a previously saved inventory.
- A file that contains information on an incoming shipment of DVDs. (See command D.)
- Single-letter commands—with arguments where necessary—that inquire about or modify the inventory and that the user will enter interactively.

**Output**

A file that contains the updated inventory. (Note that you remove from the inventory all items whose have values and want values are zero and whose wait lists are empty. Thus, such items do not appear in the file.)

Output as specified by the individual commands.

The program should be able to execute the following commands:

|           |            |                                                                                                                                                                                                                                                                                                                                                                                | Program commands |
|-----------|------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------|
| H         | (help)     | Provide a summary of the available commands.                                                                                                                                                                                                                                                                                                                                   |                  |
| <title>   | (inquire)  | Display the inventory information for a specified title.                                                                                                                                                                                                                                                                                                                       |                  |
| L         | (list)     | List the entire inventory (in alphabetical order by title).                                                                                                                                                                                                                                                                                                                    |                  |
| A <title> | (add)      | Add a new title to the inventory. Prompt for initial want value.                                                                                                                                                                                                                                                                                                               |                  |
| M <title> | (modify)   | Modify the want value for a specified title.                                                                                                                                                                                                                                                                                                                                   |                  |
| D         | (delivery) | Take delivery of a shipment of DVDs, assuming that the clerk has entered the shipment information (titles and counts) into a file. Read the file, reserve DVDs for the people on the wait list, and update the have values in the inventory accordingly. Note that the program must add an item to the inventory if a delivered title is not present in the current inventory. |                  |
| O         | (order)    | Write a purchase order for additional DVDs based on a comparison of the have and want values in the inventory, so that the have value is brought up to the want value.                                                                                                                                                                                                         |                  |
| R         | (return)   | Write a return order based on a comparison of the have and want values in the inventory and decrease the have values accordingly (make the return). The purpose is to reduce the have value to the want value.                                                                                                                                                                 |                  |
| S <title> | (sell)     | Decrease the count for the specified title by 1. If the title is sold out, put a name on the wait list for the title.                                                                                                                                                                                                                                                          |                  |
| Q         | (quit)     | Save the inventory and wait lists in a file and terminate execution.                                                                                                                                                                                                                                                                                                           |                  |

The problem-solving process that starts with a statement of the problem and ends with a program that effectively solves the problem—that is, a program that meets its specification—has three main stages:

1. The design of a solution
2. The implementation of the solution
3. The final set of refinements to the program

Realize, however, that you cannot complete one stage in total isolation from the others. Also realize that at many steps in the development of a solution, you must make choices. Although the following discussion may give the impression that the choices are clear-cut, this is not always the case. In reality, both the trade-offs between choices and the false starts (wrong choices considered) are often numerous.

This problem primarily involves data management and requires certain program commands. These commands suggest the following operations on the inventory:

- List the inventory in alphabetical order by title (L command).
- Find the inventory item associated with a title (I, M, D, O, and S commands).
- Replace the inventory item associated with a title (M, D, R, and S commands).
- Insert new inventory items (A and D commands).

Recall that each title might have an associated wait list of people who are waiting for that title. You must be able to

- Add new people to the end of the wait list when they want a DVD that is sold out (S command).
- Delete people from the beginning of the wait list when new DVDs are delivered (D command).
- Display the names on a wait list for a particular title (I and L commands).

In addition, you must be able to

- Save the current inventory and associated wait lists when program execution terminates (Q command).
- Restore the current inventory and associated wait lists when program execution begins again.

You could think of these operations as part of an ADT inventory. Your next step should be to specify each of the operations fully. Since this chapter is about linked lists and implementation issues, the completion of the specifications will be left as an exercise. We will turn our attention to a data structure that could implement the inventory.

Each data item in the ADT inventory represents a movie and contains a title, the number of DVDs in stock (a have value), the number desired (a want value), and a wait list. How will you represent the wait list? First, you need to decide what information you want to store in the wait list. For example, you might want to keep track of the full name and phone number of each person, and create a class *Customer* containing data fields for the first and last names along with the telephone number of the person. This class might be structured as follows:

A customer in the  
wait list

```
public class Customer {
 private String lastName;
```

```
private String firstName;
private String phone;

public Customer(String first, String last, String phone) {
 to be implemented
 ...
 end constructor

public String toString() {
 to be implemented
 ...
 end toString
end class Customer
```

This definition contains the minimum number of methods required to use instances of the *Customer* class in our inventory problem. You may also decide if you want to keep additional information about a person, such as their address. The *toString* method is provided for printing purposes.

Now that you have decided what information to keep in the wait list, will you implement the wait list itself in the ADT inventory? Could you use any of the implementations of the ADT list we developed previously? To make this decision, you must review the requirements of the wait list stated in the inventory problem and then see if the ADT list will be able to meet these requirements. The inventory problem requires you to be able to add to the end of the wait list and delete from the beginning of the wait list. Clearly, removing an item from the beginning of the list is easy: You can simply use the ADT list operation *remove* with an index value of 1. Adding an item to the end of the list is also fairly easy. You know the size of the list from the ADT list operation *size()*, and you could use the ADT list operation *add* with an index value of *size() + 1* to place an item at the end of the wait list.

One of the requirements of the inventory problem is that the L command sort the inventory in alphabetical order by movie title. Will you be able to use the ADT list in a way that will support this requirement? Not easily, since the ADT list is based on index position, not on a sorted order. A better choice is the ADT sorted list. Not only does it maintain the data in a sorted order for you, but it also provides an operation *locateIndex(item)* that can be used to search for an item in the sorted list.

If the ADT sorted list is not yet implemented, should you use an array-based implementation or a reference-based implementation? If you use an array to contain the items, you can use a binary search. Inserting and deleting items, however, requires you to shift array elements. Using a linked list for the items avoids these data shifts but makes a binary search impractical. (How do you quickly locate the middle item in a linked list?) Weighing these trade-offs, we choose a linked list to implement the ADT sorted list.

The ADT sorted list is the best choice for data that must be maintained in alphabetical order.

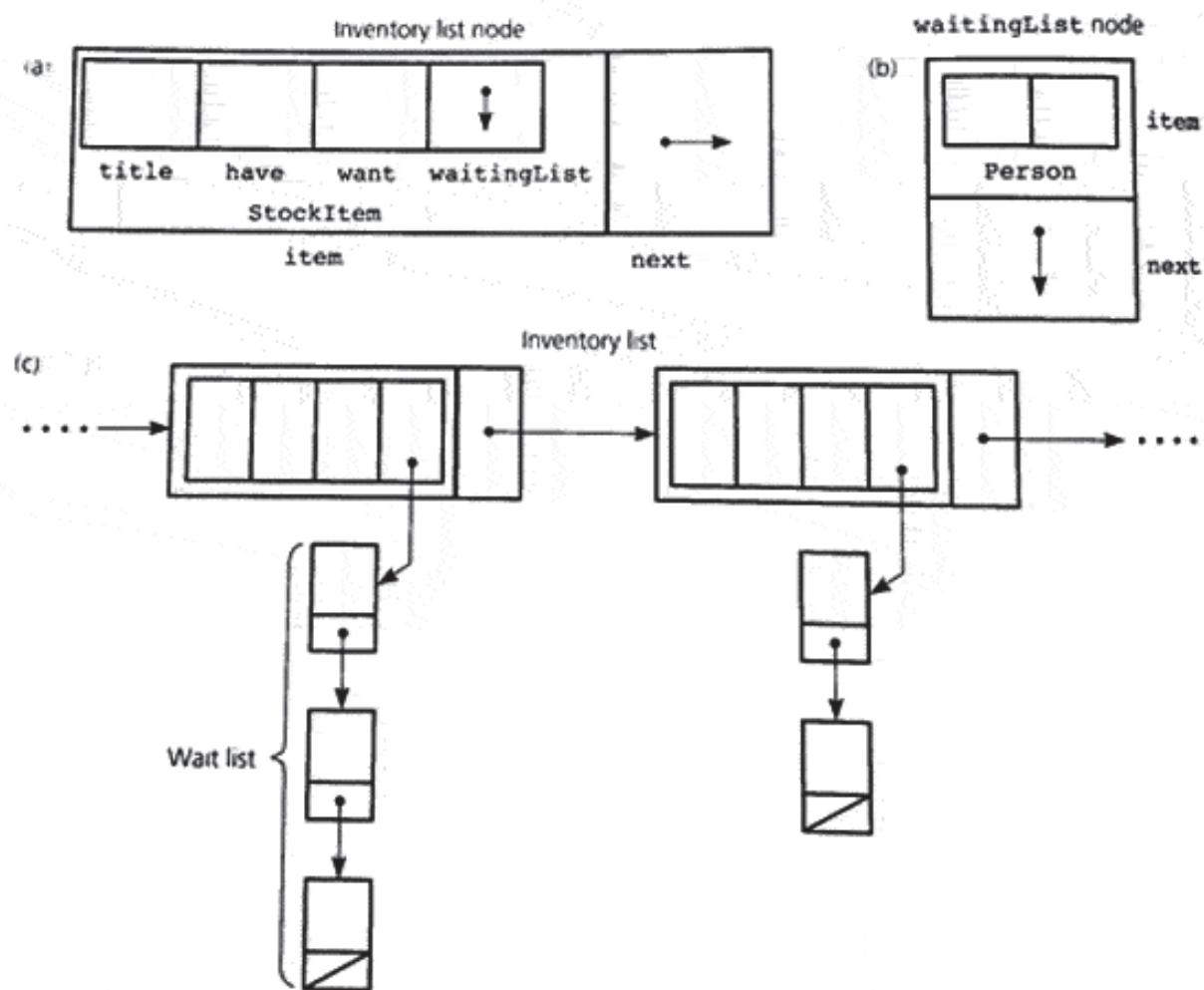
A sorted list  
represents the  
inventory

- To summarize, we have made the following choices:
    - The inventory is a sorted list of data items (the ADT sorted list implemented as a linked list of data items), sorted by the title that each item represents.
    - Each inventory item contains a title, a have value, a want value, and a list of customers (the wait list).

Figure 5-30 and the following Java statements summarize these choices:

```
public class StockItem implements java.lang.Comparable {
 private String title;
 private int have, want;
 private ListReferenceBased waitingList;

 // various constructors for StockItem
 ...
}
```



**FIGURE 5-30**

(a) Inventory list node; (b) wait list node; (c) orthogonal structure for the inventory

```
public void addToWaitingList(String lastName,
 String firstName, String phone) {
 // add a person to the waiting list
 waitingList.addSorted(
 new Customer(lastName, firstName, phone));
} // end addToWaitingList

public String toString() {
 // for displaying StockItem instances
 ...
} // end toString

public int compareTo(Object rhs) {
 // define how StockItems are compared, only by title
 return title.compareTo(((StockItem)rhs).title);
} // end compareTo

// mutator and accessor methods for other data fields
...
}

// end class StockItem
```

You declare the inventory as follows:

```
SortedList inventory = new SortedList();
```

Before you can proceed with the implementation, you must consider how you will save the inventory in a file. Java provides **object serialization**, a process that transforms an object into a stream of bytes that you can save and restore from a file. The most powerful aspect of object serialization is that when you write any object to a file, any other objects that are referenced by that object are also written to the file. As mentioned in Chapter 1, to enable this feature, you place an *implements Serializable* clause in each class that has instances that will be written to the file. Thus, to write the inventory successfully, you would include the clause in the classes *ListReferenceBased*, *Node*, *SortedList*, *StockItem*, and *Customer*. Then, when you write an inventory object to a file, all of the stock items in the inventory list and their wait lists are also be placed in the file. Here is the code that accomplishes that task:

```
try {
 FileOutputStream fos = new
 FileOutputStream("inventory.dat");
 ObjectOutputStream oos = new ObjectOutputStream(fos);
 oos.writeObject(inventory);
 fos.close();
} // end try
catch (Exception e) {
```

```

 System.out.println(e);
 } // end catch
}

```

Restoring the inventory is also straightforward:

```

ListReferenceBased restoredInventory;
try {
 FileInputStream fis = new
 FileInputStream("inventory.dat");
 ObjectInputStream ois = new ObjectInputStream(fis);
 Object o = ois.readObject();
 restoredInventory = (ListReferenceBased) o;
 System.out.println(restoredInventory);
} // end try
catch (Exception e) {
 System.out.println(e);
} // end catch
}

```

The completion of this solution is left as an exercise.

## 5.5 The Java Collections Framework

Many modern programming languages, such as Java, provide classes that implement many of the more commonly used ADTs. In Java, many of these classes are defined in the Java Collections Framework or JCF. The JCF contains a number of classes and interfaces that can be applied to nearly any type of data.

Many of the ADTs that are presented in this text have a corresponding class or interface in the JCF. For example, a *List* interface is defined in the JCF that is similar to the *ListInterface* specification presented earlier in this chapter. You may be wondering why we spend so much time developing ADTs in this text if they are already provided in the JCF. There are many reasons for doing so; here are just a few:

- Developing simple ADTs provides a foundation for learning other ADTs.
- You may find yourself working in a language that does not provide any predefined ADTs. You need to have the ability to develop ADTs on your own, and hence understand the process.
- If the ADTs defined by the language you are using are not sufficient, you may need to develop your own or enhance existing ones.

A collections framework is a unified architecture for representing and manipulating collections. It includes *interfaces*, or ADTs representing collections; *implementations*, or concrete implementations of collection interfaces; and *algorithms*, or methods that perform useful computations, such as sorting and searching, on objects that implement collection interfaces. These algorithms are *polymorphic* because the same method can be used on many different implementations of the appropriate collections interface.

The Java Collections Framework (JCF) provides classes for common ADTs

A collections framework includes interfaces, implementations, and algorithms

The JCF also contains iterators. Iterators provide a way to cycle through the contents of a collection. Before we can discuss the JCF further, we will give a brief overview of generics and iterators.

## Generics

The JCF relies heavily on Java generics. Generics allow you to develop classes and interfaces and defer certain data-type information until you are actually ready to use the class or interface. For example, our list interface was developed independently from the type of the list items by using the *Object* class. With generics, this data type is left as a data-type parameter in the definition of the class or interface. The start of the definition of the class or interface is followed by `<E>`, where the data-type parameter *E* represents the data type that the client code will specify. Here is an example of a simple generic class:

Generic classes allow data-type information to be deferred

```
public class MyClass<E> {
 private E theData;
 private int n;

 public MyClass() {
 n = 0;
 } // end constructor

 public MyClass(E initData, int num) {
 n = num;
 theData = initData;
 } // end constructor

 public void setData(E newData) {
 theData = newData;
 } // end setData

 public E getData() {
 return theData;
 } // end getData

 public int getNum() {
 return n;
 } // end getNum
} // end MyClass
```

Chapter 9 describes in more detail how to create your own generics.

When you (the client) declare instances of the class, you specify the actual data type that the parameter represents. This data type cannot be a primitive type, only object types are allowed. For example, a simple program that uses this generic class could begin as follows:

Only object types are allowed for data type parameters

```

static public void main(String[] args) {
 MyClass<String> a = new MyClass<String>();
 Double d = new Double(6.4);
 MyClass<Double> b = new MyClass<Double>(d, 51);

 a.setData("Sarah");
 System.out.println(a.getData() + ", " + b.getData());
 System.out.println(a.getNum() + ", " + b.getNum());
}

```

Notice how the declarations of *a* and *b* specify the data type of *MyClass*'s data member *theData*. Also note that when we previously used *Object* as the return type, we often had to cast the result back to the desired type that is no longer required when using generics.

## Iterators

Iterators are used to cycle through items in a collection

An **iterator** is an object that gives you the ability to cycle through the items in a collection in much the same way that we used a reference to traverse a linked list. If you have an iterator call *iter*, you can access the next item in the collection by using the notation *iter.next()*.

The JFC provides two primary iterator interfaces, *java.util.Iterator* and *java.util.ListIterator*. Note that all interface methods are implicitly public, so the *Iterator* interface is defined as follows:

```

public interface Iterator<E> {
 boolean hasNext();
 // Returns true if the iteration has more elements.

 E next();
 // Returns the next element in the iteration.

 void remove() throws UnsupportedOperationException,
 IllegalStateException;
 // Removes from the underlying collection the last
 // element returned by the iterator (optional
 // operation).
} // end Iterator

```

The method *next* is used to return the next element in the collection. When an iterator is initially created, it is positioned so that the first call to *next* on the iterator object will return the initial element in the collection. The method *hasNext* can be used to determine if another element is available in the collection.

Notice that one of the operations, *remove*, can throw the exception *UnsupportedOperationException*. The expectation is that the *remove* operation will simply throw this exception if the operation is not available in the class that implements the interface.

Unsupported iterator methods will throw an exception

Iterators are an integral part of all of the classes and interfaces used for representing collections in the JCF. Note that just as you can use inheritance to derive new classes, you can use inheritance to derive new interfaces, often called **subinterfaces**. The basis for the ADT collections in the JCF is the interface `java.util.Iterable`, with the subinterface `java.util.Collection`:

```
public interface Iterable<E> {
 Iterator<E> iterator();
 Returns an iterator over the elements in this collection
} // end Iterable

public interface Collection<E> extends Iterable<E> {
 Only a portion of the Collection interface is shown here.
 See the J2SE documentation for a complete listing of
 methods

 boolean add(E o);
 Ensures that this collection contains the specified
 element (optional operation).

 boolean remove(Object o);
 Removes a single instance of the specified element from
 this collection, if it is present (optional operation).

 void clear();
 // Removes all of the elements from this collection
 // (optional operation).

 boolean contains(Object o);
 // Returns true if this collection contains the specified
 // element.

 boolean equals(Object o);
 // Compares the specified object with this collection for
 // equality.

 boolean isEmpty()
 // Returns true if this collection contains no elements.

 int size();
 // Returns the number of elements in this collection.

 Object[] toArray();
 // Returns an array containing all of the elements in this
 // collection.
} // end Collection
```

Thus, every ADT collection in the JCF will have a method to return an iterator object for the underlying collection. The following example shows how an iterator can be used with the JCF list class *LinkedList*:

```
import java.util.LinkedList;
import java.util.Iterator;

public class TestLinkedList {
 static public void main(String[] args) {
 LinkedList<Integer> myList = new LinkedList<Integer>();

 Iterator iter = myList.iterator();
 if (!iter.hasNext()) {
 System.out.println("The list is empty");
 } // end if

 for (int i=1; i <= 5; i++) {
 myList.add(new Integer(i));
 } // end for

 iter = myList.iterator();
 while (iter.hasNext()) {
 System.out.println(iter.next());
 } // end while
 } // end main
} // end TestLinkedList
```

The behavior of an iterator is unspecified if the underlying collection is modified while the iteration is in progress in any way other than by calling the *remove* method.

Another example of a subinterface in the JCF is *java.util.ListIterator*, derived from the *java.util.Iterator* interface:

```
public interface ListIterator<E> extends Iterator<E> {
 void add(E o);
 // Inserts the specified element into the list (optional
 // operation).

 boolean hasNext();
 // Returns true if this list iterator has more elements when
 // traversing the list in the forward direction.

 boolean hasPrevious();
 // Returns true if this list iterator has more elements when
 // traversing the list in the reverse direction.
```

```

E next();
 // Returns the next element in the list.

int nextIndex();
 // Returns the index of the element that would be returned
 // by a subsequent call to next.

E previous();
 // Returns the previous element in the list.

int previousIndex();
 // Returns the index of the element that would be returned
 // by a subsequent call to previous.

void remove();
 // Removes from the list the last element that was
 // returned by next or previous (optional operation).

void set(E o);
 // Replaces the last element returned by next or previous
 // with the specified element (optional operation).
} // end ListIterator

```

The *ListIterator* interface extends *Iterator* by providing support for bidirectional access to the collection as well as adding or changing elements in the collection. A bidirectional iterator enables you to move to either the next or previous element in the collection.

Chapter 9 describes how to create your own iterator.

**Bidirectional iterators** allow you to move forward or back through a collection

## The Java Collection's Framework *List* Interface

The JCF provides an interface *java.util.List* that is quite similar to the list interface created in Chapter 4. The JCF *List* interface supports an ordered collection, also known as a sequence. Like the *ListInterface* presented in this text, users can specify by position (integer index) where elements are added to and removed from the list, and the position numbering starts at zero (as in *ListInterface*). Though the interface provides methods based upon positional access to the elements, the time to execute these methods may be proportional to the index value, depending on the implementing class. As such, it is usually preferable to use an iterator instead of index access when possible to locate and process elements in a list.

Declarations for all the methods in the *List* interface are shown here, even the ones inherited from the *Collection* interface. Notice that the methods *iterator*, *add*, *remove*, and *equals* place additional stipulations beyond those specified in the *Collection* interface. The *List* interface also provides other methods not shown here that allow multiple elements to be inserted and removed at any point in the list.

The JCF *List* interface supports an ordered collection

The *List* interface inherits methods from the *Collection* interface

Notice that the *List* interface provides a *ListIterator* that allows bidirectional access in addition to the normal operations that the *Iterator* interface provides. There is also a method to obtain a list iterator that starts at a specified position in the list.

The JCF *List* interface is derived from the JCF *Collection* interface:

```
public interface List<E> extends Collection<E>
 // Only a portion of the List interface is shown here.
 // See the J2SE documentation for a complete listing of
 // methods

 boolean add(E o);
 // Appends the specified element to the end of this list
 // (optional operation).

 void add(int index, E element);
 // Inserts the specified element at the specified position in
 // this list (optional operation).

 void clear();
 // Removes all of the elements from this list (optional
 // operation).

 boolean contains(Object o);
 // Returns true if this list contains the specified element.

 boolean equals(Object o);
 // Compares the specified object with this list for equality.

 E get(int index);
 // Returns the element at the specified position in this
 // list.

 int indexOf(Object o);
 // Returns the index in this list of the first occurrence of
 // the specified element, or -1 if this list does not contain
 // this element.

 boolean isEmpty();
 // Returns true if this list contains no elements.

 Iterator<E> iterator();
 // Returns an iterator over the elements in this list in
 // proper sequence.

 ListIterator<E> listIterator();
 // Returns a list iterator of the elements in this list (in
 // proper sequence).
```

```
ListIterator<E> listIterator(int index);
 // Returns a list iterator of the elements in this list (in
 // proper sequence), starting at the specified position in
 // this list.

 E remove(int index);
 // Removes the element at the specified position in this list
 // (optional operation).

 boolean remove(Object o);
 // Removes the first occurrence in this list of the specified
 // element (optional operation).

 E set(int index, E element);
 // Replaces the element at the specified position in this
 // list with the specified element (optional operation).

 int size();
 // Returns the number of elements in this list.

 List<E> subList(int fromIndex, int toIndex);
 // Returns a view of the portion of this list between the
 // specified fromIndex, inclusive, and toIndex,
 // exclusive.

 Object[] toArray();
 // Returns an array containing all of the elements in this
 // list in proper sequence.

/// end List
```

The JCF provides numerous classes that implement the `List` interface, including `LinkedList`, `ArrayList`, and `Vector`. Here is an example of how the JCF class `ArrayList` is used to maintain a grocery list:

```
import java.util.ArrayList;
import java.util.Iterator;

public class GroceryList {

 static public void main(String[] args) {
 ArrayList<String> groceryList = new ArrayList<String>();
 Iterator<String> iter;

 groceryList.add("apples");
 groceryList.add("bread");
 groceryList.add("juice");
```

```

 groceryList.add("carrots");
 groceryList.add("ice cream");

 System.out.println("Number of items on my grocery list: " +
 + groceryList.size());
 System.out.println("Items are: ");
 iter = groceryList.listIterator();
 while (iter.hasNext()) {
 String nextItem = iter.next();
 System.out.println(groceryList.indexOf(nextItem)+" " +
 + nextItem);
 } // end while

 } // end main

} // end GroceryList

```

The output of this program is

```

Number of items on my grocery list: 5
Items are:
0) apples
1) bread
2) juice
3) carrots
4) ice cream

```

Clearly it is more efficient to use a counter to number the items than to use the method *indexOf*; it was done for illustrative purposes.

## Summary

---

1. You can use reference variables to implement the data structure known as a linked list by using a class definition such as the following:

```

package List;

class Node {
 Object item;
 Node next;

 Node(Object newItem) {
 item = newItem;
 next = null;
 } // end constructor
}

```

```
Node (Object newItem, Node nextNode) {
 item = newItem;
 next = nextNode;
} // end constructor
end class Node
```

- Each reference in a linked list is a reference to the next node in the list. For example, if `nodeRef` is a variable of type `Node` that references a node in this linked list,
  - `nodeRef.item` is the data portion of the node.
  - `nodeRef.next` references the next node.
- Algorithms for inserting data into and deleting data from a linked list both involve these steps: Traverse the list from the beginning until you reach the appropriate position, perform reference changes to alter the structure of the list. In addition, you use the `new` operator to dynamically allocate a new node for insertion. When all references to a node are removed, the node is automatically marked for garbage collection.
- Inserting a new node at the beginning of a linked list or deleting the first node of a linked list are cases that you treat differently from insertions and deletions anywhere else in the list.
- An array-based implementation uses an implicit ordering scheme—for example, the item that follows `anArray[i]` is stored in `anArray[i+1]`. A reference-based implementation uses an explicit ordering scheme—for example, to find the item that follows the one in node *N*, you follow node *N*'s reference.
- You can access any element of an array directly, but you must traverse a linked list to access a particular node. Therefore, the access time for an array is constant, whereas the access time for a linked list depends upon the location of the node within the list.
- You can insert items into and delete items from a reference-based linked list without shifting data. This characteristic is an important advantage of a linked list over an array.
- Although you can use the `new` operator to allocate memory dynamically for either an array or a linked list, you can increase the size of a linked list one node at a time more efficiently than an array. When you increase the size of a resizable array, you must copy the original array elements into the new array and then deallocate the original array.
- A binary search of a linked list is impractical because you cannot quickly locate its middle item.
- You can use recursion to perform operations on a linked list. Such use will eliminate special cases and the need for a trailing reference.
- 11 The recursive insertion algorithm for a sorted linked list works because each smaller linked list is also sorted. When the algorithm makes an insertion at the beginning of one of these lists, the inserted node will be in the proper position in the original list. The algorithm is guaranteed to terminate because each smaller list contains one fewer node than the preceding list and because the empty list is a base case.

12. A tail reference can be used to facilitate locating the end of a list. This is especially useful when an append operation is required.
13. In a circular linked list, the last node references the first node, so that every node has a successor. If the list's external reference references the last node instead of the first node, you can access both the last node and the first node without traversing the list.
14. Dummy head nodes provide a method for eliminating the special cases for insertion into and deletion from the beginning of a linked list. The use of dummy head nodes is a matter of personal taste for singly linked lists, but it is helpful for a doubly linked list.
15. A doubly linked list allows you to traverse the list in either direction. Each node references its successor as well as its predecessor. Because insertions and deletions with a doubly linked list are more involved than with a singly linked list, it is convenient to use both a dummy head node and a circular organization to eliminate complicated special cases for the beginning and end of the list.
16. If you plan on storing the data contained in a linked list to a file, be sure to place the `implements Serializable` clause in each class that has instances that will be written to the file.
17. A generic class or interface enables you to defer the choice of certain data-type information until its use.
18. The Java Collections Framework contains interfaces, implementations, and algorithms for many common ADTs.
19. A collection is an object that holds other objects. An iterator cycles through the contents of a collection.

### Cautions

1. An uninitialized reference variable has the value `null`. Attempting to use a reference with a value of `null` will cause a `NullPointerException` to be thrown.
2. The sequence

```
Integer intRef = new Integer(5);
intRef = null;
```

allocates a memory cell and then destroys the only means of accessing it. Do not use `new` when you simply want to assign a value to a reference.

3. Insertions into and deletions from the beginning of a linked list are special cases unless you use a dummy head node. Failure to recognize this fact can result in a `null` reference being used, causing a `NullPointerException` to be thrown.

- 4 When traversing a linked list by using the reference variable `curr`, you must be careful not to reference `curr` after it has "passed" the last node in the list, because it will have the value `null` at that point. For example, the loop

```
while (value > curr.item)
 curr = curr.next;
```

is incorrect if `value` is greater than all the data values in the linked list, because `curr` becomes `null`. Instead, you should write

```
while ((curr != null) && (value > curr.item))
 curr = curr.next;
```

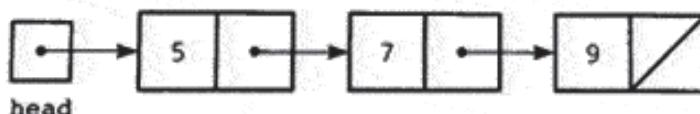
Because Java uses short-circuit evaluation of logical expressions, if `curr` becomes `null`, the expression `curr.item` is not evaluated.

- 5 A doubly linked list is a data structure that programmers tend to overuse. However, a doubly linked list is appropriate to use when you have direct access to a node. In such cases, you would not have traversed the list from its beginning. If the list were singly linked, you would not have a reference to the preceding node. Because doubly linking the list provides an easy way to get to the node's predecessor as well as its successor, you can, for example, delete the node readily.

## Self-Test Exercises

---

- 1 Given the following declarations, and the list shown, draw a picture which shows the result of each sequence of statements given below. If something illegal is done (as noted by the compiler), circle the offending statement and explain why it is illegal. Assume they all begin with this list:



```
package IntegerList;

class IntegerNode {
 int item;
 IntegerNode next;

 Node (Object newItem) {
 item = newItem;
 next = null;
 } // end constructor

 Node(Object newItem, Node nextNode) {
 item = newItem;
 next = nextNode;
 } // end constructor
} // end IntegerNode
```

```
IntegerNode head, p, q;
int x;
```

|                                                                                                             |                                                                                       |
|-------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------|
| a. p = new IntegerNode();                                                                                   | b. p = new IntegerNode(1, head);                                                      |
| c. p = new IntegerNode(1);     q = new IntegerNode(3, p);     p.next = head;     head = q;                  | d. x = 3;     p = new IntegerNode(x, head);     q = new IntegerNode(p);     head = q; |
| e. IntegerNode curr = head;     while (curr != null) {         curr.item++;         curr = curr.next;     } | f. x = 3;     p = new IntegerNode(x,     head.next);     head = p;                    |

2. Consider the algorithm for deleting a node from a linked list that this chapter describes.
- Is the deletion of the first node of a linked list a special case? Explain.
  - Is deletion of the last node of a linked list a special case? Explain.
  - Is deletion of the only node of a one-node linked list a special case? Explain.
  - Does deleting the first node take more effort than deleting the last node? Explain.
3. a. Write Java statements that create the linked list pictured in Figure 5-31, as follows. Beginning with an empty linked list, first create and attach a node for K, then create and attach a node for M, and finally create and attach a node for S.
- b. Repeat Part a, but instead create and attach nodes in the order B, E, J.
4. Consider the sorted linked list of single characters in Figure 5-31. Suppose that *prev* references the first node in this list and *curr* references the second node.
- Write Java statements that delete the second node. (*Hint:* First modify Figure 5-31.)
  - Now assume that *curr* references the first node of the remaining two nodes of the original list. Write Java statements that delete the last node.
  - Now *head* references the only node that is left in the list. Write Java statements that insert a new node that contains J into the list so that the list remains sorted.
  - Revise Figure 5-31 so that your new diagram reflects the results of the previous deletions and insertion.

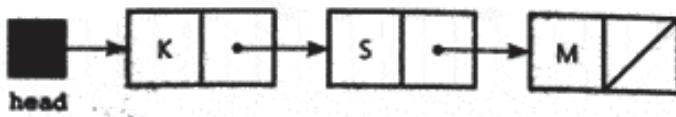
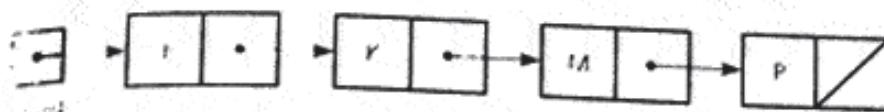


FIGURE 5-31

## Self-Test Exercises 3, 4, and 7

**FIGURE 5-32**

Consider Exercise 1.

There were many types of linked lists discussed in the chapter. What was common in the nodes used in all of these classes?

How many assignment operations does the method that you wrote for Self-Test Exercise 5 require?

Do a box trace of `writeBackwards2(head)`, where `head` references the linked list of characters pictured in Figure 5-31. Show which node `head` points to in each recursive call. The method `writebackwards2` appears on page 273 of this chapter.

## Exercises

1. For each of the following, write the Java statements that perform the requested operation on the list shown in Figure 5-32. Also draw a picture of the status of the list after each operation is complete. When you delete a node from the list, make sure it will eventually be returned to the system. All insertions into the list should maintain the list's sorted order. Do not use any of the methods that were presented in this chapter.
  - a. Assume that `prev` references the first node and `curr` references the second node. Insert L into the list.
  - b. Assume that `prev` references the second node and that `curr` references the third node of the list after you revised it in Part a. Delete the last node of the list.
  - c. Assume that `prev` references the last node of the list after you revised it in Part b, and assume that `curr` is `null`. Insert Q into the list.
2. Consider a linked list of items that are in no particular order.
  - a. Write a method that inserts a node at the beginning of the linked list and a method that deletes the first node of the linked list.
  - b. Repeat Part a, but this time perform the insertion and deletion at the end of the list instead of at the beginning. Assume the list has only a head reference.
  - c. Repeat Part b, but this time assume that the list has a tail reference as well as a head reference.
3. Write a method that randomly removes and returns an item from a linked list. Write the method such that
  - a. the method uses only the ADT List operations; that is, it is independent of the list's implementation.

- b. the method assumes and uses the reference-based implementation of the ADT *List*.
  - c. the method assumes and uses the array-based implementation of the ADT *List*.
4. Given the following *Student* class:

```
class Student {
 private String name;
 private int age;

 public Student(String n, int a) {
 name = n;
 age = a;
 } // end constructor

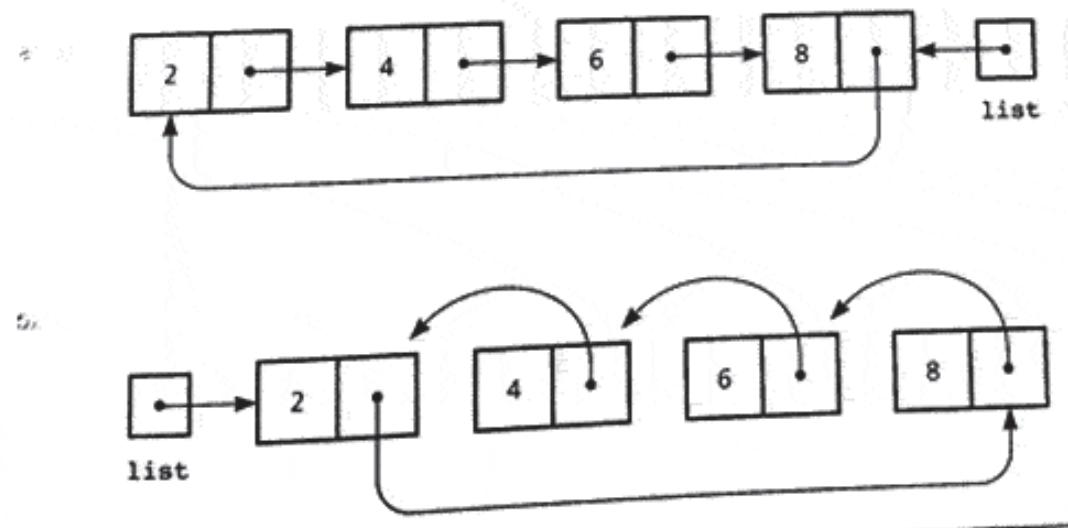
 public String getName() {
 return name;
 } // end getName

 public int getAge() {
 return age;
 } // end getAge
} // end Student
```

Write a Java method that displays only the name of the  $i^{\text{th}}$  student in a linked list of students. Assume that  $i \geq 0$  and that the linked list contains at least  $i$  nodes.

- 5. Using the *Student* class in Exercise 4, write a recursive version of the method that displays (a) the name of the  $i^{\text{th}}$  student in a linked list of students. Assume that  $i \geq 1$  and that the linked list contains at least  $i$  nodes. (Hint: If  $i = 0$ , print the name of the first student in the list; otherwise, print the  $(i - 1)^{\text{th}}$  student name from the rest of the list.)
- 6. The section "Processing Linked Lists Recursively" discussed the traversal of a linked list.
  - a. Compare the efficiencies of an iterative method that displays a linked list with the method *writeList*.
  - b. Write an iterative method that displays a linked list backward. Compare the efficiencies of your method with the method *writeListBackward2*.
- 7. Write a method to merge two linked lists of integers that are sorted into descending order. The result should be a third linked list that is the sorted combination of the original lists. Do not destroy the original lists.
- 8. The *Node* class presented in this chapter assumed that it would be declared package-private; hence, the data fields were declared for package access only.
  - a. Suppose that the data fields were declared *private*. Write accessor and mutator methods for both the *item* and *next* fields.
  - b. Give at least three different examples of how the code in the *ListReferenceBased* implementation would have to be changed.
- 9. Assume that the reference *list* references the last node of a circular linked list like the one in Figure 5-24. Write a loop that searches for an item in the list and if

- found, returns its position. If it is not found, return -1. Assume that the node referenced by `list.next` is the node in the first position.
10. Write the pseudocode for a method that inserts a new node to the end of a doubly linked list.
  11. Write the method `add(int index, Object item)` from the interface `ListInterface` for a doubly linked list with a head pointer as depicted in Figure 5-26.
  12. Write a class called `SantasList` that allows Santa to keep track of all of the children that are naughty and nice. Santa should be able to add children to either of the lists, and the lists can be maintained in any order. Also provide methods to print each list.
    - a. Implement `SantasList` using your own linked list.
    - b. Implement `SantasList` using the `ListReferenceBased` class.
    - c. What changes would you need to make if you wanted to change the implementation in part b to use the `ListArrayBased` class?
  13. Given two circular linked lists, one referenced by `p`, the other by `q`, append list `q` to list `p`. This should result in a circular list reference by `p` that contains all of the elements of the original list referenced by `p` followed by the elements of the list referenced by `q`.
  14. Imagine a circular linked list of integers that are sorted into ascending order, as Figure 5-33a illustrates. The external reference `list` references the last node, which contains the largest integer. Write a method that revises the list so that its data elements are sorted into descending order, as Figure 5-33b illustrates. Do not allocate new nodes.

**FIGURE 5-33**

Two circular linked lists

15. Revise the implementations of the ADT list operations *add*, *get*, and *remove* on pages 267–268 under the assumption that the linked list has a dummy head node.
16. Add the operations *save* and *restore* to the reference-based implementation of the ADT list. These operations have a file parameter that indicates the file to save and restore the list items.
17. Consider the sorted doubly linked list shown in Figure 5-27. This list is a circular doubly linked list and has a dummy head node. Write methods for the following operation for a sorted list:

```
*sortedAdd(in item: ListItemType)
 // Inserts item into its proper sorted position in a
 // sorted list.

*sortedRemove(in item: ListItemType)
 // Deletes item from a sorted list.
 // Throws an exception if the item is not found.
```

18. Repeat Exercise 17 for the sorted doubly linked list shown in Figure 5-26. This list is not circular and does not have a dummy head node. Watch out for the special cases at the beginning and end of the list.
19. The class `java.util.Vector` implements a growable array of objects. Here is a subset of the methods available in the class `java.util.Vector`:

#### Constructors

```
Vector();
 // Constructs an empty vector.
```

```
Vector(int initialCapacity, int capacityIncrement)
 // Constructs an empty vector with the specified
 // initial capacity and capacity increment.
```

#### Methods

```
void addElement(Object obj);
 // Adds the specified component to the end of this
 // vector, increasing its size by one.
```

```
int capacity();
 // Returns the current capacity of this vector.
```

```
Object elementAt(int index);
 // Returns the component at the specified index.
```

```
void removeElementAt(int index);
 // Deletes the component at the specified index.
```

```
int size();
 // Returns the number of components in this vector.
```

Complete the following tasks:

- Declare a vector *first* with an initial capacity of 10 and a capacity increment of 5. Write a *for* loop that initializes all ten elements of the vector to the integers 1 through 10. What do the methods *size* and *capacity* return?

- b. Add four more integers to the vector `first`. What do the methods `size` and `capacity` return now?
  - c. Write a loop to print all of the elements stored in the vector `first`.
  - d. Declare a vector `second` using the default constructor. What does `capacity` return in this case? Write a loop to add elements to the vector `second` so that this capacity is exceeded by one. What is the new capacity?
  - e. Delete all of the elements from the vector `second`. Does the capacity change? Why do you think the `Vector` class behaves this way?
10. You can have a linked list of linked lists, as Figure 5-30 indicates. Assume the Java definitions on page 288–289. Suppose that `curr` references a desired stock item (`node`) in the inventory list. Write some Java statements that add yourself as a customer to the end of the wait list associated with the node referenced by `curr`.

## Programming Problems

1. Chapter 4 introduced the ADT sorted list, which maintains its data in sorted order. For example, a sorted list of names would be maintained in alphabetical order, and a sorted list of numbers would be maintained in either increasing or decreasing order. The operations for a sorted list are summarized on page 210.

Some operations—`sortedIsEmpty`, `sortedSize`, and `sortedGet`, for example—are just like those for the ADT list. Insertion and deletion operations, however, are by value, not by position as they are for a list. For example, when you insert an item into a sorted list, you do not specify where in the list the item belongs. Instead, the insertion operation determines the correct position of the item by comparing its value with those of the existing items on the list. A new operation, `locateIndex`, determines from the value of an item its numerical position within the sorted list.

Note that the specifications given in Chapter 4 do not say anything about duplicate entries in the sorted list. Depending on your application, you might allow duplicates, or you might want to prevent duplicates from entering the list. For example, a sorted list of Social Security numbers probably should disallow duplicate entries. In this example, an attempt to insert a Social Security number that already exists in the sorted list would fail.

Write a nonrecursive, reference-based implementation of the ADT sorted list of objects as a Java class `SortedListRefBased` such that

- a. Duplicates are allowed
  - b. Duplicates are not allowed, and operations must prevent duplicates from entering the list
2. Repeat Programming Problem 1, but write a recursive, reference-based implementation instead. Recall from this chapter that the recursive methods must be in the private section of the class.

3. Write an implementation of the ADT list interface *ListInterface* that uses the JFC *vector* class to represent the list items.
4. Write a reference-based implementation of the ADT two-ended list, which has insertion and deletion operations at both ends of the list,
  - a. Without a tail reference
  - b. With a tail reference
5. Implement the node structure, including the constructors, for a circular doubly linked list with a dummy head node, assuming it will be package private. Write an implementation of the *ListInterface* using a circular doubly linked list. Note that the section "Doubly Linked Lists" has a discussion on how to insert and delete nodes from such a list.
6. Implement the ADT character string as the class *LinkedString* by using a linked list of characters. Include the following *LinkedString* constructors and methods:

**LinkedString(char[] value)**

Allocates a new character linked list so that it represents the sequence of characters currently contained in the character array argument.

**LinkedString(String original)**

Initializes a new character linked list so that it represents the same sequence of characters as the argument.

**char charAt(int index)**

Returns the char value at the specified index. The first character in the linked character string is in position zero.

**LinkedString concat(LinkedString str)**

Concatenates the specified linked character string to the end of this linked character string.

**boolean isEmpty()**

Returns true if, and only if, length() is 0.

**int length()**

Returns the length of this linked character string.

**LinkedString substring(int beginIndex, int endIndex)**

Returns a new linked character string that is a substring of this linked character string.

Implement *LinkedString* so that it is consistent with the *String* class. For example, character positions start at zero. Also, keep track of the number of characters in the string; the length should be determined without traversing the linked list and counting.

A *polynomial* of a single variable  $x$  with integer coefficients is an expression of the form

$$p(x) = c_0 + c_1x + c_2x^2 + \dots + c_nx^n,$$

where  $c_i, i = 0, 1, \dots, n$ , are integers.

Consider a sparse implementation of the ADT polynomials up to the  $n^{th}$  degree that stores only the terms with nonzero coefficients. For example, the polynomial

$$p = -3x^7 + 4x^5 + 7x^3 - x^2 + 9$$

can be represented using the linked list shown in Figure 5-35. Complete the class *Polynomial* based on this sparse implementation. Assume *Polynomial* has the following methods:

**Polynomial()**

Constructs a new polynomial of degree zero.

**int getCoefficient(int power)**

Returns an integer representing the coefficient of the  $x^{power}$  term.

**void setCoefficient(int coef, int power)**

Sets the coefficient of the  $x^{power}$  term to *coef*.

**String toString()**

Returns the *String* representation of the polynomial. For example,  $3x^2 + 2x + 1$  would be returned as  $3 * x^2 + 2 * x + 1$  or, more simply,  $3x^2 + 2x + 1$ .

Any term whose coefficient is zero should not appear in the string unless the polynomial has only a single constant term of zero.

**double evaluate(double x)**

Evaluates the polynomial for the value *x* and returns the result  $p(x)$ .

**Polynomial double add(Polynomial other)**

Add to this polynomial the polynomial *other* and return the resulting polynomial.

Round robin (RR) is a simple scheduling algorithm for processes using the CPU. Each process is given a slice of time on the CPU in equal portions and in circular order. The algorithm assumes that we know the burst time for each process—this is the amount of time that the process needs the CPU before the next I/O request.

- a. Design an ADT to represent a process. Each process has an id and keeps track of the burst time.

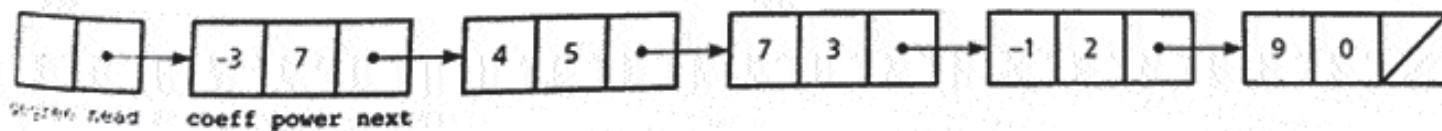


FIGURE 5-34

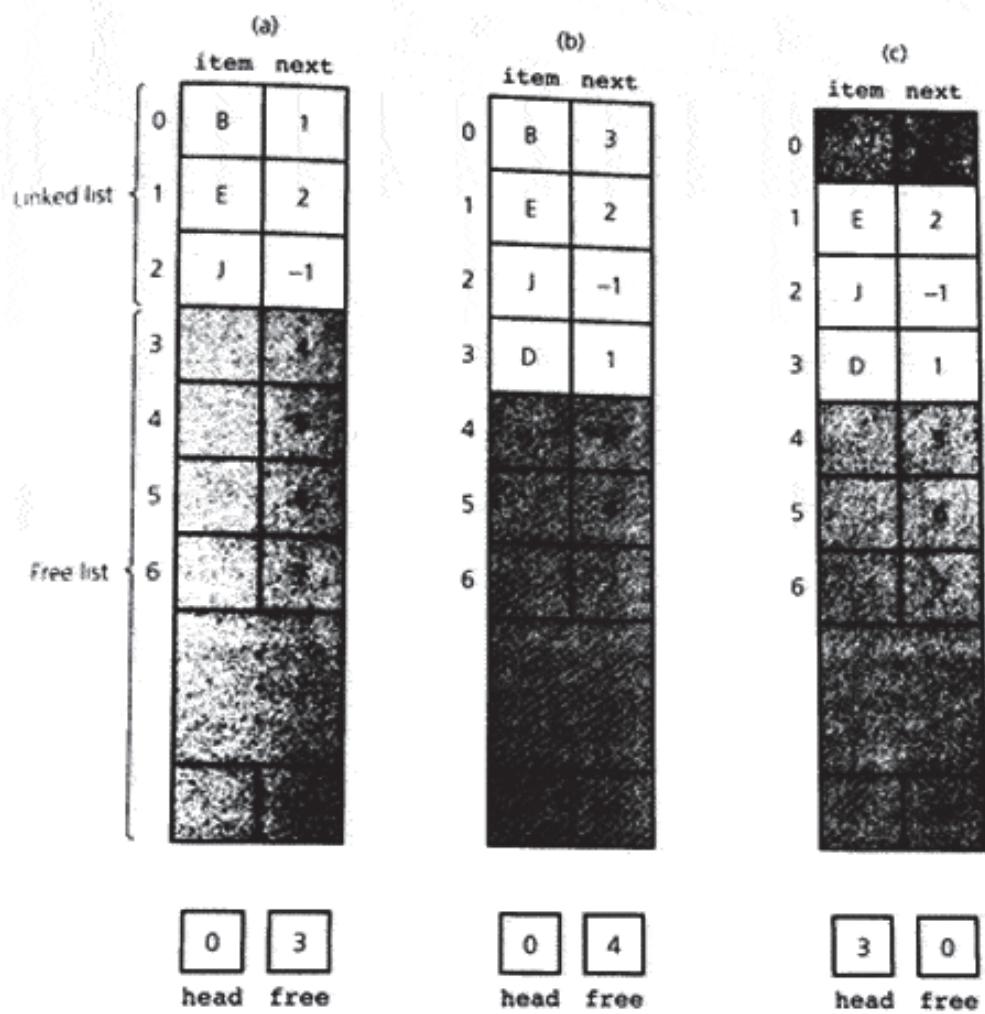
An ADT: polynomial

- b. Design an ADT that keeps track of a group of processes using the CPU with round robin scheduling. Use a circular linked list as the data structure that keeps track of the processes wanting to use the CPU. Run the schedule by traversing through the list, simulating each process getting a slice of the CPU, assuming that the process gets a 100 millisecond time slice, and that its burst time is then reduced by 100 milliseconds. Each time the process has the CPU, print to the console the process id and burst time remaining after the process uses the CPU. If a process completes execution (on its last time slice it has 100 milliseconds or less), remove it from the list. When the list of processes is empty, the schedule is complete.
  - c. Write a test program that demonstrates your ADT completely. Your program should try many different scenarios—for example, the process with the shortest burst time first, all the processes with the same burst time, and processes with burst times that are not multiples of 100 milliseconds.
9. Occasionally, a linked structure that does not use references is useful. One such structure uses an array whose items are “linked” by array indexes. Figure 5-35a illustrates an array of nodes that represents the linked list in Figure 5-31. Each node has two data fields, *item* and *next*. The *next* data field is an integer index to the array element that contains the next node in the linked list. Note that the *next* data field of the last node contains -1. The integer variable *head* contains the index of the first node in the list.

The array elements that currently are not a part of the linked list make up a free list of available nodes. These nodes form another linked list, with the integer variable *free* containing the index of the first free node. To insert an item into the original linked list, you take a free node from the beginning of the free list and insert it into the linked list (Figure 5-35b). When you delete an item from the linked list, you insert the node into the beginning of the free list (Figure 5-35c). In this way, you can avoid shifting data items.

Implement the ADT list by using this array-based linked list.

10. Write the program for the DVD inventory problem that this chapter describes.
11. Modify and expand the inventory program that you wrote for the previous programming problem. Here are a few suggestions:
  - a. Add the ability to manipulate more than one inventory with the single program.
  - b. Add the ability to keep various statistics about each of the inventory items (such as the average number sold per week for the last 10 weeks).
  - c. Add the ability to modify the *have* value for an inventory item (for example, when a DVD is damaged or returned by a customer). Consider the implications for maintaining the relationship between a *have* value and the size of the corresponding wait list.
  - d. Make the wait lists more sophisticated. For example, keep names and addresses; mail letters automatically when a DVD comes in.
  - e. Make the ordering mechanism more sophisticated. For instance, do not order DVDs that have already been ordered but have not yet been delivered.

**FIGURE 5-35**

(a) An array-based implementation of the linked list in Figure 5-31; (b) after inserting D in sorted order; (c) after deleting B.