

INTERNATIONAL
EDITION



Java Software Structures

Designing and Using Data Structures

FOURTH EDITION

John Lewis • Joseph Chase

ALWAYS LEARNING

PEARSON

Java™ Software Structures

DESIGNING AND USING
DATA STRUCTURES

4TH EDITION

This page is intentionally left blank.



Java™ Software Structures

DESIGNING AND USING
DATA STRUCTURES

4TH EDITION

JOHN LEWIS

Virginia Tech

AND

JOSEPH CHASE

Radford University

International Edition contributions by

PIYALI SENGUPTA

PEARSON

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

Acquisitions Editor, US Edition: Matt Goldstein
Editorial Assistant: Jenah Blitz-Stoehr
Senior Managing Editor: Scott Disanno
Senior Production Supervisor: Marilyn Lloyd
Marketing Manager: Yes Alayan
Marketing Coordinator: Kathryn Ferranti
Publisher, International Edition: Angshuman Chakraborty
Publishing Administrator and Business Analyst, International Edition: Shokhi Shah Khandelwal
Associate Print & Media Editor, International Edition: Anuprova Dey Chowdhuri
Acquisitions Editor, International Edition: Sandhya Ghoshal

Publishing Administrator, International Edition: Hema Mehta
Project Editor, International Edition: Karthik Subramanian
Senior Manufacturing Controller, Production, International Edition: Trudy Kimber
Marketing Coordinator: Kathryn Ferranti
Manufacturing Buyer: Lisa McDowell
Cover Design: Jodi Notowitz
Project Management and Illustrations: Cenveo® Publisher Services
Project Manager, Cenveo® Publisher Services, Inc.: Rose Kernan
Text Design, Cenveo® Publisher Services, Inc.: Jerilyn Bockorick, Alisha Webber
Cover Image: Viachaslau Kraskouski/Shutterstock

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the world

Visit us on the World Wide Web at:
www.pearsoninternationaleditions.com

© Pearson Education Limited 2014

The rights of John Lewis and Joseph Chase to be identified as authors of this work have been asserted by them in accordance with the Copyright, Designs and Patents Act 1988.

Authorized adaptation from the United States edition, entitled Java Software Structures, 4th edition, ISBN 978-0-13-325012-1, by John Lewis and Joseph Chase, published by Pearson Education © 2014.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without either the prior written permission of the publisher or a license permitting restricted copying in the United Kingdom issued by the Copyright Licensing Agency Ltd, Saffron House, 6–10 Kirby Street, London EC1N 8TS.

All trademarks used herein are the property of their respective owners. The use of any trademark in this text does not vest in the author or publisher any trademark ownership rights in such trademarks, nor does the use of such trademarks imply any affiliation with or endorsement of this book by such owners.

Microsoft and/or its respective suppliers make no representations about the suitability of the information contained in the documents and related graphics published as part of the services for any purpose. All such documents and related graphics are provided “as is” without warranty of any kind. Microsoft and/or its respective suppliers hereby disclaim all warranties and conditions with regard to this information, including all warranties and conditions of merchantability, whether express, implied or statutory, fitness for a particular purpose, title and non-infringement. In no event shall Microsoft and/or its respective suppliers be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of information available from the services.

The documents and related graphics contained herein could include technical inaccuracies or typographical errors. Changes are periodically added to the information herein. Microsoft and/or its respective suppliers may make improvements and/or changes in the product(s) and/or the program(s) described herein at any time. Partial screen shots may be viewed in full within the software version specified.

Microsoft® and Windows® are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.

ISBN 10: 0-273-79332-2

ISBN 13: 978-0-273-79332-8 (Print)

ISBN 13: 978-0-273-79368-7 (PDF)

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library

10 9 8 7 6 5 4 3 2 1

14 13 12 11 10

Typeset in Sabon LT Std Roman by Cenveo® Publisher Services

Printed and bound by Courier Westford in The United States of America
The publisher’s policy is to use paper manufactured from sustainable forests.

*To my wife Sharon and my kids:
Justin, Kayla, Nathan, and Samantha
-J. L.*

*To my loving wife Melissa for her support and encouragement
and to our families, friends, colleagues, and students who have provided
so much support and inspiration through the years.
-J. C.*

This page is intentionally left blank.

Preface

This book is designed to serve as a text for a course on data structures and algorithms. This course is typically referred to as the CS2 course because it is often taken as the second course in a computing curriculum.

Pedagogically, this book follows the style and approach of the leading CS1 book **Java Software Solutions: Foundations of Program Design**, by John Lewis and William Loftus. Our book uses many of the highly regarded features of that book, such as the Key Concept boxes and complete code examples. Together, these two books support a solid and consistent approach to either a two-course or three-course introductory sequence for computing students. That said, this book does not assume that students have used **Java Software Solutions** in a previous course.

Material that might be presented in either course (such as recursion or sorting) is presented in this book as well. We also include strong reference material providing an overview of object-oriented concepts and how they are realized in Java.

We understand the crucial role that the data structures and algorithms course plays in a curriculum and we think this book serves the needs of that course well.

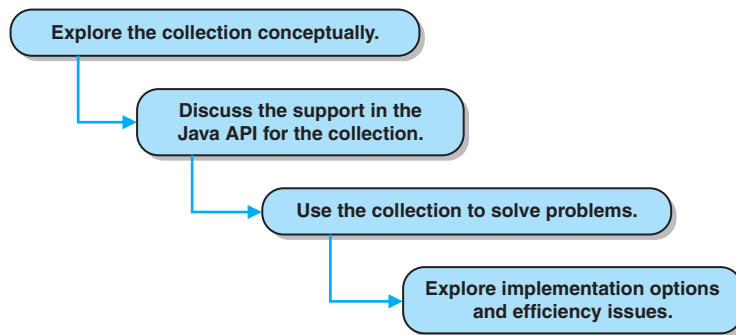
New in the Fourth Edition

We have made some key modifications in this fourth edition to enhance its pedagogy. They can be summarized as follows:

- Revised the collection chapters to provide a more complete explanation of how the Java API supports the collection.
- Added a summary of terms and definitions at the end of each chapter.
- Separated the coverage of Iterators into its own chapter and expanded the discussion.
- Added a new Code Annotation feature, used to explore key statements with graphic annotations.
- Added a new Common Error callout feature.
- Added new Design Focus callouts.

- Added a new appendices covering graphical drawing, graphical user interface development, and regular expressions.
- Reviewed and updated the text throughout to improve discussions and address issues.

In particular, we've reworked the discussion of individual collections to match the following flow:



This approach clarifies the distinction between the way the Java API supports a particular collection and the way it might be implemented from scratch. It makes it easier for instructors to point out limitations of the API implementations in a compare-and-contrast fashion. This approach also allows an instructor, on a case-by-case basis, to simply introduce a collection without exploring implementation details if desired.

The other modifications for this edition flesh out the presentation to a higher degree than previous editions did. The addition of a term list (with succinct definitions) at the end of each chapter provides a summary of core issues in ways that the other features don't. New Code Annotation and Common Error features highlight specific issues that might otherwise get lost in the body of the text, but without interrupting the flow of the topic.

We think these modifications build upon the strong pedagogy established by previous editions and give instructors more opportunity and flexibility to cover topics as they choose.

Our Approach

Books of this type vary greatly in their overall approach. Our approach is founded on a few important principles that we fervently embraced. First, we present the various collections explored in the book in a consistent manner. Second, we

emphasize the importance of sound software design techniques. Third, we organized the book to support and reinforce the big picture: the study of data structures and algorithms.

Throughout the book, we keep sound software engineering practices a high priority. Our design of collection implementations and the programs that use them follow consistent and appropriate standards.

Of primary importance is the separation of a collection's interface from its underlying implementation. The services that a collection provides are always formally defined in a Java interface. The interface name is used as the type designation of the collection whenever appropriate to reinforce the collection as an abstraction.

Chapter Breakdown

Chapter 1 (Introduction) discusses various aspects of software quality and provides an overview of software development issues. It is designed to establish the appropriate mindset before embarking on the details of data structure and algorithm design.

Chapter 2 (Analysis of Algorithms) lays the foundation for determining the efficiency of an algorithm and explains the important criteria that allow a developer to compare one algorithm to another in proper ways. Our emphasis in this chapter is understanding the important concepts more than getting mired in heavy math or formality.

Chapter 3 (Introduction to Collections—Stacks) establishes the concept of a collection, stressing the need to separate the interface from the implementation. It also conceptually introduces a stack, then explores an array-based implementation of a stack.

Chapter 4 (Linked Structures—Stacks) discusses the use of references to create linked data structures. It explores the basic issues regarding the management of linked lists, and then defines an alternative implementation of a stack (introduced in Chapter 3) using an underlying linked data structure.

Chapter 5 (Queues) explores the concept and implementation of a first-in, first-out queue. Radix sort is discussed as an example of using queues effectively. The implementation options covered include an underlying linked list as well as both fixed and circular arrays.

Chapter 6 (Lists) covers three types of lists: ordered, unordered, and indexed. These three types of lists are compared and contrasted, with discussion of the operations that they share and those that are unique to each type. Inheritance is used appropriately in the design of the various types of lists, which are implemented using both array-based and linked representations.

Chapter 7 (Iterators) is a new chapter that isolates the concepts and implementation of iterators, which are so important to collections. The expanded discussion drives home the need to separate the iterator functionality from the details of any particular collection.

Chapter 8 (Recursion) is a general introduction to the concept of recursion and how recursive solutions can be elegant. It explores the implementation details of recursion and discusses the basic idea of analyzing recursive algorithms.

Chapter 9 (Searching and Sorting) discusses the linear and binary search algorithms, as well as the algorithms for several sorts: selection sort, insertion sort, bubble sort, quick sort, and merge sort. Programming issues related to searching and sorting, such as using the Comparable interface as the basis of comparing objects, are stressed in this chapter. Searching and sorting that are based in particular data structures (such as heap sort) are covered in the appropriate chapter later in the book.

Chapter 10 (Trees) provides an overview of trees, establishing key terminology and concepts. It discusses various implementation approaches and uses a binary tree to represent and evaluate an arithmetic expression.

Chapter 11 (Binary Search Trees) builds off of the basic concepts established in Chapter 10 to define a classic binary search tree. A linked implementation of a binary search tree is examined, followed by a discussion of how the balance in the tree nodes is key to its performance. That leads to exploring AVL and red/black implementations of binary search trees.

Chapter 12 (Heaps and Priority Queues) explores the concept, use, and implementations of heaps and specifically their relationship to priority queues. A heap sort is used as an example of its usefulness as well. Both linked and array-based implementations are explored.

Chapter 13 (Sets and Maps) explores these two types of collections and their importance to the Java Collections API.

Chapter 14 (Multi-way Search Trees) is a natural extension of the discussion of the previous chapters. The concepts of 2-3 trees, 2-4 trees, and general B-trees are examined and implementation options are discussed.

Chapter 15 (Graphs) explores the concept of undirected and directed graphs and establishes important terminology. It examines several common graph algorithms and discusses implementation options, including adjacency matrices.

Appendix A (UML) provides an introduction to the Unified Modeling Language as a reference. UML is the de facto standard notation for representing object-oriented systems.

Appendix B (Object-Oriented Concepts) is a reference for anyone needing a review of fundamental object-oriented concepts and how they are accomplished

in Java. Included are the concepts of abstraction, classes, encapsulation, inheritance, and polymorphism, as well as many related Java language constructs such as interfaces.

Appendix C (Graphics) covers the basics of drawing shapes using the Java API.

Appendix D (Graphical User Interfaces) provides a detailed overview of the elements needed to develop a Swing-based GUI. It includes many examples using a variety of interface components.

Appendix E (Hashing) covers the concept of hashing and related issues, such as hash functions and collisions. Various Java Collections API options for hashing are discussed.

Appendix F (Regular Expressions) provides an introduction to the use of regular expressions, which come into play in various Java API elements, such as the Scanner class.

Supplements

The following student resources are available for this book:

- **Source code** for all programs presented in the book
- **VideoNotes** that explore select topics from the book

Resources can be accessed at www.pearsoninternationaleditions.com/lewis

The following instructor resources can be found at Pearson Education's Instructor Resource Center:

- **Solutions** for select exercises and programming projects in the book
- **Powerpoint slides** for the presentation of the book content
- **Test bank**

To obtain access, please visit www.pearsoninternationaleditions.com/lewis or contact your local Pearson Education sales representative.

Pearson would also like to thank Mohit P. Tahiliani of NITK Surathkal for reviewing the content of the International Edition.

This page is intentionally left blank.

Contents

Preface		7
Credits		25
Chapter 1	Introduction	27
	1.1 Software Quality	28
	Correctness	29
	Reliability	29
	Robustness	30
	Usability	30
	Maintainability	31
	Reusability	31
	Portability	32
	Efficiency	32
	Quality Issues	32
	1.2 Data Structures	33
	A Physical Example	33
	Containers as Objects	36
Chapter 2	Analysis of Algorithms	41
	2.1 Algorithm Efficiency	42
	2.2 Growth Functions and Big-Oh Notation	43
	2.3 Comparing Growth Functions	45
	2.4 Determining Time Complexity	48
	Analyzing Loop Execution	48
	Nested Loops	48
	Method Calls	49

Chapter 3	Introduction to Collections – Stacks	55
3.1	Collections	56
	Abstract Data Types	57
	The Java Collections API	59
3.2	A Stack Collection	59
3.3	Crucial OO Concepts	61
	Inheritance and Polymorphism	62
	Generics	63
3.4	Using Stacks: Evaluating Postfix Expressions	64
	Javadoc	71
3.5	Exceptions	72
3.6	A Stack ADT	74
3.7	Implementing a Stack: With Arrays	77
	Managing Capacity	78
3.8	The ArrayStack Class	79
	The Constructors	80
	The push Operation	82
	The pop Operation	83
	The peek Operation	85
	Other Operations	85
	The EmptyCollectionException Class	85
	Other Implementations	86
Chapter 4	Linked Structures – Stacks	93
4.1	References as Links	94
4.2	Managing Linked Lists	96
	Accessing Elements	96
	Inserting Nodes	97
	Deleting Nodes	98
4.3	Elements without Links	99
	Doubly Linked Lists	99
4.4	Stacks in the Java API	100

4.5	Using Stacks: Traversing a Maze	101
4.6	Implementing a Stack: With Links	110
	The <code>LinkedList</code> Class	110
	The <code>push</code> Operation	114
	The <code>pop</code> Operation	116
	Other Operations	117
Chapter 5	Queues	123
5.1	A Conceptual Queue	124
5.2	Queues in the Java API	125
5.3	Using Queues: Code Keys	126
5.4	Using Queues: Ticket Counter Simulation	130
5.5	A Queue ADT	135
5.6	A Linked Implementation of a Queue	137
	The <code>enqueue</code> Operation	139
	The <code>dequeue</code> Operation	141
	Other Operations	142
5.7	Implementing Queues: With Arrays	143
	The <code>enqueue</code> Operation	147
	The <code>dequeue</code> Operation	149
	Other Operations	150
5.8	Double-Ended Queues (Deque)	150
Chapter 6	Lists	155
6.1	A List Collection	156
6.2	Lists in the Java Collections API	158
6.3	Using Unordered Lists: Program of Study	159
6.4	Using Indexed Lists: Josephus	170
6.5	A List ADT	172
	Adding Elements to a List	173

	6.6	Implementing Lists with Arrays	178
		The remove Operation	180
		The contains Operation	182
		The add Operation for an Ordered List	183
		Operations Particular to Unordered Lists	185
		The addAfter Operation for an Unordered List	185
	6.7	Implementing Lists with Links	186
		The remove Operation	187
Chapter 7	Iterators		195
	7.1	What's an Iterator?	196
		Other Iterator Issues	198
	7.2	Using Iterators: Program of Study Revisited	198
		Printing Certain Courses	202
		Removing Courses	203
	7.3	Implementing Iterators: With Arrays	205
	7.4	Implementing Iterators: With Links	207
Chapter 8	Recursion		213
	8.1	Recursive Thinking	214
		Infinite Recursion	214
		Recursion in Math	215
	8.2	Recursive Programming	216
		Recursion versus Iteration	219
		Direct versus Indirect Recursion	219
	8.3	Using Recursion	220
		Traversing a Maze	220
		The Towers of Hanoi	228
	8.4	Analyzing Recursive Algorithms	233
Chapter 9	Searching and Sorting		241
	9.1	Searching	242
		Static Methods	243
		Generic Methods	243

	Linear Search	244
	Binary Search	246
	Comparing Search Algorithms	248
9.2	Sorting	249
	Selection Sort	252
	Insertion Sort	254
	Bubble Sort	256
	Quick Sort	258
	Merge Sort	262
9.3	Radix Sort	265
Chapter 10	Trees	275
10.1	Trees	276
	Tree Classifications	277
10.2	Strategies for Implementing Trees	279
	Computational Strategy for Array Implementation of Trees	279
	Simulated Link Strategy for Array Implementation of Trees	279
	Analysis of Trees	281
10.3	Tree Traversals	282
	Preorder Traversal	282
	Inorder Traversal	283
	Postorder Traversal	283
	Level-Order Traversal	284
10.4	A Binary Tree ADT	285
10.5	Using Binary Trees: Expression Trees	289
10.6	A Back Pain Analyzer	301
10.7	Implementing Binary Trees with Links	305
	The <code>find</code> Method	310
	The <code>iteratorInOrder</code> Method	312
Chapter 11	Binary Search Trees	319
11.1	A Binary Search Tree	320

11.2	Implementing Binary Search Trees: With Links	322
	The addElement Operation	323
	The removeElement Operation	326
	The removeAllOccurrences Operation	329
	The removeMin Operation	330
	Implementing Binary Search Trees: With Arrays	332
11.3	Using Binary Search Trees: Implementing Ordered Lists	332
	Analysis of the BinarySearchTreeList Implementation	335
11.4	Balanced Binary Search Trees	336
	Right Rotation	337
	Left Rotation	338
	Rightleft Rotation	339
	Leftright Rotation	339
11.5	Implementing BSTs: AVL Trees	340
	Right Rotation in an AVL Tree	341
	Left Rotation in an AVL Tree	341
	Rightleft Rotation in an AVL Tree	341
	Leftright Rotation in an AVL Tree	343
11.6	Implementing BSTs: Red/Black Trees	343
	Insertion into a Red/Black Tree	344
	Element Removal from a Red/Black Tree	347
Chapter 12	Heaps and Priority Queues	357
12.1	A Heap	358
	The addElement Operation	360
	The removeMin Operation	361
	The findMin Operation	362
12.2	Using Heaps: Priority Queues	362
12.3	Implementing Heaps: With Links	366
	The addElement Operation	368
	The removeMin Operation	370
	The findMin Operation	373

12.4	Implementing Heaps: With Arrays	373
	The addElement Operation	375
	The removeMin Operation	376
	The findMin Operation	378
12.5	Using Heaps: Heap Sort	378
Chapter 13	Sets and Maps	385
13.1	Set and Map Collections	386
13.2	Sets and Maps in the Java API	386
13.3	Using Sets: Domain Blocker	389
13.4	Using Maps: Product Sales	392
13.5	Using Maps: User Management	396
13.6	Implementing Sets and Maps Using Trees	401
13.7	Implementing Sets and Maps Using Hashing	401
Chapter 14	Multi-Way Search Trees	409
14.1	Combining Tree Concepts	410
14.2	2-3 Trees	410
	Inserting Elements into a 2-3 Tree	411
	Removing Elements from a 2-3 Tree	413
14.3	2-4 Trees	416
14.4	B-Trees	418
	B*-Trees	419
	B+-Trees	419
	Analysis of B-Trees	420
14.5	Implementation Strategies for B-Trees	420
Chapter 15	Graphs	427
15.1	Undirected Graphs	428
15.2	Directed Graphs	429

15.3	Networks	431
15.4	Common Graph Algorithms	432
	Traversals	432
	Testing for Connectivity	436
	Minimum Spanning Trees	438
	Determining the Shortest Path	441
15.5	Strategies for Implementing Graphs	441
	Adjacency Lists	442
	Adjacency Matrices	442
15.6	Implementing Undirected Graphs with an Adjacency Matrix	443
	The addEdge Method	448
	The addVertex Method	448
	The expandCapacity Method	449
	Other Methods	450
Appendix A	UML	455
	The Unified Modeling Language (UML)	456
	UML Class Diagrams	456
	UML Relationships	458
Appendix B	Object-Oriented Design	463
B.1	Overview of Object-Oriented Design	464
B.2	Using Objects	464
	Abstraction	465
	Creating Objects	466
B.3	Class Libraries and Packages	468
	The import Declaration	468
B.4	State and Behavior	469
B.5	Classes	470
	Instance Data	473

B.6	Encapsulation	474
	Visibility Modifiers	474
	Local Data	476
B.7	Constructors	476
B.8	Method Overloading	477
B.9	References Revisited	478
	The Null Reference	478
	The <code>this</code> Reference	479
	Aliases	481
	Garbage Collection	482
	Passing Objects as Parameters	483
B.10	The <code>static</code> Modifier	483
	Static Variables	484
	Static Methods	484
B.11	Wrapper Classes	485
B.12	Interfaces	486
	The Comparable Interface	487
B.13	Inheritance	488
	Derived Classes	488
	The <code>protected</code> Modifier	490
	The <code>super</code> Reference	491
	Overriding Methods	491
B.14	Class Hierarchies	492
	The Object Class	493
	Abstract Classes	494
	Interface Hierarchies	496
B.15	Polymorphism	496
	References and Class Hierarchies	497
	Polymorphism via Inheritance	498
	Polymorphism via Interfaces	498
B.16	Exceptions	501
	Exception Messages	502
	The <code>try</code> Statement	502
	Exception Propagation	503
	The Exception Class Hierarchy	504

Appendix C	Java Graphics	515
C.1	Pixels and Coordinates	516
C.2	Representing Color	517
C.3	Drawing Shapes	518
C.4	Polygons and Polylines	527
	The Polygon Class	530
Appendix D	Graphical User Interfaces	537
D.1	GUI Elements	538
	Frames and Panels	539
	Buttons and Action Events	543
	Determining Event Sources	545
D.2	More Components	548
	Text Fields	548
	Check Boxes	551
	Radio Buttons	555
	Sliders	559
	Combo Boxes	564
	Timers	569
D.3	Layout Managers	574
	Flow Layout	576
	Border Layout	579
	Grid Layout	583
	Box Layout	586
	Containment Hierarchies	589
D.4	Mouse and Key Events	589
	Mouse Events	589
	Key Events	598
	Extending Adapter Classes	604
D.5	Dialog Boxes	605
	File Choosers	608
	Color Choosers	611

D.6	Some Important Details	612
	Borders	612
	Tool Tips and Mnemonics	616
D.7	GUI Design	623
Appendix E	Hashing	633
E.1	Hashing	634
E.2	Hashing Functions	636
	The Division Method	636
	The Folding Method	637
	The Mid-Square Method	637
	The Radix Transformation Method	638
	The Digit Analysis Method	638
	The Length-Dependent Method	638
	Hashing Functions in the Java Language	639
E.3	Resolving Collisions	639
	Chaining	639
	Open Addressing	642
E.4	Deleting Elements from a Hash Table	646
	Deleting from a Chained Implementation	646
	Deleting from an Open Addressing Implementation	647
E.5	Hash Tables in the Java Collections API	648
	The Hashtable Class	648
	The HashSet Class	650
	The HashMap Class	650
	The IdentityHashMap Class	651
	The WeakHashMap Class	652
	LinkedHashSet and LinkedHashMap	653
Appendix F	Regular Expressions	661
Index		665



LOCATION OF VIDEONOTES IN THE TEXT

Chapter 3	An overview of the <code>ArrayStack</code> implementation, page 80
Chapter 4	Using a stack to solve a maze, page 103
Chapter 5	An array-based queue implementation, page 143
Chapter 6	List categories, page 156
Chapter 8	Analyzing recursive algorithms, page 234
Chapter 9	Demonstration of a binary search, page 247
Chapter 10	Demonstration of the four basic tree traversals, page 285
Chapter 11	Demonstration of the four basic tree rotations, page 340
Chapter 12	Demonstration of a heap sort on an array, page 379
Chapter 14	Inserting elements into, and removing elements from, a 2-3 tree, page 414
Chapter 15	Illustration of depth-first and breadth-first traversals of a graph, page 433

Credits

Cover: Viachaslau Kraskouski/Shutterstock

Chapter 1: Quote: “If I have seen further, it is by standing on the shoulders of giants” Newton, Issac. Letter to Robert Hooke. 1676.

Chapter 2: Paraphrase of another way of looking at the effect of algorithm complexity Aho, A. A., J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Reading, Mass.: 2nd Edition Addison-Wesley, 1974.

Chapter 11: Adel’son-Vel’skii and Landis developed a method called AVL trees that is a variation on this theme. For each node in the tree, we will keep track of the height of the left and right subtrees. Georgii M. Adelson-Velskii and Evgenii M. Landis, “An Algorithm for the Organization of Information.” *Doklady Akademii Nauk SSSR*, 146:263–266, 1962 (Russian). English translation by Myron J. Ricci in *Soviet Math. Doklady*, 3:1259-1263, 1962.

Quote: “Another alternative to the implementation of binary search trees is the concept of a red/black tree, developed by Bayer and extended by Guibas and Sedgewick.” Guibas, L. and R. Sedgewick. “A Diochromatic Framework for Balanced Trees.” *Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science* (1978): 8–21.

Chapter 15: The second possibility for determining the shortest path is to look for the cheapest path in a weighted graph. Dijkstra developed an algorithm for this possibility that is similar to our previous algorithm. Dijkstra, E. W. “A Note on Two Problems in Connection with Graphs.” *Numerische Mathematik* 1 (1959): 269–271.

Reference: The algorithm for developing a minimum spanning tree was developed by Prim (1957) and is quite elegant. Prim, R. C. “Shortest Connection Networks and Some Generalizations.” *Bell System Technical Journal* 36 (1957): 1389–1401.

MICROSOFT AND/OR ITS RESPECTIVE SUPPLIERS MAKE NO REPRESENTATIONS ABOUT THE SUITABILITY OF THE INFORMATION CONTAINED IN THE DOCUMENTS AND RELATED GRAPHICS PUBLISHED AS PART OF THE SERVICES FOR ANY PURPOSE. ALL SUCH DOCUMENTS AND RELATED GRAPHICS ARE PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND. MICROSOFT AND/OR ITS RESPECTIVE SUPPLIERS HEREBY DISCLAIM ALL WARRANTIES AND CONDITIONS WITH REGARD TO THIS INFORMATION, INCLUDING ALL WARRANTIES AND CONDITIONS OF MERCHANTABILITY, WHETHER EXPRESS, IMPLIED OR STATUTORY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL MICROSOFT AND/OR ITS RESPECTIVE SUPPLIERS BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF INFORMATION AVAILABLE FROM THE SERVICES.

THE DOCUMENTS AND RELATED GRAPHICS CONTAINED HEREIN COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN. MICROSOFT AND/OR ITS RESPECTIVE SUPPLIERS MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED HEREIN AT ANY TIME. PARTIAL SCREEN SHOTS MAY BE VIEWED IN FULL WITHIN THE SOFTWARE VERSION SPECIFIED.

MICROSOFT® AND WINDOWS® ARE REGISTERED TRADEMARKS OF THE MICROSOFT CORPORATION IN THE U.S.A. AND OTHER COUNTRIES. THIS BOOK IS NOT SPONSORED OR ENDORSED BY OR AFFILIATED WITH THE MICROSOFT CORPORATION.



Introduction

Our exploration of data structures begins with an overview of the underlying issues surrounding the quality of software systems. It is important for us to understand that it is necessary for systems to work as specified, but simply working is not sufficient. We must also develop quality systems. This chapter discusses a variety of issues related to software quality and data structures, and it establishes some terminology that is crucial to our exploration of data structures and software design.

CHAPTER OBJECTIVES

- Identify various aspects of software quality.
- Motivate the need for data structures based upon quality issues.
- Introduce the basic concept of a data structure.
- Introduce several elementary data structures.

1.1 Software Quality

Imagine a scenario where you are approaching a bridge that has recently been built over a large river. As you approach, you see a sign informing you that the bridge was designed and built by local construction workers and that engineers were not involved in the project. Would you continue across the bridge? Would it make a difference if the sign informed you that the bridge was designed by engineers and built by construction workers?

The word *engineer* in this context refers to an individual who has been educated in the history, theory, method, and practice of the engineering discipline. This definition includes fields of study such as electrical engineering, mechanical engineering, and chemical engineering. *Software engineering* is the study of the techniques and theory that underlie the development of high-quality software.

When the term *software engineering* was first coined in the 1970s, it was an aspiration—a goal set out by leaders in the industry who realized that much of the software being created was of poor quality. They wanted developers to move away from the simplistic idea of writing programs and toward the disciplined idea of engineering software. To engineer software we must first realize that this term is more than just a title and that, in fact, it represents a completely different attitude.

Many arguments have been started over the question of whether software engineering has reached the state of a true engineering discipline. We will leave that argument for software engineering courses. For our purposes, it is sufficient to understand that as software developers we share a common history, we are constrained by common theory, and we must understand current methods and practices in order to work together.

Ultimately, we want to satisfy the *client*, the person or organization who pays for the software to be developed, as well as the final *users* of the system, who may include the client, depending on the situation.

The goals of software engineering are much the same as those for other engineering disciplines:

- Solve the right problem.
- Deliver a solution on time and within budget.
- Deliver a high-quality solution.
- Accomplish these things in an ethical manner (see www.acm.org/about/code-of-ethics).

Sir Isaac Newton is credited with having declared, “If I have seen further, it is by standing on the shoulders of giants.” As modern software developers, we stand upon the shoulders of the giants who founded and developed our field. To truly stand upon their shoulders, we must understand, among other things, the fundamentals of quality

Quality Characteristic	Description
Correctness	The degree to which software adheres to its specific requirements.
Reliability	The frequency and criticality of software failure.
Robustness	The degree to which erroneous situations are handled gracefully.
Usability	The ease with which users can learn and execute tasks within the software.
Maintainability	The ease with which changes can be made to the software.
Reusability	The ease with which software components can be reused in the development of other software systems.
Portability	The ease with which software components can be used in multiple computer environments.
Efficiency	The degree to which the software fulfills its purpose without wasting resources.

FIGURE 1.1 Aspects of software quality

software systems and the organization, storage, and retrieval of data. These are the building blocks upon which modern software development is based.

To maximize the quality of our software, we must first realize that quality means different things to different people. And there are a variety of quality characteristics to consider. Figure 1.1 lists several aspects of high-quality software.

Correctness

The concept of *correctness* goes back to our original goal of developing the appropriate solution. At each step along the way in developing a program, we want to make sure that we are addressing the problem as defined by the requirements specification and that we are providing an accurate solution to that problem. Almost all other aspects of quality are meaningless if the software doesn't solve the right problem.

Reliability

If you have ever attempted to access your bank account electronically and been unable to do so, or if you have ever lost all of your work because of a failure of the software or hardware you were using, you are already familiar with the concept

KEY CONCEPT

Reliable software seldom fails, and when it does, it minimizes the effects of that failure.

of *reliability*. A software *failure* can be defined as any unacceptable behavior that occurs within permissible operating conditions. We can compute measures of reliability, such as the mean time between failures or the mean number of operations or tasks between failures.

In some situations, reliability is an issue of life and death. In the early 1980s, a piece of medical equipment called the Therac-25 was designed to deliver a dose of radiation according to the settings made by a technician on a special keyboard. An error existed in the software that controlled the device, and when the technician made a very specific adjustment to the values on the keyboard, the internal settings of the device were changed drastically and a lethal dose of radiation was issued. The error occurred so infrequently that several people died before the source of the problem was determined.

In other cases, reliability repercussions are financial. On a particular day in November 1998, the entire AT&T network infrastructure in the eastern United States failed, causing a major interruption in communications capabilities. The problem was eventually traced back to a specific software error. That one failure cost the companies affected millions of dollars in lost revenue.

Robustness

Reliability is related to how *robust* a system is. A robust system handles problems gracefully. For example, if a particular input field is designed to handle numeric data, what happens when alphabetic information is entered? The program could be allowed to terminate abnormally because of the resulting error. However, a more robust solution would be to design the system to acknowledge and handle the situation transparently or with an appropriate error message.

Developing a thoroughly robust system may or may not be worth the development cost. In some cases, it may be perfectly acceptable for a program to terminate abnormally if very unusual conditions occur. On the other hand, if adding such protections is not excessively costly, it is simply good development practice. Furthermore, well-defined system requirements should carefully spell out the situations in which robust error handling is required.

Usability

To be effective, a software system must be truly *usable*. If a system is too difficult to use, it doesn't matter if it provides wonderful functionality. Within the discipline of computer science, there is a field of study called Human-Computer Interaction (HCI) that focuses on the analysis and design of user interfaces of software systems. The interaction between user and system must be well designed,

including such things as help options, meaningful messages, consistent layout, appropriate use of color, error prevention, and error recovery.

Maintainability

Software developers must *maintain* their software. That is, they must make changes to software in order to fix errors, to enhance the functionality of the system, or simply to keep up with evolving requirements. A useful software system may need to be maintained for many years after its original development. The software engineers who perform maintenance tasks are often not the same ones as those who originally developed the software. Thus, it is important that a software system be well structured, well written, and well documented in order to maximize its maintainability.

Large software systems are rarely written by a single individual or even a small group of developers. Instead, large teams, often working from widely distributed locations, work together to develop systems. For this reason, communication among developers is critical. Therefore, creating maintainable software is beneficial for the long term as well as for the initial development effort.

KEY CONCEPT

Software systems must be carefully designed, written, and documented to support the work of developers, maintainers, and users.

Reusability

Suppose that you are a contractor involved in the construction of an office building. It is possible that you might design and build each door in the building from scratch. That would require a great deal of engineering and construction effort, not to mention money. Another option is to use pre-engineered, prefabricated doors for the doorways in the building. This approach represents a great savings of time and money because you can rely on a proven design that has been used many times before. You can be confident that it has been thoroughly tested and that you know its capabilities. However, this does not exclude the possibility that a few doors in the building will be custom engineered and custom built to fit a specific need.

When developing a software system, it often makes sense to use pre-existing software components if they fit the needs of the developer and the client. Why reinvent the wheel? Pre-existing components can range in scope from the entire system, to entire subsystems, to individual classes and methods. They may come from part of another system developed earlier or from libraries of components that are created to support the development of future systems. Some pre-existing components are referred to as Commercial Off-The-Shelf (COTS) products. Pre-existing components are often reliable because they have been tested in other systems.

Using pre-existing components can reduce the development effort. However, reuse comes at a price. The developer must take the time to investigate potential components to find the right one. Often the component must be modified or

extended to fit the criteria of the new system. Thus, it is helpful if the component is truly *reusable*. That is, software should be written and documented so that it can be easily incorporated into new systems and easily modified or extended to accommodate new requirements.

Portability

Software that is easily *portable* can be moved from one computing environment to another with little or no effort. Software developed using a particular operating system and underlying central processing unit (CPU) may not run well—or may not run at all—in another environment. One obvious problem is a program that has been compiled into a particular CPU’s machine language. Because each type of CPU has its own machine language, porting it to another machine would require another, translated version. Differences among the various translations may cause the “same” program on two types of machines to behave differently.

Using the Java programming language addresses this issue because Java source code is compiled into *bytecode*, which is a low-level language that is not the machine language for any particular CPU. Bytecode runs on a *Java Virtual Machine* (JVM), which is software that interprets the bytecode and executes it. Therefore, at least theoretically, any system that has a JVM can execute any Java program.

Efficiency

The last software quality characteristic listed in Figure 1.1 is efficiency. Software systems should make *efficient* use of the resources allocated to them. Two key resources are CPU time and memory. User demands on computers and their software have risen steadily ever since computers were first created. Software must always make the best use of its resources in order to meet those demands. The efficiency of individual algorithms is an important part of this issue and is discussed in more detail in the next chapter and throughout the book.

KEY CONCEPT

Software must make efficient use of resources such as CPU time and memory.

Quality Issues

To a certain extent, quality is in the eye of the beholder. That is, some quality characteristics are more important to certain people than to others. We must consider the needs of the various *stakeholders*, the people affected one way or another by the project. For example, the end user certainly wants to maximize reliability, usability, and efficiency but doesn’t necessarily care about the software’s maintainability or reusability. The client wants to make sure the user is satisfied

but is also worried about the overall cost. The developers and maintainers want the internal system quality to be high.

Note also that some quality characteristics are in competition with each other. For example, to make a program more efficient, we may choose to use a complex algorithm that is difficult to understand and therefore hard to maintain. These types of trade-offs require us to carefully prioritize the issues related to a particular project and, within those boundaries, maximize all quality characteristics as much as possible. If we decide that we must use the more complex algorithm for efficiency, we can also document the code especially well to assist with future maintenance tasks.

All of these quality characteristics are important, but for our exploration of data structures in this book, we will focus on reliability, robustness, reusability, and efficiency. In the creation of data structures, we are not creating applications or end-user systems; rather, we are creating reusable components that may be used in a variety of systems. Thus, usability is not an issue, because there will not be a user-interface component of our data structures. By implementing in Java and adhering to Javadoc standards, we also address the issues of portability and maintainability.

KEY CONCEPT

Quality characteristics must first be prioritized and then be maximized to the greatest extent possible.

1.2 Data Structures

Why spend so much time talking about software engineering and software quality in a text that focuses on data structures and their algorithms? Well, as you begin to develop more complex programs, it's important to develop a more mature outlook on the process. As we discussed at the beginning of this chapter, the goal should be to engineer software, not just write code. The data structures we examine in this book lay the foundation for complex software that must be carefully designed. Let's consider an example that will illustrate the need for and the various approaches to data structures.

A Physical Example

Imagine that you must design the process for handling shipping containers being unloaded from cargo ships at a dock. In a perfect scenario, the trains and trucks that will haul these containers to their destinations would be waiting as the ship is unloaded and thus there would not be a need to store the containers at all. However, such timing is unlikely, and it would not necessarily provide the most efficient result for the trucks and the trains, because they would have to sit and wait while each ship was unloaded. Instead, as each shipping container is unloaded

from a ship, it is moved to a storage location where it will be held until it is loaded on either a truck or a train. Our goal should be to make both the unloading of containers and the retrieval of containers for transportation as efficient as possible. This will mean minimizing the amount of searching required in either process and minimizing the number of times a container is moved.

Before we go any further, let's examine the initial assumptions underlying this problem. First, our shipping containers are generic. By this we mean that they are all the same shape, are all the same size, and can hold the same volume and types of materials. Second, each shipping container has a unique identification number. This number is the key that determines the final destination of each container and whether it is to be shipped by truck or train. Third, the dock workers handling the containers do not need to know—nor can they know—what is inside each container.

Given these assumptions as a starting point, how might we design our storage process for these containers? One possibility might be to simply lay out a very large array indexed by the identification number of the container. Keep in mind, however, that the identification number is unique, not just for a single ship but for all ships and all shipping containers. Thus, we would need to lay out an array of storage at each dock large enough to handle all of the shipping containers in the world. This would also mean that at any given point in time, the vast majority of the units in these physical arrays would be empty. This would appear to be a terrible waste of space and very inefficient. We will explore issues surrounding the efficiency of algorithms in the next chapter.

We could try this same solution but allow the array to collapse and expand like an `ArrayList` as containers are removed or added. However, given the physical nature of this array, such a solution may involve having to move containers multiple times as other containers with lower ID numbers are added or removed. Again, such a solution would be very inefficient.

What if we could estimate the maximum number of shipping containers that we would be storing on our dock at any given point in time? This would enable us to create an array of that maximum size and then simply to place each container into the next available position in the array as it is unloaded from the ship. This would be relatively efficient for unloading purposes since the dock workers would simply keep track of which locations in the array were empty and place each new container in an empty location. However, using this approach, what would happen when a truck driver arrived looking for a particular container? Since the array is not ordered by container ID, the driver would have to search for the container in the array. In the worst-case scenario, the container the driver was seeking would be the last one in the array, necessitating a search of the entire array. Of course, the average case would be that each driver would have to search half of the array. But that still seems less efficient than it could be.

Before we try a new design, let's reconsider what we know about each container. Of course, we have the container ID. But from that, we also have the destination of each container. What if, instead of an arbitrary ordering of containers in a one-dimensional array, we create a two-dimensional array where the first dimension is the destination? Thus we can implement our previous solution as the second dimension within each destination. Unloading a container is still quite simple. The dock workers would simply take the container to the array for its destination and place it in the first available empty slot. Then our truck drivers would not have to search the entire dock for their shipping container, but only that portion, the array within the array, that is bound for their destination. This solution, which is beginning to behave like a data structure called a *hash table* (discussed in Chapter 13), is an improvement, yet it still requires searching at least a portion of the storage.

The first part of this solution, organizing the first dimension by destination, seems like a good idea. However, using a simple unordered array for the second dimension still does not accomplish our goal. There is one additional piece of information that we have for each container but have not yet examined—the order in which it was unloaded from the ship. Let's consider, as an example, the components of an oilrig being shipped in multiple containers. Does the order of those containers matter? Yes. The crew constructing the oilrig must receive and install the foundation and base components before they can construct the top of the rig. Now, the order in which the containers are removed from the ship and the order in which they are placed in storage is important. To this point, we have been considering only the problem of unloading, storing, and shipping storage containers. Now we begin to see that this problem exists within a larger context, which includes how the ship was loaded at its point of origin. Three possibilities exist for how the ship was loaded: Containers that are order dependent were loaded in the same order in which they are needed, containers that are order dependent were loaded in the reverse order of the order in which they are needed, or containers that are order dependent were loaded without regard to order but with order included as part of the information associated with the container ID. Let's consider each of these cases separately.

Keep in mind that the process of unloading the ship will reverse the order of the loading process onto the ship. This behavior is very much like that of a *stack*, data structure, which we discuss further in Chapters 3 and 4. If the ship was loaded in the order in which the components are needed, then the unloading process will reverse the order. Thus our storage and retrieval process must reverse the order *again* to get the containers back into the correct order. This would be accomplished by taking the containers to the array for their destination and storing them in the order in which they were unloaded. Then the trucker retrieving these containers would simply start with the last one stored. Finally, with this solution we have a storage and retrieval process that involves no searching at all and no extra handling of containers.

KEY CONCEPT

A stack can be used to reverse the order of a set of data.

What if the containers were loaded onto the ship in the reverse order of the order in which they are needed? In that case, unloading the ship will bring the containers off in the order in which they will be needed, and our storage and retrieval process must preserve that order. This would be accomplished by taking the containers to the array for their destination, storing them in the order in which they were unloaded, and then having the truckers start with the first container stored rather than with the last. Again, this solution does not involve any searching or extra handling of containers. This behavior is that of a *queue* data structure, which we will discuss in detail in Chapter 5.

KEY CONCEPT

A queue preserves the order of its data.

Both of our previous solutions, though very efficient, have been dependent on the order in which the containers were placed aboard ship. What do we do if the containers were not placed aboard in any particular order? In that case, we will need to order the destination array by the priority order of the containers. Rather than trying to exhaust this example, let's just say that there are several collections that might accomplish this purpose, including *ordered lists* (Chapter 6), *priority queues* (Chapter 12), and *maps* (Chapter 13). We also have the additional issue of how best to organize the file that relates container ID to the other information about each container. Solutions to this problem might include *binary search trees* (Chapter 11) and *multi-way search trees* (Chapter 14).

Containers as Objects

Our shipping container example illustrates another issue related to data structures beyond our discussion of how to store the containers. It also illustrates the issue of what the containers store. Early in our discussion, we made the assumption that all shipping containers have the same shape and size and that they can all hold the same volume and type of material. Although the first two assumptions must remain true in order for containers to be stored and shipped interchangeably, the latter assumptions may not be true.

For example, consider the shipment of materials that must remain refrigerated. It may be useful to develop refrigerated shipping containers for this purpose. Similarly, some products require very strict humidity control and might need a container with additional environmental controls. Other containers may be designed for hazardous material and might be lined and/or double-walled for protection. Once we develop multiple types of containers, our containers are no longer generic. We can then think of shipping containers as a hierarchy much like a class hierarchy in object-oriented programming. Thus, assigning material to a shipping container becomes analogous to assigning an object to a reference. With both shipping containers and objects, not all assignments are valid. Appendix B discusses issues related to type compatibility, inheritance, polymorphism, and creating generic data structures.

Summary of Key Concepts

- Reliable software seldom fails, and when it does, it minimizes the effects of that failure.
- Software systems must be carefully designed, written, and documented to support the work of developers, maintainers, and users.
- Software must make efficient use of resources such as CPU time and memory.
- Quality characteristics must first be prioritized and then be maximized to the greatest extent possible.
- A stack can be used to reverse the order of a set of data.
- A queue preserves the order of its data.

Summary of Terms

bytecode A low-level representation of software that is executed on the Java Virtual Machine (JVM).

correctness A software quality characteristic that indicates the degree to which the software adheres to its specific requirements.

efficiency A software quality characteristic that indicates the degree to which the software fulfills its purpose without wasting resources.

maintainability A software quality characteristic that indicates the ease with which changes can be made to the software.

portability A software quality characteristic that indicates the ease with which software components can be used in multiple computer environments.

reliability A software quality characteristic that indicates the frequency and criticality of failures in the software.

reusability A software quality characteristic that indicates the ease with which software components can be reused in the development of other software systems.

robustness A software quality characteristic that indicates the degree to which erroneous situations are handled gracefully.

software failure Any unacceptable behavior that occurs in a software system within permissible operating conditions.

stakeholder A person with a vested interest in a project, who is therefore concerned with specific issues related to it.

usability A software quality characteristic that indicates the ease with which users can learn and execute tasks within the software.

Self-Review Questions

- SR 1.1 Define Software Engineering.
- SR 1.2 What are the basic goals of software engineering?
- SR 1.3 What is the difference between the terms Correctness and Robustness in relation with software designing?
- SR 1.4 What do you understand by usability and reusability of software?
- SR 1.5 How do you ensure that a software system is maintainable?
- SR 1.6 What do you understand by data structures?
- SR 1.7 What do you understand by the *efficiency* of software?

Exercises

- EX 1.1 You have developed a software program that can be used in different computer environments. Which particular quality aspect does this software adhere to? What would happen if it did not have this quality?
- EX 1.2 What are the various quality aspects to be followed while designing and developing a software program?
- EX 1.3 Provide examples and describe the implications of code reuse, and reuse of algorithms and data structures, in terms of development of a software system.

Answers to Self-Review Questions

- SRA 1.1 Software engineering is the study of the techniques and theory that underlie the development of high-quality software.
- SRA 1.2 Software engineering has the following basic goals: solving the right problem, delivering a solution on time and within budget, delivering a high-quality solution, and accomplishing all these goals in an ethical manner.
- SRA 1.3 Correctness of software specifies the degree to which software adheres to its specific requirements. Robustness defines the degree to which the software handles erroneous situations gracefully.

- SRA 1.4 Usability of software refers to the ease with which users can learn and execute tasks within the software. Reusability of software refers to the ease with which software components can be reused in the development of other software systems.
- SRA 1.5 To ensure maintainability, the software system should be well structured, well written, and well documented.
- SRA 1.6 A data structure is a specific way of storing and organizing data in a computer.
- SRA 1.7 Software must make efficient use of resources such as CPU time and memory.

This page is intentionally left blank.



Analysis of Algorithms

2

It is important that we understand the concepts surrounding the efficiency of algorithms before we begin building data structures. A data structure built correctly and with an eye toward efficient use of both the CPU and memory is one that can be reused effectively in many different applications. However, using a data structure that is not built efficiently is similar to using a damaged original as the master from which to make copies.

CHAPTER OBJECTIVES

- Discuss the goals of software development with respect to efficiency.
- Introduce the concept of algorithm analysis.
- Explore the concept of asymptotic complexity.
- Compare various growth functions.

2.1 Algorithm Efficiency

One of the characteristics that determines the quality of software is the efficient use of resources. One of the most important resources is CPU time. The efficiency of an algorithm we use to accomplish a particular task is a major factor that determines how fast a program executes. Although the techniques that we will discuss here may also be used to analyze an algorithm in terms of the amount of memory it uses, we will focus our discussion on the efficient use of processing time.

KEY CONCEPT

Algorithm analysis is a fundamental computer science topic.

The *analysis of algorithms* is a fundamental computer science topic and involves a variety of techniques and concepts. It is a primary theme that we return to throughout this text. This chapter introduces the issues related to algorithm analysis and lays the groundwork for using analysis techniques.

Let's start with an everyday example: washing dishes by hand. If we assume that washing a dish takes 30 seconds and drying a dish takes an additional 30 seconds, then we can see quite easily that it would take n minutes to wash and dry n dishes. This computation could be expressed as follows:

$$\begin{aligned}\text{Time (n dishes)} &= n * (30 \text{ seconds wash time} + 30 \text{ seconds dry time}) \\ &= 60n \text{ seconds}\end{aligned}$$

or, written more formally,

$$\begin{aligned}f(x) &= 30x + 30x \\ f(x) &= 60x\end{aligned}$$

On the other hand, suppose we were careless while washing the dishes and splashed too much water around. Suppose that each time we washed a dish, we had to dry not only that dish but also all of the dishes we had washed before that one. It would still take 30 seconds to wash each dish, but now it would take 30 seconds to dry the last dish (once), $2 * 30$ or 60 seconds to dry the second-to-last dish (twice), $3 * 30$ or 90 seconds to dry the third-to-last dish (three times), and so on. This computation could be expressed as follows:

$$\text{Time (n dishes)} = n * (30 \text{ seconds wash time}) + \sum_{i=1}^n (i * 30)$$

When we use the formula for an arithmetic series, $\sum_{i=1}^n i = n(n + 1)/2$, the function becomes

$$\begin{aligned}\text{Time (n dishes)} &= 30n + 30n(n + 1)/2 \\ &= 15n^2 + 45n \text{ seconds}\end{aligned}$$

If there were 30 dishes to wash, the first approach would take 30 minutes, whereas the second (careless) approach would take 247.5 minutes. The more

dishes we wash, the worse that discrepancy becomes. For example, if there were 300 dishes to wash, the first approach would take 300 minutes or 5 hours, whereas the second approach would take 908,315 minutes or roughly 15,000 hours!

2.2 Growth Functions and Big-Oh Notation

For every algorithm we want to analyze, we need to define the size of the problem. For our dishwashing example, the size of the problem is the number of dishes to be washed and dried. We also must determine the value that represents efficient use of time or space. For time considerations, we often pick an appropriate processing step that we'd like to minimize, such as our goal of minimizing the number of times a dish has to be washed and dried. The overall amount of time spent on the task is directly related to how many times we have to perform that task. The algorithm's efficiency can be defined in terms of the problem size and the processing step.

Consider an algorithm that sorts a list of numbers into increasing order. One natural way to express the size of the problem would be in terms of the number of values to be sorted. The processing step we are trying to optimize could be expressed as the number of comparisons we have to make for the algorithm to put the values in order. The more comparisons we make, the more CPU time is used.

A *growth function* shows the relationship between the size of the problem (n) and the value we hope to optimize. This function represents the *time complexity* or *space complexity* of the algorithm.

The growth function for our second dishwashing algorithm is

$$t(n) = 15n^2 + 45n$$

However, it is not typically necessary to know the exact growth function for an algorithm. Instead, we are mainly interested in the *asymptotic complexity* of an algorithm. That is, we want to focus on the general nature of the function as n increases. This characteristic is based on the *dominant term* of the expression—the term that increases most quickly as n increases. As n gets very large, the value of the dishwashing growth function is dominated by the n^2 term because the n^2 term grows much faster than the n term. The constants, in this case 15 and 45, and the secondary term, in this case $45n$, quickly become irrelevant as n increases. That is to say, the value of n^2 dominates the growth in the value of the expression.

The table in Figure 2.1 shows how the two terms and the value of the expression grow. As you can see from the table, as n gets larger, the $15n^2$ term dominates the value of the expression. It is important to note that the $45n$ term is larger for very small values of n . Saying that a term is the dominant term as n gets large does not mean that it is larger than the other terms for all values of n .

KEY CONCEPT

A growth function shows time or space utilization relative to the problem size.

Number of dishes (n)	$15n^2$	$45n$	$15n^2 + 45n$
1	15	45	60
2	60	90	150
5	375	225	600
10	1,500	450	1,950
100	150,000	4,500	154,500
1,000	15,000,000	45,000	15,045,000
10,000	1,500,000,000	450,000	1,500,450,000
100,000	150,000,000,000	4,500,000	150,004,500,000
1,000,000	15,000,000,000,000	45,000,000	15,000,045,000,000
10,000,000	1,500,000,000,000,000	450,000,000	1,500,000,450,000,000

FIGURE 2.1 Comparison of terms in growth function

The asymptotic complexity is called the *order* of the algorithm. Thus our second dishwashing algorithm is said to have order n^2 time complexity, which is written $O(n^2)$. Our first, more efficient dishwashing example, with growth function $t(n) = 60(n)$, would have order n time complexity, written $O(n)$. Thus the reason for the difference between our $O(n)$ original algorithm and our $O(n^2)$ sloppy algorithm is the fact that each dish will have to be dried multiple times.

This notation is referred to as $O()$ or Big-Oh notation. A growth function that executes in constant time regardless of the size of the problem is said to have $O(1)$. In general, we are concerned only with executable statements in a program or algorithm in determining its growth function and efficiency. Keep in mind, however, that some declarations may include initializations, and some of these may be complex enough to factor into the efficiency of an algorithm.

KEY CONCEPT

The order of an algorithm is found by eliminating constants and all but the dominant term in the algorithm's growth function.

As an example, assignment statements and if statements that are executed only once regardless of the size of the problem are $O(1)$. Therefore, it does not matter how many of those you string together; it is still $O(1)$. Loops and method calls may result in higher-order growth functions, because they may result in a statement or series of statements being executed more than once based on the size of the problem. We will discuss these separately in later sections of this chapter. Figure 2.2 shows several growth functions and their asymptotic complexity.

KEY CONCEPT

The order of an algorithm provides an upper bound to the algorithm's growth function.

More formally, saying that the growth function $t(n) = 15n^2 + 45n$ is $O(n^2)$ means that there exists a constant m and some value of n (n_0), such that $t(n) \leq m * n^2$ for all $n > n_0$. Another way of stating this is to say that the order of an algorithm provides an upper bound to its growth function. It is also important to

Growth Function	Order	Label
$t(n) = 17$	$O(1)$	constant
$t(n) = 3 \log n$	$O(\log n)$	logarithmic
$t(n) = 20n - 4$	$O(n)$	linear
$t(n) = 12n \log n + 100n$	$O(n \log n)$	$n \log n$
$t(n) = 3n^2 + 5n - 2$	$O(n^2)$	quadratic
$t(n) = 8n^3 + 3n^2$	$O(n^3)$	cubic
$t(n) = 2^n + 18n^2 + 3n$	$O(2^n)$	exponential

FIGURE 2.2 Some growth functions and their asymptotic complexity

note that there are other, related notations—such as omega (Ω), which refers to a function that provides a lower bound, and theta (Θ), which refers to a function that provides both an upper bound and a lower bound. We will focus our discussion on order.

Because the order of the function is the key factor, the other terms and constants are often not even mentioned. All algorithms within a given order are considered generally equivalent in terms of efficiency. For example, two algorithms to accomplish the same task may have different growth functions, but if they are both $O(n^2)$, then they are considered roughly equivalent with respect to efficiency.

2.3 Comparing Growth Functions

One might assume that, with the advances in the speed of processors and the availability of large amounts of inexpensive memory, algorithm analysis would no longer be necessary. However, nothing could be further from the truth. Processor speed and memory cannot make up for the differences in efficiency of algorithms. Keep in mind that we have been eliminating constants as irrelevant when discussing the order of an algorithm. Increasing processor speed simply adds a constant to the growth function. When possible, finding a more efficient algorithm is a better solution than finding a faster processor.

Another way of looking at the effect of algorithm complexity was proposed by Aho, Hopcroft, and Ullman (1974). If a system can currently handle a problem of size n in a given time period, what happens to the allowable size of the problem if we increase the speed of the processor tenfold? As shown in Figure 2.3, the linear

Algorithm	Time Complexity	Max Problem Size Before Speedup	Max Problem Size After Speedup
A	n	s_1	$10s_1$
B	n^2	s_2	$3.16s_2$
C	n^3	s_3	$2.15s_3$
D	2^n	s_4	$s_4 + 3.3$

FIGURE 2.3 Increase in problem size with a tenfold increase in processor speed

case is relatively simple. Algorithm A, with a linear time complexity of n , is indeed improved by a factor of 10, meaning that this algorithm can process 10 times the data in the same amount of time, given a tenfold speedup of the processor. However, algorithm B, with a time complexity of n^2 , is improved only by a factor of 3.16. Why do we not get the full tenfold increase in problem size? Because the

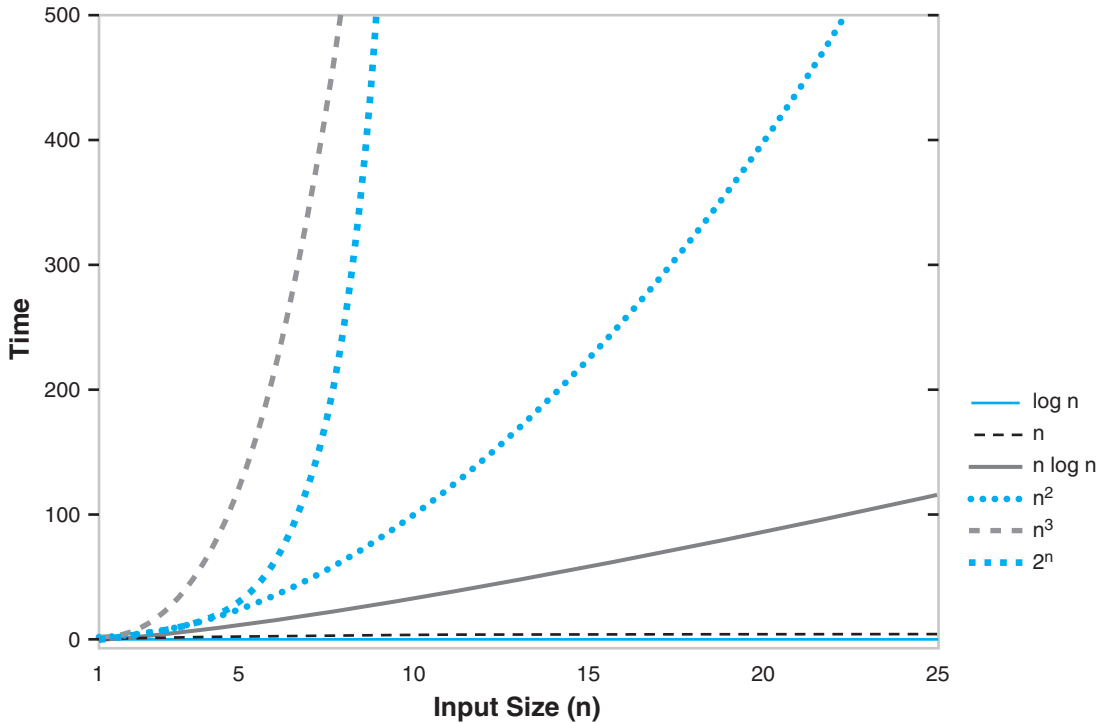


FIGURE 2.4 Comparison of typical growth functions for small values of n

complexity of algorithm B is n^2 , our effective speedup is only the square root of 10, or 3.16.

Similarly, algorithm C, with complexity n^3 , is improved only by a factor of 2.15, or the cube root of 10. For algorithms with *exponential complexity* like algorithm D, in which the size variable is in the exponent of the complexity term, the situation is far worse. In such a case the speedup is $\log_2 n$, or in this case, 3.3. Note this is not a factor of 3, but the original problem size plus 3. In the grand scheme of things, if an algorithm is inefficient, speeding up the processor will not help.

KEY CONCEPT

If the algorithm is inefficient, a faster processor will not help in the long run.

Figure 2.4 illustrates various growth functions graphically for relatively small values of n . Note that when n is small, there is little difference between the algorithms. That is, if you can guarantee a very small problem size (5 or less), it doesn't really matter which algorithm is used. However, notice that in Figure 2.5, as n gets very large, the differences between the growth functions become obvious.

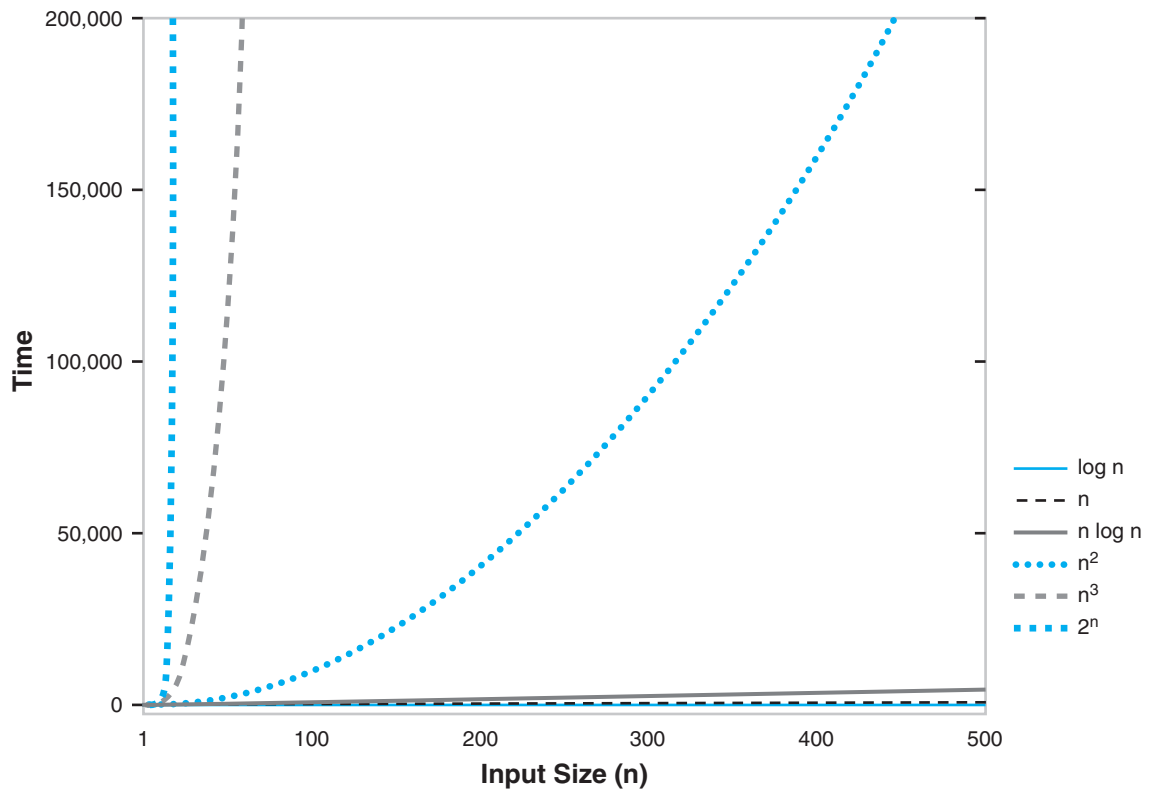


FIGURE 2.5 Comparison of typical growth functions for large values of n

2.4 Determining Time Complexity

Analyzing Loop Execution

KEY CONCEPT

Analyzing algorithm complexity often requires analyzing the execution of loops.

To determine the order of an algorithm, we have to determine how often a particular statement or set of statements is executed. Therefore, we often have to determine how many times the body of a loop is executed. To analyze loop execution, first determine the order of the body of the loop, and then multiply that by the number of times the loop will execute relative to n . Keep in mind that n represents the problem size.

Assuming that the body of a loop is $O(1)$, then a loop such as

```
for (int count = 0; count < n; count++)
{
    /* some sequence of O(1) steps */
}
```

will have $O(n)$ time complexity. This is because the body of the loop has $O(1)$ complexity but is executed n times by the loop structure. In general, if a loop structure steps through n items in a linear fashion and the body of the loop is $O(1)$, then the loop is $O(n)$. Even in a case where the loop is designed to skip some number of elements, as long as the progression of elements to skip is linear, the loop is still $O(n)$. For example, if the preceding loop skipped every other number (for example, `count += 2`), the growth function of the loop would be $n/2$, but since constants don't affect the asymptotic complexity, the order is still $O(n)$.

Let's look at another example. If the progression of the loop is logarithmic, such as the following:

```
count = 1
while (count < n)
{
    count *= 2;
    /* some sequence of O(1) steps */
}
```

KEY CONCEPT

The time complexity of a loop is found by multiplying the complexity of the body of the loop by how many times the loop will execute.

then the loop is said to be $O(\log n)$. Note that when we use a logarithm in an algorithm complexity, we almost always mean log base 2. This can be explicitly written as $O(\log_2 n)$. Each time through the loop, the value of `count` is multiplied by 2, so the number of times the loop is executed is $\log_2 n$.

Nested Loops

A slightly more interesting scenario arises when loops are nested. In this case, we must multiply the complexity of the outer loop by the complexity of the inner loop to find the resulting complexity. For example, the nested loops

```

for (int count = 0; count < n; count++)
{
    for (int count2 = 0; count2 < n; count2++)
    {
        /* some sequence of O(1) steps */
    }
}

```

have complexity $O(n^2)$. The body of the inner loop is $O(1)$, and the inner loop will execute n times. This means the inner loop is $O(n)$. Multiplying this result by the number of times the outer loop will execute (n) results in $O(n^2)$.

What is the complexity of the following nested loop?

```

for (int count = 0; count < n; count++)
{
    for (int count2 = count; count2 < n; count2++)
    {
        /* some sequence of O(1) steps */
    }
}

```

In this case, the inner loop index is initialized to the current value of the index for the outer loop. The outer loop executes n times. The inner loop executes n times the first time, $n-1$ times the second time, and so on. However, remember that we are interested only in the dominant term, not in constants or any lesser terms. If the progression is linear, then regardless of whether some elements are skipped, the order is still $O(n)$. Thus the resulting complexity for this code is $O(n^2)$.

KEY CONCEPT

The analysis of nested loops must take into account both the inner and the outer loops.

Method Calls

Let's suppose that we have the following segment of code:

```

for (int count = 0; count < n; count++)
{
    printsum(count);
}

```

We know from our previous discussion that we find the order of the loop by multiplying the order of the body of the loop by the number of times the loop will execute. In this case, however, the body of the loop is a method call. Therefore, we must determine the order of the method before we can determine the order of the code segment. Let's suppose that the purpose of the method is to print the sum of the integers from 1 to n each time it is called. We might be tempted to create a brute force method such as

```

public void printsum(int count)
{
    int sum = 0;

```

```

    for (int i = 1; i < count; i++)
        sum += i;
    System.out.println(sum);
}

```

What is the time complexity of this `printsum` method? Keep in mind that only executable statements contribute to the time complexity, so in this case, all of the executable statements are $O(1)$ except for the loop. The loop, on the other hand, is $O(n)$, and thus the method itself is $O(n)$. Now, to compute the time complexity of the original loop that called the method, we simply multiply the complexity of the method, which is the body of the loop, by the number of times the loop will execute. Our result, then, is $O(n^2)$ using this implementation of the `printsum` method.

However, we know from our earlier discussion that we do not have to use a loop to calculate the sum of the numbers from 1 to n . In fact, we know that $\sum_1^n i = n(n + 1)/2$. Now let's rewrite our `printsum` method and see what happens to our time complexity:

```

public void printsum(int count)
{
    sum = count*(count+1)/2;
    System.out.println (sum);
}

```

Now the time complexity of the `printsum` method is made up of an assignment statement that is $O(1)$ and a print statement that is also $O(1)$. The result of this change is that the time complexity of the `printsum` method is now $O(1)$, which means that the loop that calls this method now goes from being $O(n^2)$ to being $O(n)$. We know from our earlier discussion and from Figure 2.5 that this is a very significant improvement. Once again, we see that there is a difference between merely delivering correct results and doing so efficiently.

What if the body of a method is made up of multiple method calls and loops? Consider the following code using our `printsum` method above:

```

public void sample(int n)
{
    printsum(n); /* this method call is O(1) */
    for (int count = 0; count < n; count++) /* this loop is O(n) */
        printsum (count);
    for (int count = 0; count < n; count++) /* this loop is O(n^2) */
        for (int count2 = 0; count2 < n; count2++)
            System.out.println (count, count2);
}

```

The initial call to the `printsum` method with the parameter `temp` is $O(1)$ since the method is $O(1)$. The `for` loop containing the call to the `printsum` method with the parameter `count` is $O(n)$ since the method is $O(1)$ and the loop executes n times. The nested loops are $O(n^2)$ since the inner loop will execute n times each time the outer loop executes and the outer loop will also execute n times. The entire method is then $O(n^2)$ since only the dominant term matters.

More formally, the growth function for the method sample is given by

$$f(x) = 1 + n + n^2$$

Then, given that we eliminate constants and all but the dominant term, the time complexity is $O(n^2)$.

There is one additional issue to deal with when analyzing the time complexity of method calls, and that is recursion—the situation when a method calls itself. We will save that discussion for Chapter 8.

Summary of Key Concepts

- Software must make efficient use of resources such as CPU time and memory.
- Algorithm analysis is a fundamental computer science topic.
- A growth function shows time or space utilization relative to the problem size.
- The order of an algorithm is found by eliminating constants and all but the dominant term in the algorithm's growth function.
- The order of an algorithm provides an upper bound to the algorithm's growth function.
- If the algorithm is inefficient, a faster processor will not help in the long run.
- Analyzing algorithm complexity often requires analyzing the execution of loops.
- The time complexity of a loop is found by multiplying the complexity of the body of the loop by how many times the loop will execute.
- The analysis of nested loops must take into account both the inner and the outer loops.

Summary of Terms

analysis of algorithms The computer science topic that focuses on the efficiency of software algorithms.

Big-Oh notation The notation used to represent the order, or asymptotic complexity, of a function.

growth function A function that describes time or space utilization relative to the problem size.

asymptotic complexity A limit on a growth function, defined by the growth function's dominant term; functions similar in asymptotic complexity are grouped into the same general category.

Self-Review Questions

- SR 2.1 What is a growth function? What is the purpose of a growth function?
- SR 2.2 What is the asymptotic complexity of an algorithm?

- SR 2.3 How do you define the order of an algorithm? How do you find the order of an algorithm?
- SR 2.4 What would be the time complexity of a loop with a logarithmic progression?
- SR 2.5 What would be the time complexity of a loop with a loop body that calls a method with a quadratic time complexity?

Exercises

- EX 2.1 Determine the order of each of the following growth functions.
- $30n^3 + 200n^2 + 1000n + 1$
 - $4n^4 - 4$
 - $2^n + n^3 + n^5$
 - $n^3 \log n$
- EX 2.2 Arrange the growth functions of the previous exercise in ascending order of efficiency for $n = 10$ and again for $n = 1,000,000$.
- EX 2.3 Write the code necessary to find the smallest element in an unsorted array of integers. What is the time complexity of this algorithm?
- EX 2.4 Determine the growth function and order of the following code fragment:

```
for (int count=n; count > 0; count--)  
{  
    for (int count2=n; count2 > 0; count2-= 2)  
    {  
        System.out.println(count + ", " + count2);  
    }  
}
```

- EX 2.5 Determine the growth function and order of the following code fragment:

```
for (int count=0; count < n; count=count*10)  
{  
    for (int count2=1; count2 < n; count2++)  
    {  
        System.out.println(count + ", " + count2);  
    }  
}
```

- EX 2.6 You are using an algorithm with a quadratic time complexity of n^2 . How much improvement would you expect in efficiency of the algorithm if there is a processor speed up of 25 times?

Answers to Self-Review Questions

- SRA 2.1 A growth function shows time or space utilization relative to the problem size. The growth function of an algorithm represents the time complexity or space complexity of the algorithm.
- SRA 2.2 Asymptotic complexity gives the changes in the general nature of the function as n or the problem size increases. This characteristic is based on the dominant term of the expression—the term that increases most quickly as n increases.
- SRA 2.3 The order of an algorithm is nothing but the asymptotic complexity of the algorithm. It gives the upper bound to the algorithm's growth function. The order of an algorithm is found by eliminating constants and all but the dominant term in the algorithm's growth function.
- SRA 2.4 If the progression of the loop is logarithmic, then its time complexity would be $O(\log n)$.
- SRA 2.5 Here the loop structure steps through n items in a linear fashion and the body of the loop is $O(n^2)$. Therefore, the time complexity of the entire loop is $O(n^3)$.

Reference

- Aho, A. V., J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Reading, Mass.: Addison-Wesley, 1974.



Introduction to Collections – Stacks

3

This chapter begins our exploration of collections and the underlying data structures used to implement them. It lays the groundwork for the study of collections by carefully defining the issues and goals related to their design. This chapter also introduces a collection called a stack and uses it to exemplify the issues related to the design, implementation, and use of collections.

CHAPTER OBJECTIVES

- Define the concepts and terminology related to collections.
- Explore the basic structure of the Java Collections API.
- Discuss the abstract design of collections.
- Define a stack collection.
- Use a stack collection to solve a problem.
- Examine an array implementation of a stack.

3.1 Collections

KEY CONCEPT

A collection is an object that gathers and organizes other objects.

A *collection* is an object that gathers and organizes other objects. It defines the specific ways in which those objects, which are called *elements* of the collection, can be accessed and managed. The user of a collection, which is generally another class or object in the software system, must interact with the collection only in the prescribed ways.

Over time, several specific types of collections have been defined by software developers and researchers. Each type of collection lends itself to solving particular kinds of problems. A large portion of this text is devoted to exploring these classic collections.

KEY CONCEPT

Elements in a collection are typically organized in terms of the order of their addition to the collection or in terms of some inherent relationship among the elements.

Collections can be separated into two broad categories: linear and nonlinear. As the name implies, a *linear collection* is one in which the elements of the collection are organized in a straight line. A *nonlinear collection* is one in which the elements are organized in something other than a straight line, such as a hierarchy or a network. For that matter, a nonlinear collection may not have any organization at all.

Figure 3.1 shows a linear and a nonlinear collection. It usually doesn't matter whether the elements in a linear collection are depicted horizontally or vertically.

The organization of the elements in a collection, relative to each other, is usually determined by one of two things:

- The order in which they were added to the collection
- Some inherent relationship among the elements themselves

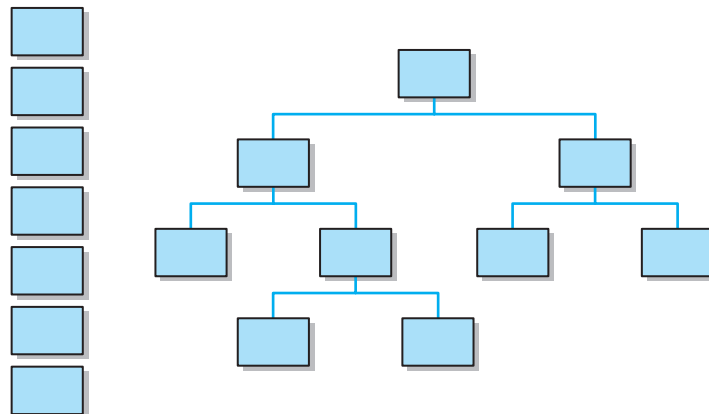


FIGURE 3.1 A linear collection and a nonlinear collection

For example, one linear collection may always add new elements to one end of the line, so the order of the elements is determined by the order in which they are added. Another linear collection may be kept in sorted order based on some characteristic of the elements. For example, a list of people may be kept in alphabetic order based on the characters that make up their name. The specific organization of the elements in a nonlinear collection can be determined in either of these two ways as well.

Abstract Data Types

An *abstraction* hides certain details at certain times. Dealing with an abstraction is easier than dealing with too many details at one time. In fact, we couldn't get through a day without relying on abstractions. For example, we couldn't possibly drive a car if we had to worry about all the details that make the car work: the spark plugs, the pistons, the transmission, and so on. Instead, we can focus on the *interface* to the car: the steering wheel, the pedals, and a few other controls. These controls are an abstraction, hiding the underlying details and enabling us to control an otherwise very complicated machine.

A collection, like any well-designed object, is an abstraction. A collection defines the interface operations through which the user can manage the objects in the collection, such as adding and removing elements. The user interacts with the collection through this interface, as depicted in Figure 3.2. However, the details of how a collection is implemented to fulfill that definition are another issue altogether. A class that implements the collection's interface must fulfill the conceptual definition of the collection, but it can do so in many ways.

KEY CONCEPT

A collection is an abstraction wherein the details of the implementation are hidden.

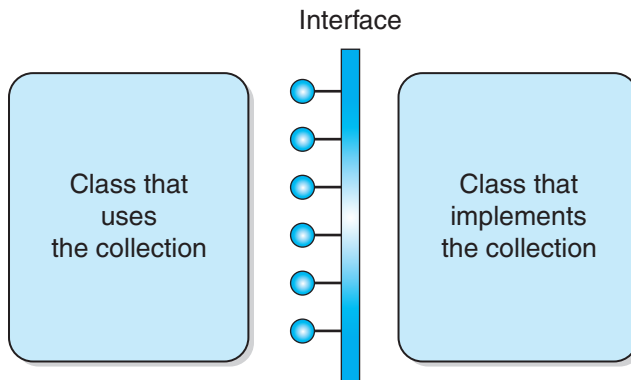


FIGURE 3.2 A well-defined interface masks the implementation of the collection

Abstraction is another important software engineering concept. In large software systems, it is virtually impossible for any one person to grasp all the details of the system at once. Instead, the system is divided into abstract subsystems such that the purpose of and the interactions among those subsystems can be specified. Subsystems may then be assigned to different developers or groups of developers that will develop the subsystem to meet its specification.

An object is the perfect mechanism for creating a collection because, if it is designed correctly, the internal workings of an object are *encapsulated* from the rest of the system. In most cases, the instance variables defined in a class should be declared with private visibility. Therefore, only the methods of that class can access and modify them. The only interaction a user has with an object should be through its public methods, which represent the services that the object provides.

As we progress through our exploration of collections, we will always stress the idea of separating the interface from the implementation. Therefore, for every collection that we examine, we should consider the following issues:

- How does the collection operate, conceptually?
- How do we formally define the interface to the collection?
- What kinds of problems does the collection help us solve?
- What support is already available to us for this type of collection?
- In which various ways might we implement the collection?
- What are the benefits and costs of each implementation?

Before we continue, let's carefully define some other terms related to the exploration of collections. A *data type* is a group of values and the operations defined on those values. The primitive data types defined in Java are the primary examples. For example, the integer data type defines a set of numeric values and the operations (addition, subtraction, etc.) that can be used on them.

An *abstract data type* (ADT) is a data type whose values and operations are not inherently defined within a programming language. It is abstract only in that the details of its implementation must be defined and should be hidden from the user. A collection, therefore, is an abstract data type.

A *data structure* is the collection of programming constructs used to implement a collection. For example, a collection might be implemented using a fixed-size structure such as an array. One interesting artifact of these definitions and our design decision to separate the interface from the implementation (i.e., the collection from the data structure that implements it) is that we may, and often do, end up with a linear data structure, such as an array, being used to implement a nonlinear collection, such as a tree.

KEY CONCEPT

A data structure is the underlying programming construct used to implement a collection.

Historically, the terms *ADT* and *data structure* have been used in various ways. We carefully define them here to avoid any confusion, and will use

them consistently. Throughout this text, we will examine various data structures and how they can be used to implement various collections.

The Java Collections API

The Java programming language is accompanied by a very large library of classes that can be used to support the development of software. Parts of the library are organized into *application programming interfaces* (APIs). The *Java Collections API* is a set of classes that represent a few specific types of collections, implemented in various ways.

You might ask why we should learn how to design and implement collections if a set of collections has already been provided for us. There are several reasons. First, the Java Collections API provides only a subset of the collections you may want to use. Second, the classes that are provided may not implement the collections in the ways you desire. Third, and perhaps most important, the study of software development requires a deep understanding of the issues involved in the design of collections and the data structures used to implement them.

As we explore various types of collections, we will also examine the appropriate classes of the Java Collections API. In each case, we will analyze the various implementations that we develop and compare them to the approach used by the classes in the standard library.

3.2 A Stack Collection

Let's look at an example of a collection. A *stack* is a linear collection whose elements are added and removed from the same end. We say that a stack is processed in a *last in, first out* (LIFO) manner. That is, the last element to be put on a stack will be the first one that gets removed. Said another way, the elements of a stack are removed in the reverse order of their placement on it. In fact, one of the principal uses of a stack in computing is to reverse the order of something (e.g., an undo operation).

The processing of a stack is shown in Figure 3.3. Usually a stack is depicted vertically, and we refer to the end to which elements are added and from which they are removed as the *top* of the stack.

Recall from our earlier discussions that we define an abstract data type (ADT) by identifying a specific set of operations that establishes the valid ways in which we can manage the elements stored in the data structure. We always want to use this concept to formally define the operations for a collection and work within the functionality it provides. That way, we can cleanly separate the interface to the collection from any particular implementation technique used to create it.

KEY CONCEPT

Stack elements are processed in a LIFO manner—the last element in is the first element out.

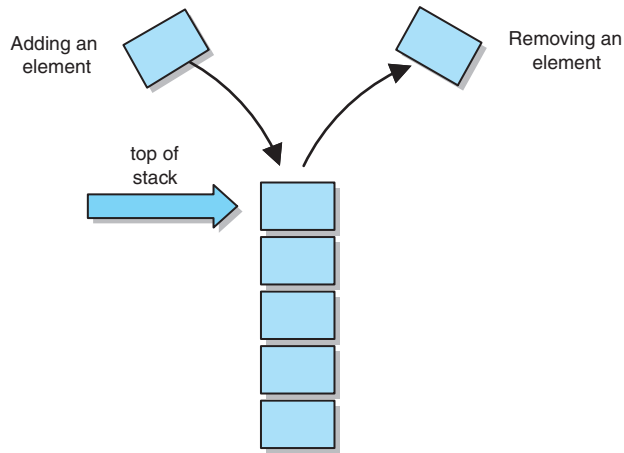


FIGURE 3.3 A conceptual view of a stack

The operations for a stack ADT are listed in Figure 3.4. In stack terminology, we *push* an element onto a stack and we *pop* an element off a stack. We can also *peek* at the top element of a stack, examining it or using it as needed, without actually removing it from the collection. And there are general operations that enable us to determine whether the stack is empty and, if it is not empty, how many elements it contains.

KEY CONCEPT

A programmer should choose the structure that is appropriate for the type of data management needed.

Sometimes there are variations on the naming conventions for the operations on a collection. For a stack, the use of the terms *push* and *pop* is relatively standard. The *peek* operation is sometimes referred to as *top*.

Operation	Description
<code>push</code>	Adds an element to the top of the stack.
<code>pop</code>	Removes an element from the top of the stack.
<code>peek</code>	Examines the element at the top of the stack.
<code>isEmpty</code>	Determines if the stack is empty.
<code>size</code>	Determines the number of elements on the stack.

FIGURE 3.4 The operations on a stack

DESIGN FOCUS

In the design of the stack ADT, we see the separation between the role of the stack and the role of the application that is using the stack. Notice that any implementation of this stack ADT is expected to throw an exception if a `pop` or `peek` operation is requested on an empty stack. The role of the collection is not to determine how such an exception is handled but merely to report it to the application using the stack. Similarly, the concept of a full stack does not exist in the stack ADT. Thus, it is the role of the stack collection to manage its own storage to eliminate the possibility of being full.

Keep in mind that the definition of a collection is not universal. You will find variations in the operations defined for specific data structures from one text to another. We've been very careful in this text to define the operations on each collection so that they are consistent with its purpose.

For example, note that none of the stack operations in Figure 3.4 enables us to reach down into the stack to modify, remove, or reorganize the elements in the stack. That is the very nature of a stack—all activity occurs at one end. If we discover that, to solve a particular problem, we need to access the elements in the middle or at the bottom of the collection, then a stack is not the appropriate collection to use.

We do provide a `toString` operation for the collection. This is not a classic operation defined for a stack, and it could be argued that this operation violates the prescribed behavior of a stack. However, it provides a convenient means to traverse and display the stack's contents without allowing modification of the stack, and this is quite useful for debugging purposes.

3.3 Crucial OO Concepts

Now let's consider what we will store in our stack. One possibility would be to simply re-create our stack data structure each time we need it and create it to store the specific object type for that application. For example, if we needed a stack of strings, we would simply copy and paste our stack code and change the object type to `String`. Even though copy, paste, and modify is technically a form of reuse, this brute force type of reuse is not our goal. Reuse, in its purest form, should mean that we create a collection that is written once, is compiled into bytecode once, and will then handle any objects we choose to store in it safely, efficiently, and effectively. To accomplish these goals, we must take *type compatibility* and

type checking into account. Type compatibility indicates whether a particular assignment of an object to a reference is legal. For example, the following assignment is not legal because you cannot assign a reference declared to be of type `String` to point to an object of type `Integer`.

```
String x = new Integer(10);
```

Java provides compile-time type checking that will flag this invalid assignment. A second possibility is to take advantage of the concepts of *inheritance* and *polymorphism* to create a collection that can store objects of any class.

Inheritance and Polymorphism

A complete discussion of the concepts of inheritance and polymorphism is provided in Appendix B. To review, a *polymorphic reference* is a reference variable that can refer to different types of objects at different points in time. Inheritance can be used to create a class hierarchy where a reference variable can be used to point to any object related to it by inheritance.

Carrying this to the extreme, an `Object` reference can be used to refer to any object, because ultimately all classes are descendants of the `Object` class. An `ArrayList`, for example, uses polymorphism in that it is designed to hold `Object` references. That's why an `ArrayList` can be used to store any kind of object. A particular `ArrayList` can hold several different types of objects at one time, because all objects are compatible with type `Object`.

The result of this discussion would seem to be that we could simply store `Object` references in our stack and take advantage of polymorphism via inheritance to create a collection that can store any type of objects. However, this possible solution creates some unexpected consequences. Because in this chapter we focus on implementing a stack with an array, let's examine what can happen when we are dealing with polymorphic references and arrays. Consider the classes represented in Figure 3.5. Since `Animal` is a superclass of all of the other classes in this diagram, an assignment such as the following is allowable:

```
Animal creature = new Bird();
```

However, this also means that the following assignments will compile as well:

```
Animal[] creatures = new Mammal[];
creatures[1] = new Reptile();
```

Note that by definition, `creatures[1]` should be both a `Mammal` and an `Animal`, but not a `Reptile`. This code will compile but will generate a `java.lang.ArrayStoreException` at run-time. Thus, because using the `Object` class will not provide us with compile-time type checking, we should look for a better solution.

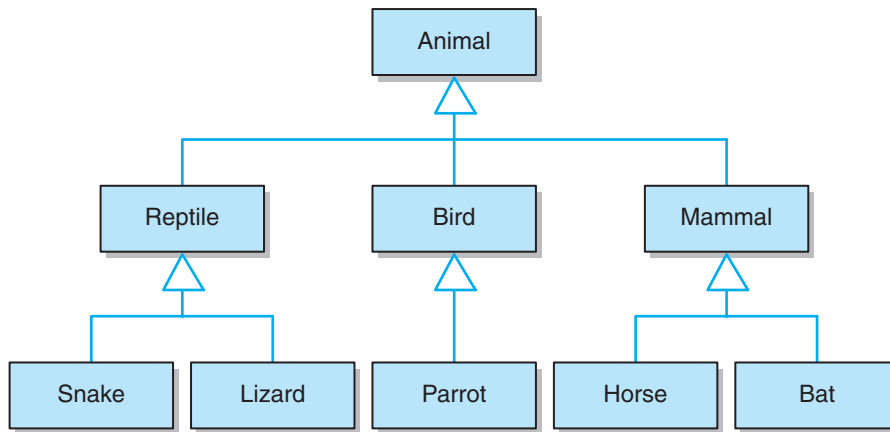


FIGURE 3.5 A UML class diagram showing a class hierarchy

Generics

Beginning with Java 5.0, Java enables us to define a class based on a *generic type*. That is, we can define a class so that it stores, operates on, and manages objects whose type is not specified until the class is instantiated. Generics are an integral part of our discussions of collections and their underlying implementations throughout the rest of this text.

Let's assume we need to define a class called `Box` that stores and manages other objects. As we discussed, using polymorphism, we could simply define `Box` so that internally it stores references to the `Object` class. Then, any type of object could be stored inside a `Box`. In fact, multiple types of unrelated objects could be stored in `Box`. We lose a lot of control with that level of flexibility in our code.

A better approach is to define the `Box` class to store a generic type `T`. (We can use any identifier we want for the generic type, but using `T` has become a convention.) The header of the class contains a reference to the type in angle brackets. For example:

```
class Box<T>
{
    // declarations and code that manage objects of type T
}
```

Then, when a `Box` is needed, it is instantiated with a specific class used in place of `T`. For example, if we wanted a `Box` of `Widget` objects, we could use the following declaration:

```
Box<Widget> box1 = new Box<Widget>();
```


The type of the `box1` variable is `Box<Widget>`. In essence, for the `box1` object, the `Box` class replaces `T` with `Widget`. Now suppose we wanted a `Box` in which to store `Gadget` objects; we could make the following declaration:

```
Box<Gadget> box2 = new Box<Gadget>();
```

For `box2`, the `Box` class essentially replaces `T` with `Gadget`. So, although the `box1` and `box2` objects are both boxes, they have different types because the generic type is taken into account. This is a safer implementation, because at this point we cannot use `box1` to store gadgets (or anything else for that matter), nor could we use `box2` to store widgets. A generic type such as `T` cannot be instantiated. It is merely a placeholder to enable us to define the class that will manage a specific type of object that is established when the class is instantiated.

Given that we now have a mechanism using generic types for creating a collection that can be used to store any type of object safely and effectively, let's continue our discussion of the stack collection.

The following section explores in detail an example of using a stack to solve a problem.

3.4 Using Stacks: Evaluating Postfix Expressions

Traditionally, arithmetic expressions are written in *infix* notation, meaning that the operator is placed between its operands in the form

$$\langle operand \rangle \langle operator \rangle \langle operand \rangle$$

such as in the expression

$$4 + 5$$

When evaluating an infix expression, we rely on precedence rules to determine the order of operator evaluation. For example, the expression

$$4 + 5 * 2$$

evaluates to 14 rather than 18 because of the precedence rule that in the absence of parentheses, multiplication evaluates before addition.

In a *postfix* expression, the operator comes after its two operands. Therefore, a postfix expression takes the form

$$\langle operand \rangle \langle operand \rangle \langle operator \rangle$$

For example, the postfix expression

$$6 \ 9 \ -$$

is equivalent to the infix expression

$$6 - 9$$

A postfix expression is generally easier to evaluate than an infix expression because precedence rules and parentheses do not have to be taken into account. The order of the values and operators in the expression is sufficient to determine the result. For this reason, programming language compilers and run-time environments often use postfix expressions in their internal calculations.

The process of evaluating a postfix expression can be stated in one simple rule: *Scanning from left to right, apply each operation to the two operands immediately preceding it, and replace the operator with the result.* At the end we are left with the final value of the expression.

Consider the infix expression we looked at earlier:

$$4 + 5 * 2$$

In postfix notation, this expression would be written

$$4 \ 5 \ 2 \ * \ +$$

Let's use our evaluation rule to determine the final value of this expression. We scan from the left until we encounter the multiplication (*) operator. We apply this operator to the two operands immediately preceding it (5 and 2) and replace it with the result (10), which leaves us with

$$4 \ 10 \ +$$

Continuing our scan from left to right, we immediately encounter the plus (+) operator. Applying this operator to the two operands immediately preceding it (4 and 10) yields 14, which is the final value of the expression.

Let's look at a slightly more complicated example. Consider the following infix expression:

$$(3 * 4 - (2 + 5)) * 4 / 2$$

The equivalent postfix expression is

$$3 \ 4 \ * \ 2 \ 5 \ + \ - \ 4 \ * \ 2 \ /$$

Applying our evaluation rule results in

```

12 2 5 + - 4 * 2 /
then 12 7 - 4 * 2 /
then 5 4 * 2 /
then 20 2 /
then 10

```

Now let's consider the design of a program that will evaluate a postfix expression. The evaluation rule relies on being able to retrieve the previous two operands whenever we encounter an operator. Furthermore, a large postfix expression will have many operators and operands to manage. It turns out that a stack is the perfect collection to use in this case. The operations provided by a stack coincide nicely with the process of evaluating a postfix expression.

KEY CONCEPT

A stack is the ideal data structure to use when evaluating a postfix expression.

The algorithm for evaluating a postfix expression using a stack can be expressed as follows: Scan the expression from left to right, identifying each token (operator or operand) in turn. If it is an operand, push it onto the stack. If it is an operator, pop the top two elements off the stack, apply the operation to them, and push the result onto the stack. When we reach the end of the expression, the element remaining on the stack is the result of the expression. If at any point we attempt to pop two elements off the stack but there are not two elements on the stack, then our postfix expression was not properly formed. Similarly, if we reach the end of the expression and more than one element remains on the stack, then our expression was not well formed. Figure 3.6 depicts the use of a stack to evaluate a postfix expression.

The `PostfixTester` program in Listing 3.1 evaluates multiple postfix expressions entered by the user. It uses the `PostfixEvaluator` class shown in Listing 3.2.

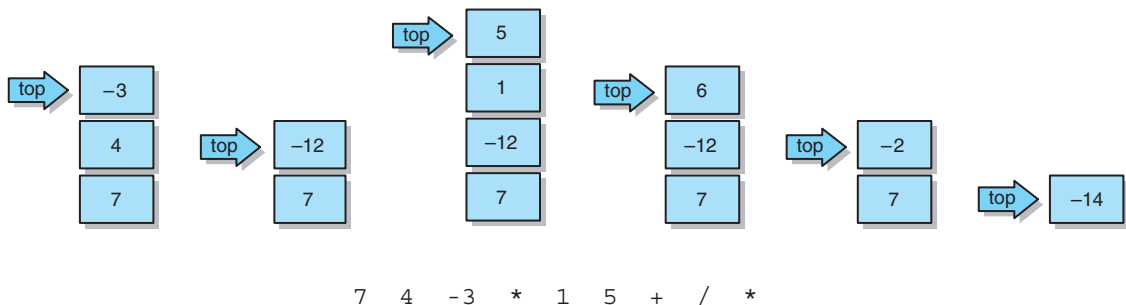


FIGURE 3.6 Using a stack to evaluate a postfix expression

To keep things simple, this program assumes that the operands to the expression are integers and are literal values (not variables). When executed, the program repeatedly accepts and evaluates postfix expressions until the user chooses not to.

LISTING 3.1

```
import java.util.Scanner;

/**
 * Demonstrates the use of a stack to evaluate postfix expressions.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class PostfixTester
{
    /**
     * Reads and evaluates multiple postfix expressions.
     */
    public static void main(String[] args)
    {
        String expression, again;
        int result;

        Scanner in = new Scanner(System.in);

        do
        {
            PostfixEvaluator evaluator = new PostfixEvaluator();
            System.out.println("Enter a valid post-fix expression one token " +
                "at a time with a space between each token " +
                "(e.g. 5 4 + 3 2 1 - + *)");

            System.out.println("Each token must be an integer or an " +
                "operator (+, -, *, /)");
            expression = in.nextLine();
            result = evaluator.evaluate(expression);
            System.out.println();
            System.out.println("That expression equals " + result);
            System.out.print("Evaluate another expression [Y/N]? ");
            again = in.nextLine();
            System.out.println();
        }
        while (again.equalsIgnoreCase("y"));
    }
}
```

LISTING 3.2

```

import java.util.Stack;
import java.util.Scanner;

/**
 * Represents an integer evaluator of postfix expressions. Assumes
 * the operands are constants.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class PostfixEvaluator
{
    private final static char ADD = '+';
    private final static char SUBTRACT = '-';
    private final static char MULTIPLY = '*';
    private final static char DIVIDE = '/';
    private Stack<Integer> stack;

    /**
     * Sets up this evaluator by creating a new stack.
     */
    public PostfixEvaluator()
    {
        stack = new Stack<Integer>();
    }

    /**
     * Evaluates the specified postfix expression. If an operand is
     * encountered, it is pushed onto the stack. If an operator is
     * encountered, two operands are popped, the operation is
     * evaluated, and the result is pushed onto the stack.
     * @param expr string representation of a postfix expression
     * @return value of the given expression
     */
    public int evaluate(String expr)
    {
        int op1, op2, result = 0;
        String token;
        Scanner parser = new Scanner(expr);
        while (parser.hasNext())
        {
            token = parser.next();
            if (isOperator(token))

```

LISTING 3.2 *continued*

```
        {
            op2 = (stack.pop()).intValue();
            op1 = (stack.pop()).intValue();
            result = evaluateSingleOperator(token.charAt(0), op1, op2);
            stack.push(new Integer(result));
        }
        else
            stack.push(new Integer(Integer.parseInt(token)));
    }
    return result;
}

/**
 * Determines if the specified token is an operator.
 * @param token the token to be evaluated
 * @return true if token is operator
 */
private boolean isOperator(String token)
{
    return ( token.equals("+") || token.equals("-") ||
            token.equals("*") || token.equals("/") );
}

/**
 * Performs integer evaluation on a single expression consisting of
 * the specified operator and operands.
 * @param operation operation to be performed
 * @param op1 the first operand
 * @param op2 the second operand
 * @return value of the expression
 */
private int evaluateSingleOperator(char operation, int op1, int op2)
{
    int result = 0;
    switch (operation)
    {
        case ADD:
            result = op1 + op2;
            break;
        case SUBTRACT:
            result = op1 - op2;
            break;
        case MULTIPLY:
```

LISTING 3.2 *continued*

```
        result = op1 * op2;
        break;
    case DIVIDE:
        result = op1 / op2;
    }
    return result;
}
```

The `PostfixEvaluator` class uses the `java.util.Stack` class to create the stack attribute. The `java.util.Stack` class is one of two stack implementations provided by the Java Collections API. We revisit the other implementation, the `Deque` interface, in Chapter 4.

The `evaluate` method performs the evaluation algorithm described earlier, supported by the `isOperator` and `evalSingleOp` methods. Note that in the `evaluate` method, only operands are pushed onto the stack. Operators are used as they are encountered and are never put on the stack. This is consistent with the evaluation algorithm we discussed. An operand is put on the stack as an `Integer` object, instead of as an `int` primitive value, because the stack data structure is designed to store objects.

When an operator is encountered, the two most recent operands are popped off the stack. Note that the first operand popped is actually the second operand in the expression and that the second operand popped is the first operand in the expression. This order doesn't matter in the cases of addition and multiplication, but it certainly matters for subtraction and division.

Note also that the postfix expression program assumes that the postfix expression entered is valid, meaning that it contains a properly organized set of operators and operands. A postfix expression is invalid if either (1) two operands are not available on the stack when an operator is encountered or (2) there is more than one value on the stack when the tokens in the expression are exhausted. Either situation indicates that there was something wrong with the format of the expression, and both can be caught by examining the state of the stack at the appropriate point in the program. We will discuss how we might deal with these situations and other exceptional cases in the next section.

Perhaps the most important aspect of this program is the use of the class that defined the stack collection. At this point, we don't know how the stack was implemented. We simply trusted the class to do its job. In this example, we used the class

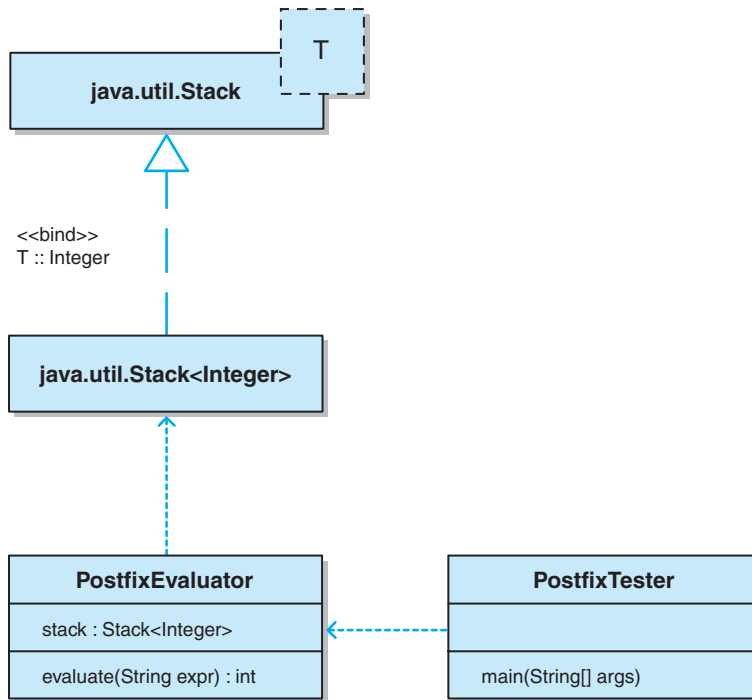


FIGURE 3.7 UML class diagram for the postfix expression evaluation program

`java.util.Stack`, but we could have used any class that implemented a stack, as long as it performed the stack operations as expected. From the point of view of evaluating postfix expressions, the manner in which the stack is implemented is largely irrelevant. Figure 3.7 shows a UML class diagram for the postfix expression evaluation program. The diagram illustrates that the `PostfixEvaluator` class uses an `Integer` instance of the `java.util.Stack` class and represents the binding of the `Integer` to the generic type `T`. We will not always include this level of detail in our UML diagrams.

Javadoc

Before moving on, let's mention the documentation style used for comments in Listings 3.1 and 3.2. These are *Javadoc comments*, which are written in a format that allows the javadoc tool (part of the JDK tool set) to parse the comments and extract information about the classes and methods. Javadoc comments begin with a `/**` and end with a `*/`.

Javadoc is used to create online documentation in HTML about a set of classes. You’ve already seen the results; the online Java API documentation is created using this technique. When changes are made to the API classes (and their comments), the javadoc tool is run again to generate the documentation. It’s a clever way to ensure that the documentation does not lag behind the evolution of the code.

There’s nothing special about the Java API classes in this regard. Documentation for any program or set of classes can be generated using Javadoc. And even if it’s not used to generate online documentation, the Javadoc commenting style is the official standard for adding comments to Java code.

Javadoc tags are used to identify particular types of information. For example, the `@author` tag is used to identify the programmer who wrote the code. The `@version` tag is used to specify the version number of the code. In the header of a method, the `@return` tag is used to indicate what value is returned by the method, and the `@param` tag is used to identify each parameter that’s passed to the method.

We won’t discuss Javadoc further at this point, but we use the Javadoc commenting style throughout this text.

Javadoc for a Method

```

/**
 * Retrieves the count of ...
 * @param cat the category to match
 * @return the number of ...
 */
public int getCount(Category cat)
{ ... }

```

Diagram annotations:

- `/**` begins a Javadoc comment
- `Retrieves the count of ...` method description
- `@param` tags
- `@return` tags

3.5 Exceptions

One concept that we will explore with each of the collections we discuss is that of exceptional behavior. What action should the collection take in the exceptional case? There are some such cases that are inherent in the collection itself. For example, in the case of a stack, what should happen if an attempt is made to pop an element from an empty stack? In this case, it does not matter what data structure is being used to implement the collection; the exception will still apply. Some such

cases are artifacts of the data structure being used to implement the collection. For example, if we are using an array to implement a stack, what should happen if an attempt is made to push an element onto the stack but the array is full? Let's take a moment to explore this concept further.

Problems that arise in a Java program may generate exceptions or errors. An *exception* is an object that defines an unusual or erroneous situation. An exception is thrown by a program or the run-time environment, and it can be caught and handled appropriately if desired. An *error* is similar to an exception, except that an error generally represents an unrecoverable situation, and it should not be caught. Java has a predefined set of exceptions and errors that may occur during the execution of a program.

In our postfix evaluation example, there were several potential exceptional situations. For instance:

- If the stack were full on a push
- If the stack were empty on a pop
- If the stack held more than one value at the completion of the evaluation

Let's consider each of these separately. The possibility that the stack might be full on a push is an issue for the underlying data structure, not the collection. Conceptually speaking, there is no such thing as a full stack. Now we know that this is not reality and that all data structures will eventually reach a limit. However, even when this physical limit is reached, the stack is not full; only the data structure that implements the stack is full. We will discuss strategies for handling this situation as we implement our stack in the next section.

What if the stack is empty on a pop? This is an exceptional case that has to do with the problem, not the underlying data structure. In our postfix evaluation example, if we attempt to pop two operands and there are not two operands available on the stack, our postfix expression was not properly formed. This is a case where the collection needs to report the exception, and the application then must interpret that exception in context.

The third case is equally interesting. What if the stack holds more than one value at the completion of the evaluation? From the perspective of the stack collection, this is not an exception. However, from the perspective of the application, this is a problem that means, once again, that the postfix expression was not well formed. Because it will not generate an exception from the collection, this is a condition for which the application must test.

Appendix B includes a complete discussion of exceptions and exception handling, including exception propagation and the try/catch statement. As we explore particular implementation techniques for a collection, we will also discuss the appropriate use of exceptions.

KEY CONCEPT

Errors and exceptions represent unusual or invalid processing.

3.6 A Stack ADT

KEY CONCEPT

A Java interface defines a set of abstract methods and is useful in separating the concept of an abstract data type from its implementation.

To facilitate separation of the interface operations from the methods that implement them, we can define a Java interface structure for a collection. A Java interface provides a formal mechanism for defining the set of operations for any collection.

Recall that a Java interface defines a set of abstract methods, specifying each method's signature but not its body. A class that implements an interface provides definitions for the methods defined in the interface. The interface name can be used as the type of a reference, which can be assigned any object of any class that implements the interface.

Listing 3.3 defines a Java interface for a stack collection. We name a collection interface using the collection name followed by the abbreviation ADT (for abstract data type). Thus, `StackADT.java` contains the interface for a stack collection. It is defined as part of the `jsjf` package, which contains all of the collection classes and interfaces presented in this text.

KEY CONCEPT

By using the interface name as a return type, we ensure that the interface doesn't commit the method to the use of any particular class that implements a stack.

Note that the stack interface is defined as `StackADT<T>`, operating on a generic type `T`. In the methods of the interface, the type of various parameters and return values is often expressed using the generic type `T`. When this interface is implemented, it will be based on a type that is substituted for `T`.

LISTING 3.3

```
package jsjf;

/**
 * Defines the interface to a stack collection.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public interface StackADT<T>
{
    /**
     * Adds the specified element to the top of this stack.
     * @param element element to be pushed onto the stack
     */
    public void push(T element);

    /**
     * Removes and returns the top element from this stack.
     */
}
```

LISTING 3.3 *continued*

```

    * @return the element removed from the stack
    */
    public T pop();

    /**
     * Returns without removing the top element of this stack.
     * @return the element on top of the stack
     */
    public T peek();

    /**
     * Returns true if this stack contains no elements.
     * @return true if the stack is empty
     */
    public boolean isEmpty();

    /**
     * Returns the number of elements in this stack.
     * @return the number of elements in the stack
     */
    public int size();

    /**
     * Returns a string representation of this stack.
     * @return a string representation of the stack
     */
    public String toString();
}

```

Stack Interface

```

    public interface StackADT <T>
    {
        ...
    }

```

reserved word interface name
 | |
 public interface StackADT <T> — generic type parameter
 {
 ... — list of method signatures
 }

Each time we introduce an interface, a class, or a system in this text, we will accompany that description with the UML description of that interface, class, or system. This should help you become accustomed to reading UML descriptions and to creating them for other classes and systems. Figure 3.8 illustrates the UML description of the `StackADT` interface.

Stacks are used quite frequently in the computing world. For example, the undo operation in a word processor is usually implemented using a stack. As we make changes to a document (add data, delete data, make format changes, etc.), the word processor keeps track of each operation by pushing some representation of it onto a stack. If we choose to undo an operation, the word processing software pops the most recently performed operation off the stack and reverses it. If we choose to undo again (undoing the second-to-last operation we performed), another element is popped from the stack. In most word processors, many operations can be reversed in this manner.

DESIGN FOCUS

Undo operations are often implemented using a special type of stack called a drop-out stack. The basic operations on a drop-out stack are the same as those for a stack (`push`, `pop`, and `peek`). The only difference is that a drop-out stack has a limit to the number of elements it will hold, and once that limit is reached, the element on the bottom of the stack drops off the stack when a new element is pushed on. The development of a drop-out stack is left as an exercise.

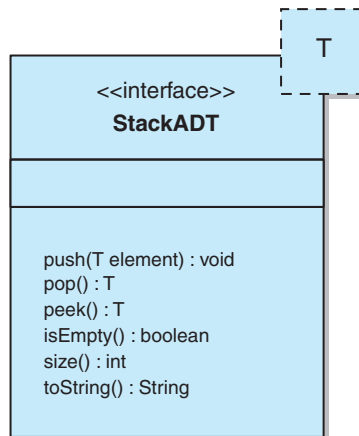


FIGURE 3.8 The `StackADT` interface in UML

3.7 Implementing a Stack: With Arrays

So far in our discussion of a stack collection, we have described its basic conceptual nature and the operations that enable the user to interact with it. In software engineering terms, we would say that we have done the analysis for a stack collection. We have also used a stack, without knowing the details of how it was implemented, to solve a particular problem. Now let's turn our attention to the implementation details. There are various ways to implement a class that represents a stack. As mentioned earlier, the Java Collections API provides multiple implementations, including the `Stack` class and the `Deque` interface. In this section, we examine an implementation strategy that uses an array to store the objects contained in the stack. In the next chapter, we examine a second technique for implementing a stack.

To explore this implementation, we must recall several key characteristics of Java arrays. The elements stored in an array are indexed from 0 to $n-1$, where n is the total number of cells in the array. An array is an object, which is instantiated separately from the objects it holds. And when we talk about an array of objects, we are actually talking about an array of references to objects, as illustrated in Figure 3.9.

Keep in mind the separation between the collection and the underlying data structure used to implement it. Our goal is to design an efficient implementation that provides the functionality of every operation defined in the stack abstract data type. The array is just a convenient data structure in which to store the objects.

KEY CONCEPT

The implementation of the collection operations should not affect the way users interact with the collection.

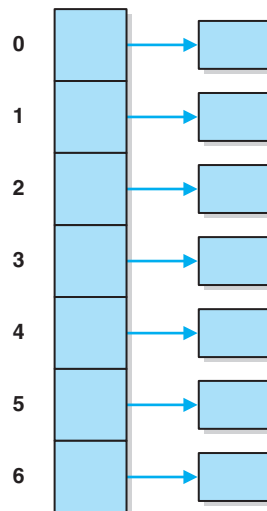


FIGURE 3.9 An array of object references

Managing Capacity

When an array object is created, it is allocated a specific number of cells into which elements can be stored. For example, the following instantiation creates an array that can store 500 elements, indexed from 0 to 499:

```
Object[] collection = Object[500];
```

The number of cells in an array is called its *capacity*. This value is stored in the `length` constant of the array. The capacity of an array cannot be changed once the array has been created.

When using an array to implement a collection, we have to deal with the situation in which all cells of the array are being used to store elements. That is, because we are using a fixed-size data structure, at some point the data structure may become “full.” However, just because the data structure is full, should that mean that the collection is full?

A crucial question in the design of a collection is what to do in the case in which a new element is added to a full data structure. Three basic options exist:

- We could implement operations that add an element to the collection such that they throw an exception if the data structure is full.
- We could implement the `add` operations to return a status indicator that can be checked by the user to see whether the `add` operation was successful.
- We could automatically expand the capacity of the underlying data structure whenever necessary so that, essentially, it would never become full.

KEY CONCEPT

How we handle exceptional conditions determines whether the collection or the user of the collection controls the particular behavior.

In the first two cases, the user of the collection must be aware that the collection could get full and must take steps to deal with it when needed. For these solutions, we would also provide extra operations that enable the user to check whether the collection is full and to expand the capacity of the data structure as desired. The advantage of these approaches is that they give the user more control over the capacity.

However, given that our goal is to separate the interface from the implementation, the third option is attractive. The capacity of the underlying data structure is an implementation detail that, in general, should be hidden from the user. Furthermore, the capacity issue is particular to this implementation. Other techniques used to implement the collection, such as the one we explore in the next chapter, are not restricted by a fixed capacity and therefore never have to deal with this issue.

In the solutions presented in this text, we opt to implement fixed data structure solutions by automatically expanding the capacity of the underlying data structure. Occasionally, other options are explored as programming projects.

3.8 The ArrayStack Class

In the Java Collections API framework, class names indicate both the underlying data structure and the collection. We follow that naming convention in this text. Thus, we define a class called `ArrayStack` to represent a stack with an underlying array-based implementation.

To be more precise, we define a class called `ArrayStack<T>` that represents an array-based implementation of a stack collection that stores objects of generic type `T`. When we instantiate an `ArrayStack` object, we specify what the generic type `T` represents.

An array implementation of a stack can be designed by making the following four assumptions: The array is an array of object references (type determined when the stack is instantiated), the bottom of the stack is always at index 0 of the array, the elements of the stack are stored in order and contiguously in the array, and there is an integer variable `top` that stores the index of the array immediately following the top element in the stack.

Figure 3.10 illustrates this configuration for a stack that currently contains the elements A, B, C, and D, assuming that they have been pushed on in that order. To simplify the figure, the elements are shown in the array itself rather than as objects referenced from the array. Note that the variable `top` represents both the next cell into which a pushed element should be stored and the count of the number of elements currently in the stack.

In this implementation, the bottom of the stack is always held at index 0 of the array, and the stack grows and shrinks at the higher indexes. This is considerably more efficient than if the stack were reversed within the array. Consider the processing that would be necessary if the top of the stack were kept at index 0.

From these assumptions, we can determine that our class will need a constant to store the default capacity, a variable to keep track of the top of the stack, and a variable for the array to store the stack. This results in the class header shown on

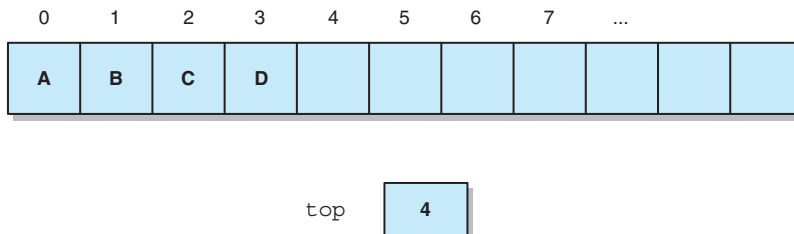


FIGURE 3.10 An array implementation of a stack

KEY CONCEPT

For efficiency, an array-based stack implementation keeps the bottom of the stack at index 0.



VideoNote

An overview of the `ArrayStack` implementation

the following page. Note that our `ArrayStack` class will be part of the `jsjf` package and will make use of a package called `jsjf.exceptions`.

```
package jsjf;

import jsjf.exceptions.*;
import java.util.Arrays;

/**
 * An array implementation of a stack in which the bottom of the
 * stack is fixed at index 0.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class ArrayStack<T> implements StackADT<T>
{
    private final static int DEFAULT_CAPACITY = 100;

    private int top;
    private T[] stack;
```

The Constructors

Our class will have two constructors, one to use the default capacity and the other to use a specified capacity.

```
/**
 * Creates an empty stack using the default capacity.
 */
public ArrayStack()
{
    this(DEFAULT_CAPACITY);
}

/**
 * Creates an empty stack using the specified capacity.
 * @param initialCapacity the initial size of the array
 */
public ArrayStack(int initialCapacity)
{
    top = 0;
    stack = (T[]) (new Object[initialCapacity]);
}
```

Just to refresh our memory, this is an excellent example of method overloading (that is, two methods with the same name that differ only in the parameter list). It is also interesting to note that the constructor for the default capacity makes use of the other constructor by passing it the `DEFAULT_CAPACITY` constant.

From our previous discussion of generics, we recall that you cannot instantiate a generic type. This also means that you cannot instantiate an array of a generic type. This results in an interesting line of code in our constructor:

```
stack = (T[]) (new Object [initialCapacity]);
```

Note that in this line, we are instantiating an array of `Object`s and then casting it as an array of our generic type. This will create a compile-time warning for an unchecked type conversion, because the Java compiler cannot guarantee the type safety of this cast. As we have seen, it is worth dealing with this warning to gain the flexibility and type safety of generics. This warning can be suppressed using the following Java annotation placed before the offending statement:

```
@SuppressWarnings ("unchecked")
```

COMMON ERROR

A common error made by programmers new to generics is to attempt to create an array of a generic type:

```
stack = new T [initialCapacity];
```

Generic types cannot be instantiated, and that includes arrays of a generic type. That's why we have to create an array that holds `Object` references and then cast it into an array of the generic type.

Creating an Array of Generic Elements

```
stack = (T[]) (new Object [initialCapacity]);
```

create an array of `Object`

cast as an array of generic type `T`

The push Operation

To push an element onto the stack, we simply insert it in the next available position in the array as specified by the variable `top`. Before doing so, however, we must determine whether the array has reached its capacity and expand it if necessary. After storing the value, we must update the value of `top` so that it continues to represent the number of elements in the stack.

Implementing these steps results in the following code:

```
/**
 * Adds the specified element to the top of this stack, expanding
 * the capacity of the array if necessary.
 * @param element generic element to be pushed onto stack
 */
public void push(T element)
{
    if (size() == stack.length)
        expandCapacity();

    stack[top] = element;
    top++;
}
```

The `expandCapacity` method is implemented to double the size of the array as needed. Of course, since an array cannot be resized once it is instantiated, this method simply creates a new, larger array and copies the contents of the old array into the new one. It serves as a support method of the class and can therefore be implemented with private visibility.

```
/**
 * Creates a new array to store the contents of this stack with
 * twice the capacity of the old one.
 */
private void expandCapacity()
{
    stack = Arrays.copyOf(stack, stack.length * 2);
}
```

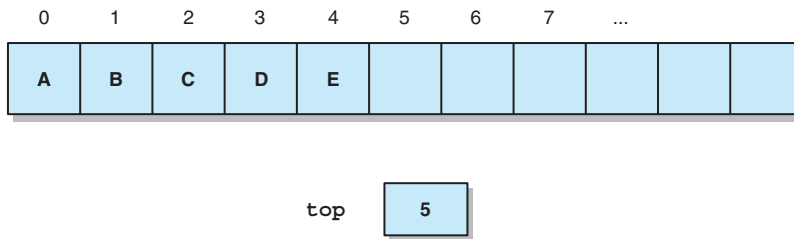


FIGURE 3.11 The stack after pushing element E

Figure 3.11 illustrates the result of pushing an element E onto the stack that was depicted in Figure 3.10.

The `push` operation for the array implementation of a stack consists of the following steps:

- Make sure that the array is not full.
- Set the reference in position `top` of the array to the object being added to the stack.
- Increment the values of `top`.

Each of these steps is $O(1)$. Thus the operation is $O(1)$. We might wonder about the time complexity of the `expandCapacity` method and about the impact it might have on the analysis of the `push` method. This method does contain a linear `for` loop, and intuitively, we would call that $O(n)$. However, given how seldom the `expandCapacity` method is called relative to the number of times `push` may be called, we can amortize that complexity across all instances of `push`.

The `pop` Operation

The `pop` operation removes and returns the element at the top of the stack. For an array implementation, that means returning the element at index `top-1`. Before attempting to return an element, however, we must ensure that there is at least one element in the stack to return.

The array-based version of the `pop` operation can be implemented as follows:

```
/**
 * Removes the element at the top of this stack and returns a
 * reference to it.
 * @return element removed from top of stack
```

```

    * @throws EmptyCollectionException if stack is empty
    */
    public T pop() throws EmptyCollectionException
    {
        if (isEmpty())
            throw new EmptyCollectionException("stack");

        top--;
        T result = stack[top];
        stack[top] = null;

        return result;
    }

```

If the stack is empty when the `pop` method is called, an `EmptyCollectionException` is thrown. Otherwise, the value of `top` is decremented, and the element stored at that location is stored into a temporary variable so that it can be returned. That cell in the array is then set to null. Note that `top` ends up with the appropriate value relative to the now smaller stack. Figure 3.12 illustrates the results of a `pop` operation on the stack from Figure 3.11, which brings it back to its earlier state (identical to Figure 3.10).

The `pop` operation for the array implementation consists of the following steps:

- Make sure the stack is not empty.
- Decrement the `top` counter.
- Set a temporary reference equal to the element in `stack[top]`.
- Set `stack[top]` equal to null.
- Return the temporary reference.

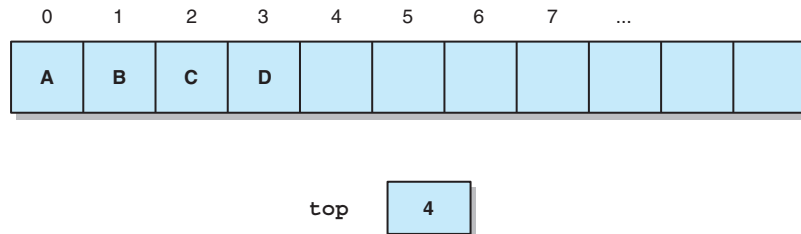


FIGURE 3.12 The stack after popping the top element

All of these steps are also $O(1)$. Thus, the `pop` operation for the array implementation has time complexity $O(1)$.

The peek Operation

The `peek` operation returns a reference to the element at the top of the stack without removing it from the array. For an array implementation, that means returning a reference to the element at position `top-1`. This one step is $O(1)$, and thus the `peek` operation is $O(1)$ as well.

```
/**
 * Returns a reference to the element at the top of this stack.
 * The element is not removed from the stack.
 * @return element on top of stack
 * @throws EmptyCollectionException if stack is empty
 */
public T peek() throws EmptyCollectionException
{
    if (isEmpty())
        throw new EmptyCollectionException("stack");

    return stack[top-1];
}
```

Other Operations

The `isEmpty`, `size`, and `toString` operations and their analysis are left as programming projects and exercises.

The EmptyCollectionException Class

Now that we have examined the implementation of our `ArrayStack` class, let's revisit our choices with respect to exception handling. We chose to have our collection handle the case where the underlying data structure becomes full, because that is an issue that is internal to the collection. On the other hand, we chose to throw an exception if an attempt is made to access an element in the collection through either a `pop` or a `peek` operation when the collection is empty. This situation reveals a problem with the use of the collection, not with the collection itself.

Exceptions are classes in Java, so we have the choice of using existing exceptions provided in the Java API or creating our own. In this case, we could have

chosen to create a specific empty stack exception. However, creating a parameterized exception enables us to reuse this exception with any of our collections classes. Listing 3.4 shows the `EmptyCollectionException` class. Notice that our exception class extends the `RuntimeException` class and then makes use of the parent's constructor by using a `super` reference.

LISTING 3.4

```
package jsjf.exceptions;

/**
 * Represents the situation in which a collection is empty.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class EmptyCollectionException extends RuntimeException
{
    /**
     * Sets up this exception with an appropriate message.
     * @param collection the name of the collection
     */
    public EmptyCollectionException(String collection)
    {
        super("The " + collection + " is empty.");
    }
}
```

Other Implementations

In this chapter we examined the concept of a stack, used the `Stack` class available in Java API to solve a postfix evaluation problem, and then implemented our own `ArrayStack` class that used an array to store the underlying elements on a stack.

We're not finished with stacks yet, though. In Chapter 4, we'll examine another technique for implementing collections using linked structures instead of arrays, and implement a `LinkedStack` class.

Armed with both of these two broad implementation techniques—array-based and linked-based—we'll be set to explore many other collections.

Summary of Key Concepts

- A collection is an object that gathers and organizes other objects.
- Elements in a collection are typically organized in terms of the order of their addition to the collection or in terms of some inherent relationship among the elements.
- A collection is an abstraction wherein the details of the implementation are hidden.
- A data structure is the underlying programming construct used to implement a collection.
- Stack elements are processed in a LIFO manner—the last element in is the first element out.
- A programmer should choose the structure that is appropriate for the type of data management needed.
- A stack is the ideal data structure to use when evaluating a postfix expression.
- Errors and exceptions represent unusual or invalid processing.
- A Java interface defines a set of abstract methods and is useful in separating the concept of an abstract data type from its implementation.
- By using the interface name as a return type, we ensure that the interface doesn't commit the method to the use of any particular class that implements a stack.
- A programmer must carefully consider how exceptions should be handled, if at all, and at what level.
- The implementation of the collection operations should not affect the way users interact with the collection.
- How we handle exceptional conditions determines whether the collection or the user of the collection controls the particular behavior.
- For efficiency, an array-based stack implementation keeps the bottom of the stack at index 0.

Summary of Terms

abstraction A point of view that hides or ignores certain details, usually to make concepts easier to manage.

abstract data type A data type whose values and operations are not inherently defined within a programming language.

class hierarchy The relationship among classes created by inheritance in which the child of one parent can itself be the parent of other classes.

collection An object that gathers and organizes other objects.

data structure (1) An organization of objects that allows certain operations to be performed efficiently; (2) The programming constructs used to implement a collection.

exception An object that defines an unusual or erroneous situation.

generic type A placeholder for an object type that is not made concrete until the class that refers to it is instantiated.

inheritance The object-oriented principle of deriving one class from an existing class.

interface (1) The manner in which one thing interacts with another; (2) A set of public methods that enables one object to interact with another.

Java Collections API The subset of the Java application programming interfaces (APIs) that represent or deal with collections.

LIFO (1) Last in, first out; (2) A description of a collection in which the last element added will be the first element removed.

polymorphism The object-oriented principle that enables a reference variable to point to related but distinct types of objects over time, and in which method invocations are bound to code at run-time.

pop A stack operation in which an element is removed from the top of a stack.

push A stack operation in which an element is added to the top of a stack.

stack A linear collection whose elements are added and removed from the same end in a LIFO manner.

Self-Review Questions

- SR 3.1 How are elements organized in a collection?
- SR 3.2 Give two real world examples of nonlinear collection.
- SR 3.3 What is the difference between data type and abstract data type?
- SR 3.4 Point out the data type, collection, and the data structure you would use in implementing an array-based stack of integers.
- SR 3.5 What do you understand by the statement: ‘Stack elements are processed in a LIFO manner’?
- SR 3.6 Which data structure would be ideal to implement recursion of methods?

- SR 3.7 Name the methods used to add an element to and remove an element from a stack.
- SR 3.8 What is a Java Interface?
- SR 3.9 Can an interface be called a class?
- SR 3.10 What do you understand by a generic type?
- SR 3.11 What is *polymorphic reference*?
- SR 3.12 Given the example in Figure 3.5, list the subclasses of Reptile.
- SR 3.13 Given the example in Figure 3.5, will the following code compile?
- ```
Animal creature = new Bat();
```
- SR 3.14 Given the example in Figure 3.5, will the following code compile?
- ```
Parrot creature = new Bird();
```
- SR 3.15 What is the difference between infix and postfix expressions?
- SR 3.16 How do you evaluate a postfix notation?

Exercises

- EX 3.1 Object-oriented programming allows you to define and compile a general form of a class. Later you can define a specialized version of this class, starting with the original class. Which particular OOPs technique does this depict?
- EX 3.2 Why is an object used as the perfect mechanism for creating a collection?
- EX 3.3 What type of collection is a stack? Give one natural example of the usage of stacks in computer science.
- EX 3.4 Hand trace an initially empty stack X through the following operations:

```
X.push(new Integer(20));
  X.push(new Integer(15));
  Integer Y = X.pop();
  X.push(new Integer(35));
  X.push(new Integer(10));
X.push(new Integer(25));
X.push(new Integer(45));
Integer Y = X.pop();
X.push(new Integer(15));
X.push(new Integer(45));
```

- EX 3.5 Given the resulting stack `X` from the previous exercise, what would be the result of each of the following?
- `Y = X.peek();`
 - `Y = X.pop();`
`Z = X.peek();`
 - `Y = X.push(30);`
`Z = X.peek();`
- EX 3.6 Write Java statements to define a `Bag` class to store a generic type `T`. The class should have two generic methods for setting and retrieving data.
- EX 3.7 Write Java statements to instantiate the `Bag` class created in the above example, for storing `String` type data items. Store a value `Money` as a data item.
- EX 3.8 When you type a line of text on a keyboard, you use the backspace key to remove the previous character if you have made a mistake. Consecutive application of the backspace key thus erases several characters. Show how the use of backspace key can be supported by the use of a stack. Give specific examples and draw the contents of the stack after various actions are taken.
- EX 3.9 Draw an example using the five integers (21, 32, 11, 54, 90) showing how a stack could be used to reverse the order (90, 54, 11, 32, 21) of these elements.
- EX 3.10 Convert the following infix expressions to postfix form by using the algorithm discussed in this chapter.
- $(a * (b / c)) - d + e / f$
 - $a - (b * c / d) + e$

Programming Projects

- PP 3.1 Complete the implementation of the `ArrayStack` class presented in this chapter. Specifically, complete the implementations of the `isEmpty`, `size`, and `toString` methods.
- PP 3.2 Design and implement an application that reads a sentence from the user and prints the sentence with the characters of each word backward. Use a stack to reverse the characters of each word.
- PP 3.3 Modify the solution to the postfix expression evaluation problem so that it checks for the validity of the expression that is entered by the user. Issue an appropriate error message when an erroneous situation is encountered.

- PP 3.4 The array implementation in this chapter keeps the top variable pointing to the next array position above the actual top of the stack. Rewrite the array implementation such that `stack[top]` is the actual top of the stack.
- PP 3.5 There is a data structure called a drop-out stack that behaves like a stack in every respect except that if the stack size is n , when the $n + 1$ element is pushed, the first element is lost. Implement a drop-out stack using an array. (*Hint: A circular array implementation would make sense.*)
- PP 3.6 Implement an integer adder using three stacks.
- PP 3.7 Implement an infix-to-postfix translator using stacks.
- PP 3.8 Implement a class called `reverse` that uses a stack to output a set of elements input by the user in reverse order.
- PP 3.9 Create a graphical application that provides a button for push and pop from a stack, a text field to accept a string as input for push, and a text area to display the contents of the stack after each operation.

Answers to Self-Review Questions

- SRA 3.1 Elements in a *collection* are typically organized in terms of the order of their addition to the collection, or in terms of some inherent relationship among the elements.
- SRA 3.2 The organizational hierarchy of a company, and the flight map of an airline.
- SRA 3.3 A *data type* is a group of values and the operations defined on those values. An *abstract data type (ADT)* is a data type whose values and operations are not inherently defined within a programming language. The details of its implementation must be defined and should be hidden from the user.
- SRA 3.4 The data type is `int` (primitive data type for storing integers), the collection is `stack`, and data structure is `array type`.
- SRA 3.5 The last element to be put on a stack will be the first one that gets removed. In other words, the elements of a stack are removed in the reverse order of their placement on it.
- SRA 3.6 A `Stack` data structure, as it would be used to store the activation records of the method.
- SRA 3.7 As per stack terminology, the *push* method adds an element to the top of the stack. And the *pop* method removes an element from the top of the stack.

- SRA 3.8 A Java interface defines a set of abstract methods and is useful in separating the concept of an abstract data type from its implementation.
- SRA 3.9 No an interface is not a class. However, it is like an extreme case of an abstract class. An interface is basically a type, which allows a programmer to define methods with parameters of an interface type. This code applies to all the classes that implement the interface.
- SRA 3.10 A generic type is a placeholder for an object type that is not made concrete until the class that refers to it is instantiated.
- SRA 3.11 A *polymorphic reference* is a reference variable that can refer to different types of objects at different points in time. Inheritance can be used to create a class hierarchy where a reference variable can be used to point to any object related to it by inheritance.
- SRA 3.12 The subclasses of Reptile are Snake and Lizard.
- SRA 3.13 Yes, a reference variable of a parent class or any superclass may hold a reference to one of its descendants.
- SRA 3.14 No, a reference variable for a child or subclass may not hold a reference to a parent or superclass. To make this assignment, you would have to explicitly cast the parent class into the child class
`(Parrot creature = (Parrot) (new Bird()));`
- SRA 3.15 In an *infix* expression, the operator is placed between its operands in the form: <operand> <operator> <operand>; such as in the expression `4 + 5`. Whereas, in a postfix expression, the operator comes after its two operands in the form: <operand> <operand> <operator>. The above example in postfix notation would be `4 5 +`.
- SRA 3.16 The process of evaluating a postfix expression can be stated as: Scanning from left to right, apply each operation to the two operands immediately preceding it, and replace the operator with the result. At the end we are left with the final value of the expression. A stack is the ideal data structure to use when evaluating a postfix expression.



Linked Structures – Stacks

4

This chapter explores a technique for creating data structures using references to create links between objects. Linked structures are fundamental to the development of software, especially the design and implementation of collections. This approach has both advantages and disadvantages when compared to a solution using arrays.

CHAPTER OBJECTIVES

- Describe the use of references to create linked structures.
- Compare linked structures to array-based structures.
- Explore the techniques for managing a linked list.
- Discuss the need for a separate node object to form linked structures.
- Implement a stack collection using a linked list.

4.1 References as Links

In Chapter 3, we discussed the concept of collections and explored one collection in particular: a stack. We defined the operations on a stack collection and designed an implementation using an underlying array-based data structure. In this chapter, we explore an entirely different approach to designing a data structure and use it to create another stack implementation.

A *linked structure* is a data structure that uses object reference variables to create links between objects. Linked structures are the primary alternative to an array-based implementation of a collection. After discussing various issues involved in linked structures, we will define a new implementation of a stack collection that uses an underlying linked data structure.

KEY CONCEPT

Object reference variables can be used to create linked structures.

Recall that an object reference variable holds the address of an object, indicating where the object is stored in memory. The following declaration creates a variable called `obj` that is only large enough to hold the numeric address of an object:

```
Object obj;
```

Usually the specific address that an object reference variable holds is irrelevant. That is, even though it is important to be able to use the reference variable to access an object, the specific location in memory where it is stored is unimportant. Therefore, instead of showing addresses, we usually depict a reference variable as a name that “points to” an object, as shown in Figure 4.1. A reference variable, used in this context, is sometimes called a *pointer*.

Consider the situation in which a class defines as instance data a reference to another object of the same class. For example, suppose we have a class named `Person` that contains a person’s name, address, and other relevant information. Now suppose that in addition to these data, the `Person` class contains a reference variable to another `Person` object:

```
public class Person
{
    private String name;
    private String address;

    private Person next; // a link to another Person object

    // whatever else
}
```

Using only this one class, we can create a linked structure. One `Person` object contains a link to a second `Person` object. This second object also contains a

reference to a `Person`, which contains another, and so on. This type of object is sometimes called *self-referential*.



FIGURE 4.1 An object reference variable pointing to an object

This kind of relationship forms the basis of a *linked list*, which is a linked structure in which one object refers to the next, creating a linear ordering of the objects in the list. A linked list is depicted in Figure 4.2. Often the objects stored in a linked list are referred to generically as the *nodes* of the list.

Note that a separate reference variable is needed to indicate the first node in the list. The list is terminated in a node whose `next` reference is `null`.

A linked list is only one kind of linked structure. If a class is set up to have multiple references to objects, a more complex structure can be created, such as the one depicted in Figure 4.3. The way in which the links are managed dictates the specific organization of the structure.

KEY CONCEPT
A linked list is composed of objects that each point to the next object in the list.

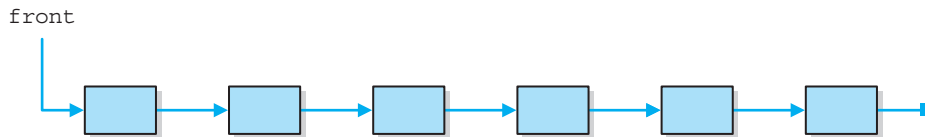


FIGURE 4.2 A linked list

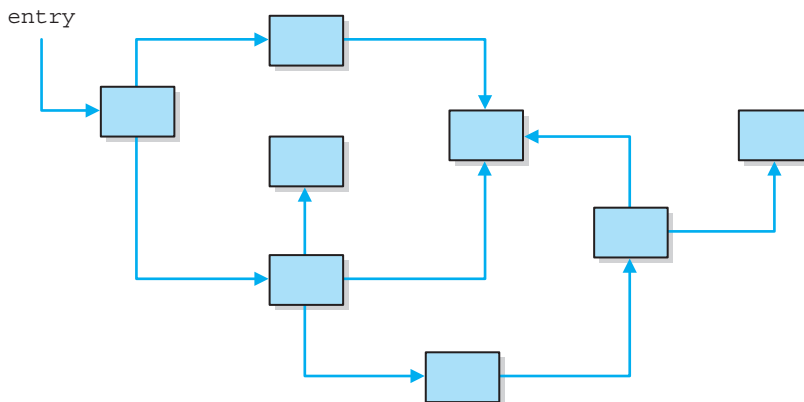


FIGURE 4.3 A complex linked structure

KEY CONCEPT

A linked list dynamically grows as needed and essentially has no capacity limitations.

For now, we will focus on the details of a linked list. Many of these techniques apply to more complicated linked structures as well.

Unlike an array, which has a fixed size, a linked list has no upper bound on its capacity other than the limitations of memory in the computer. A linked list is considered to be a *dynamic* structure because its size grows and shrinks as needed to accommodate the number of elements stored. In Java, all objects are created dynamically from an area of memory called the *system heap* or *free store*.

The next section explores some of the primary ways in which a linked list is managed.

4.2 Managing Linked Lists

Keep in mind that our goal is to use linked lists and other linked structures to create collections—specifically, in this chapter, to create a stack. Because the principal purpose of a collection is to be able to add, remove, and access elements, we must first examine how to accomplish these fundamental operations using links. We will focus our discussion in this chapter on adding and removing from the end of a linked list, as we will need to do for our stack. We will revisit this discussion as the situation warrants.

No matter what a linked list is used to store, there are a few basic techniques involved in managing the nodes in the list. Specifically, elements in the list are accessed, elements are inserted into the list, and elements are removed from the list.

Accessing Elements

Special care must be taken when dealing with the first node in the list so that the reference to the entire list is maintained appropriately. When using linked lists, we maintain a pointer to the first element in the list. To access other elements, we must access the first one and then follow the next pointer from that one to the next one, and so on. Consider our previous example of a `Person` class containing the attributes `name`, `address`, and `next`. If we wanted to find the fourth person in the list, and assuming that we had a variable `first` of type `Person` that pointed to the first person in the list and that the list contained at least four nodes, we might use the following code:

```
Person current = first;
for (int i = 0; i < 3; i++)
    current = current.next;
```

After executing this code, `current` will point to the fourth person in the list. Notice that it is very important to create a new reference variable, in this case

current, and then start by setting that reference variable to point to the first element in the list. Consider what would happen if we used the `first` pointer in the loop instead of `current`. Once we moved the `first` pointer to point to the second element in the list, we would no longer have a pointer to the first element and would not be able to access it. Keep in mind that with a linked list, the only way to access the elements in the list is to start with the first element and progress through the list.

Of course, a more likely scenario is that we would need to search our list for a particular person. Assuming that the `Person` class overrides the `equals` method such that it returns true if the given `String` matches the name stored for that person, the following code will search the list for Tom Jones:

```
String searchstring = "Tom Jones";
Person current = first;
while ((not(current.equals(searchstring)) && (current.next != null))
       current = current.next;
```

Note that this loop will terminate when the string is found or when the end of the list is encountered. Now that we have seen how to access elements in a linked list, let's consider how to insert elements into a list.

Inserting Nodes

A node may be inserted into a linked list at any location: at the front of the list, among the interior nodes in the middle of the list, or at the end of the list. Adding a node to the front of the list requires resetting the reference to the entire list, as shown in Figure 4.4. First, the next reference of the added node is set to point to the current first node in the list. Second, the reference to the front of the list is reset to point to the newly added node.

KEY CONCEPT

The order in which references are changed is crucial to maintaining a linked list.

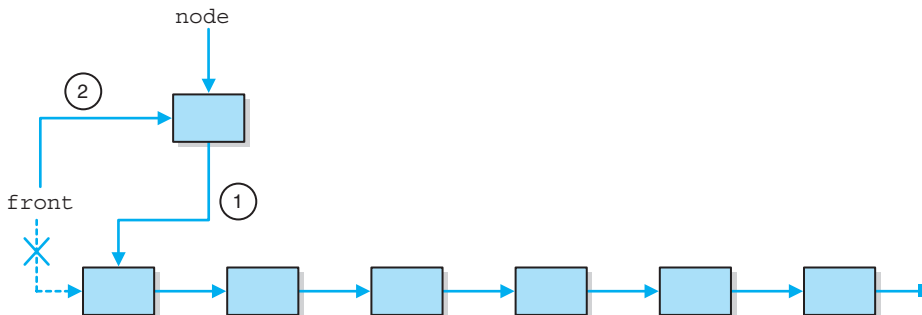


FIGURE 4.4 Inserting a node at the front of a linked list

Note that difficulties would arise if these steps were reversed. If we were to reset the `front` reference first, we would lose the only reference to the existing list, and it could not be retrieved.

Inserting a node into the middle of a list requires some additional processing but is not needed for our stack collection. We will come back to this topic in Chapter 6.

Deleting Nodes

Any node in the list can be deleted. We must maintain the integrity of the list no matter which node is deleted. As with the process of inserting a node, dealing with the first node in the list represents a special case.

To delete the first node in a linked list, we reset the reference to the front of the list so that it points to the current second node in the list. This process is shown in Figure 4.5. If the deleted node is needed elsewhere, we must set up a separate reference to it before resetting the `front` reference. The general case of deleting a node from the interior of the list is left for Chapter 6.

KEY CONCEPT

Dealing with the first node in a linked list often requires special handling.

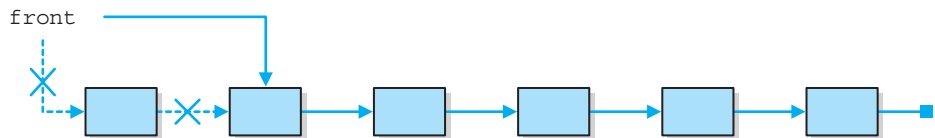


FIGURE 4.5 Deleting the first node in a linked list

DESIGN FOCUS

We've described insertion into and deletion from a list as having two cases: the case when dealing with the first node and the case when dealing with any other node. It is possible to eliminate the special case involving the first node by introducing a *sentinel node* or *dummy node* at the front of the list. A sentinel node serves as a false first node and doesn't actually represent an element in the list. When a sentinel node is used, all insertions and deletions will fall under the second case, and the implementations will not have as many special situations to consider.

4.3 Elements without Links

Now that we have explored some of the techniques needed to manage the nodes of a linked list, we can turn our attention to using a linked list as an alternative implementation approach for a collection. To do so, however, we need to carefully examine one other key aspect of linked lists. We must separate the details of the linked-list structure from the elements that the list stores.

Earlier in this chapter we discussed the idea of a `Person` class that contains, among its other data, a link to another `Person` object. The flaw in this approach is that the self-referential `Person` class must be designed so that it “knows” it may become a node in a linked list of `Person` objects. This assumption is impractical, and it violates our goal of separating the implementation details from the parts of the system that use the collection.

The solution to this problem is to define a separate node class that serves to link the elements together. A node class is fairly simple, containing only two important references: one to the next node in the linked list and another to the element that is being stored in the list. This approach is depicted in Figure 4.6.

KEY CONCEPT

Objects that are stored in a collection should not contain any implementation details of the underlying data structure.

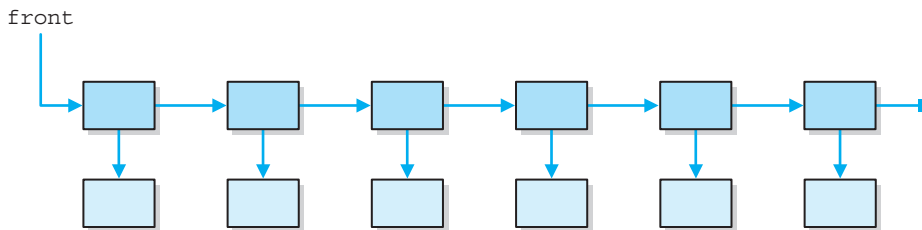


FIGURE 4.6 Using separate node objects to store and link elements

The linked list of nodes can still be managed using the techniques discussed in the previous section. The only additional aspect is that the actual elements stored in the list are accessed using separate references in the node objects.

Doubly Linked Lists

An alternative implementation for linked structures is the concept of a doubly linked list, as illustrated in Figure 4.7. In a doubly linked list, two references are maintained: one to point to the first node in the list and another to point to the last node in the list. Each node in the list stores both a reference to the next

element and a reference to the previous one. If we were to use sentinel nodes with a doubly linked list, we would place sentinel nodes on both ends of the list. We discuss doubly linked lists further in Chapter 6.

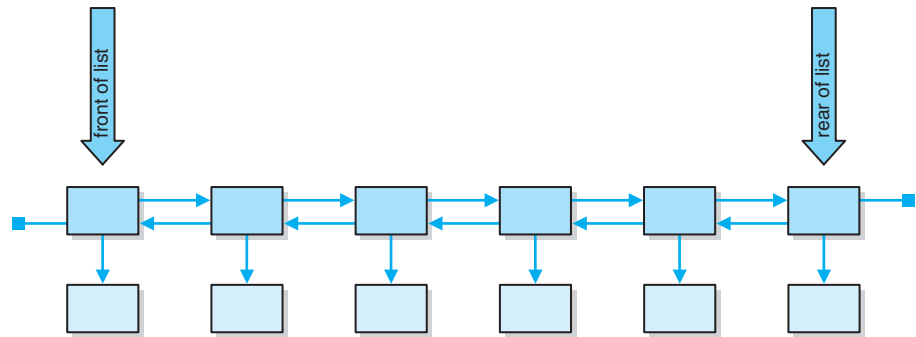


FIGURE 4.7 A doubly linked list

4.4 Stacks in the Java API

In Chapter 3, we used the `java.util.Stack` class from the Java Collections API to solve the postfix expression problem. The `Stack` class is an array-based implementation of a stack provided in the Java Collections API framework. This implementation provides the same basic operations we have been discussing:

- The `push` operation accepts a parameter item that is a reference to an object to be placed on the stack.
- The `pop` operation removes the object on top of the stack and returns a reference to it.
- The `peek` operation returns a reference to the object on top of the stack.

The `Stack` class is derived from the `Vector` class and uses its inherited capabilities to store the elements in the stack. Because this implementation is built on a vector, it exhibits the characteristics of both a vector and a stack and thus allows operations that violate the basic premise of a stack.

KEY CONCEPT

The `java.util.Stack` class is derived from `Vector`, which gives a stack inappropriate operations.

The `Deque` interface implemented by the `LinkedList` class provides a linked implementation of a stack and provides the same basic stack operations. A deque (pronounced like “deck”) is a double-ended queue, and we discuss that concept further in Chapter 5. Unfortunately, since a deque operates on both ends of the collection,

there are a variety of operations available that violate the premise of a stack. It is up to developers to limit themselves to the use of stack operations. Let's see how we use a Deque as a stack.

4.5 Using Stacks: Traversing a Maze

Another classic use of a stack data structure is to keep track of alternatives in maze traversal or other, similar algorithms that involve trial and error. Suppose that we build a grid as a two-dimensional array of integer values where each number represents either a path (1) or a wall (0) in a maze. We store our grid in a file where the first line of the file describes the number of rows and columns in the grid.

```

9 13
1 1 1 0 1 1 0 0 0 1 1 1 1
1 0 0 1 1 0 1 1 1 1 0 0 1
1 1 1 1 1 0 1 0 1 0 1 0 0
0 0 0 0 1 1 1 0 1 0 1 1 1
1 1 1 0 1 1 1 0 1 0 1 1 1
1 0 1 0 0 0 0 1 1 1 0 0 1
1 0 1 1 1 1 1 1 0 1 1 1 1
1 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1

```

The goal is to start in the top-left corner of this grid and traverse to the bottom-right corner of this grid, traversing only positions that are marked as a path. Valid moves will be those that are within the bounds of the grid and are to cells in the grid marked with a 1. We will mark our path as we go by changing the 1's to 2's, and we will push only valid moves onto the stack.

Starting in the top-left corner, we have two valid moves: down and right. We push these moves onto the stack, pop the top move off the stack (right), and then move to that location. This means that we moved right one position:

```

2 2 1 0 1 1 0 0 0 1 1 1 1
1 0 0 1 1 0 1 1 1 1 0 0 1
1 1 1 1 1 0 1 0 1 0 1 0 0
0 0 0 0 1 1 1 0 1 0 1 1 1
1 1 1 0 1 1 1 0 1 0 1 1 1
1 0 1 0 0 0 0 1 1 1 0 0 1
1 0 1 1 1 1 1 1 0 1 1 1 1
1 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1

```

We now have only one valid move. We push that move onto the stack, pop the top element off the stack (right), and then move to that location. Again we moved right one position:

```

2 2 2 0 1 1 0 0 0 1 1 1 1
1 0 0 1 1 0 1 1 1 1 0 0 1
1 1 1 1 1 0 1 0 1 0 1 0 0
0 0 0 0 1 1 1 0 1 0 1 1 1
1 1 1 0 1 1 1 0 1 0 1 1 1
1 0 1 0 0 0 0 1 1 1 0 0 1
1 0 1 1 1 1 1 1 0 1 1 1 1
1 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1

```

From this position, we do not have any valid moves. At this point, however, our stack is not empty. Keep in mind that we still have a valid move on the stack left from the first position. We pop the next (and currently last) element off the stack (down from the first position). We move to that position, push the valid move(s) from that position onto the stack, and continue processing.

```

2 2 2 0 1 1 0 0 0 1 1 1 1
2 0 0 1 1 0 1 1 1 1 0 0 1
1 1 1 1 1 0 1 0 1 0 1 0 0
0 0 0 0 1 1 1 0 1 0 1 1 1
1 1 1 0 1 1 1 0 1 0 1 1 1
1 0 1 0 0 0 0 1 1 1 0 0 1
1 0 1 1 1 1 1 1 0 1 1 1 1
1 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1

```

Using a stack in this way is actually simulating *recursion*, a process whereby a method calls itself either directly or indirectly. Recursion, which we discuss in detail in Chapter 8, uses the concept of a program stack. A *program stack* (or *runtime stack*) is used to keep track of methods that are invoked. Every time a method is called, an *activation record* that represents the invocation is created and pushed onto the program stack. Therefore, the elements on the stack represent the series of method invocations that occurred to reach a particular point in an executing program.

For example, when the `main` method of a program is called, an activation record for it is created and pushed onto the program stack. When `main` calls another method (say `m2`), an activation record for `m2` is created and pushed onto the stack. If `m2` calls method `m3`, then an activation record for `m3` is created and pushed onto the stack. When method `m3` terminates, its activation record is popped off the stack, and control returns to the calling method (`m2`), which is now on the top of the stack.

If an exception occurs during the execution of a Java program, the programmer can examine the *call stack trace* to see what method the problem occurred within and what method calls were made to arrive at that point.

An activation record contains various administrative data to help manage the execution of the program. It also contains a copy of the method's data (local variables and parameters) for that invocation of the method.

Because of the relationship between stacks and recursion, we can always rewrite a recursive program into a nonrecursive program that uses a stack. Instead of using recursion to keep track of the data, we can create our own stack to do so.

Listings 4.1, 4.2, and 4.3 illustrate the `Maze`, `MazeSolver`, and `MazeTester` classes that implement our stack-based solution to traversing a maze. We will revisit this same example in our discussion of recursion in Chapter 8.

Note that the constructor of the `Maze` class reads the initial maze data from a file specified by the user. This solution assumes that all issues regarding the file I/O will proceed without a problem, which, of course, is not a safe assumption. The file might not be present, the data might not be in the correct format, and so on. Several different exceptions could occur during the execution of the constructor, which doesn't catch or handle them. If any occur, the program will terminate. In a more robust program, these exceptions would be handled more elegantly.

This solution uses a class called `Position` to encapsulate the coordinates of a position within the maze. The `traverse` method loops, popping the top position off the stack, marking it as tried, and then testing to see whether we are done. If we are not done, then all of the valid moves from this position are pushed onto the stack, and the loop continues. A private method called `push_new_pos` has been created to handle the task of putting the valid moves from the current position onto the stack:

```
private StackADT<Position> push_new_pos(int x, int y,
    StackADT<Position> stack)
{
    Position npos = new Position();
    npos.setx(x);
    npos.sety(y);
    if (valid(npos.getx(), npos.gety()))
        stack.push(npos);
    return stack;
}
```

KEY CONCEPT

Recursive processing can be simulated using a stack to keep track of the appropriate data.



VideoNote

Using a stack to solve a maze.

LISTING 4.1

```

import java.util.*;
import java.io.*;

/**
 * Maze represents a maze of characters. The goal is to get from the
 * top left corner to the bottom right, following a path of 1's. Arbitrary
 * constants are used to represent locations in the maze that have been TRIED
 * and that are part of the solution PATH.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class Maze
{
    private static final int TRIED = 2;
    private static final int PATH = 3;
    private int numberOfRows, numberOfColumns;
    private int[][] grid;

    /**
     * Constructor for the Maze class. Loads a maze from the given file.
     * Throws a FileNotFoundException if the given file is not found.
     *
     * @param filename the name of the file to load
     * @throws FileNotFoundException if the given file is not found
     */
    public Maze(String filename) throws FileNotFoundException
    {
        Scanner scan = new Scanner(new File(filename));
        numberOfRows = scan.nextInt();
        numberOfColumns = scan.nextInt();

        grid = new int[numberOfRows][numberOfColumns];
        for (int i = 0; i < numberOfRows; i++)
            for (int j = 0; j < numberOfColumns; j++)
                grid[i][j] = scan.nextInt();
    }

    /**
     * Marks the specified position in the maze as TRIED
     *
     * @param row the index of the row to try
     * @param col the index of the column to try
     */
}

```

LISTING 4.1*continued*

```
public void tryPosition(int row, int col)
{
    grid[row][col] = TRIED;
}

/**
 * Return the number of rows in this maze
 *
 * @return the number of rows in this maze
 */
public int getRows()
{
    return grid.length;
}

/**
 * Return the number of columns in this maze
 *
 * @return the number of columns in this maze
 */
public int getColumns()
{
    return grid[0].length;
}

/**
 * Marks a given position in the maze as part of the PATH
 *
 * @param row the index of the row to mark as part of the PATH
 * @param col the index of the column to mark as part of the PATH
 */
public void markPath(int row, int col)
{
    grid[row][col] = PATH;
}

/**
 * Determines if a specific location is valid. A valid location
 * is one that is on the grid, is not blocked, and has not been TRIED.
 *
 * @param row the row to be checked
 * @param column the column to be checked
 * @return true if the location is valid
 */
```

LISTING 4.1 *continued*

```

public boolean validPosition(int row, int column)
{
    boolean result = false;

    // check if cell is in the bounds of the matrix
    if (row >= 0 && row < grid.length &&
        column >= 0 && column < grid[row].length)

        // check if cell is not blocked and not previously tried
        if (grid[row][column] == 1)
            result = true;
    return result;
}

/**
 * Returns the maze as a string.
 *
 * @return a string representation of the maze
 */
public String toString()
{
    String result = "\n";
    for (int row=0; row < grid.length; row++)
    {
        for (int column=0; column < grid[row].length; column++)
            result += grid[row][column] + " ";
        result += "\n";
    }
    return result;
}
}

```

LISTING 4.2

```
import java.util.*;

/**
 * MazeSolver attempts to recursively traverse a Maze. The goal is to get from the
 * given starting position to the bottom right, following a path of 1's. Arbitrary
 * constants are used to represent locations in the maze that have been TRIED
 * and that are part of the solution PATH.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class MazeSolver
{
    private Maze maze;

    /**
     * Constructor for the MazeSolver class.
     */
    public MazeSolver(Maze maze)
    {
        this.maze = maze;
    }

    /**
     * Attempts to recursively traverse the maze. Inserts special
     * characters indicating locations that have been TRIED and that
     * eventually become part of the solution PATH.
     *
     * @param row row index of current location
     * @param column column index of current location
     * @return true if the maze has been solved
     */
    public boolean traverse()
    {
        boolean done = false;
        int row, column;
        Position pos = new Position();
        Deque<Position> stack = new LinkedList<Position>();
        stack.push(pos);
    }
}
```

LISTING 4.2 *continued*

```

while (!(done) && !stack.isEmpty())
{
    pos = stack.pop();
    maze.tryPosition(pos.getx(),pos.gety()); // this cell has been tried
    if (pos.getx() == maze.getRows()-1 && pos.gety() == maze.getColumns()-1)
        done = true; // the maze is solved
    else
    {
        push_new_pos(pos.getx() - 1,pos.gety(), stack);
        push_new_pos(pos.getx() + 1,pos.gety(), stack);
        push_new_pos(pos.getx(),pos.gety() - 1, stack);
        push_new_pos(pos.getx(),pos.gety() + 1, stack);
    }
}

return done;
}

/**
 * Push a new attempted move onto the stack
 * @param x represents x coordinate
 * @param y represents y coordinate
 * @param stack the working stack of moves within the grid
 * @return stack of moves within the grid
 */
private void push_new_pos(int x, int y,
                          Deque<Position> stack)
{
    Position npos = new Position();
    npos.setx(x);
    npos.sety(y);
    if (maze.validPosition(x,y))
        stack.push(npos);
}
}

```

LISTING 4.3

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

/**
 * MazeTester uses recursion to determine if a maze can be traversed.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class MazeTester
{
    /**
     * Creates a new maze, prints its original form, attempts to
     * solve it, and prints out its final form.
     */
    public static void main(String[] args) throws FileNotFoundException
    {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter the name of the file containing the maze: ");
        String filename = scan.nextLine();

        Maze labyrinth = new Maze(filename);

        System.out.println(labyrinth);

        MazeSolver solver = new MazeSolver(labyrinth);
        if (solver.traverse())
            System.out.println("The maze was successfully traversed!");
        else
            System.out.println("There is no possible path.");
        System.out.println(labyrinth);
    }
}
```

The UML description for the maze problem is left as an exercise.

4.6 Implementing a Stack: With Links

Let's use a linked list to implement a stack collection, which was defined in Chapter 3. Note that we are not changing the way in which a stack works. Its conceptual nature remains the same, as does the set of operations defined for it. We are merely changing the underlying data structure used to implement it.

The purpose of the stack, and the solutions it helps us to create, also remain the same. The postfix expression evaluation example from Chapter 3 used the `java.util.Stack<T>` class, but any valid implementation of a stack could be used instead. Once we create the `LinkedStack<T>` class to define an alternative implementation, we could substitute it into the postfix expression solution without having to change anything but the class name. That is the beauty of abstraction.

KEY CONCEPT

Any implementation of a collection can be used to solve a problem, as long as it validly implements the appropriate operations.

In the following discussion, we show and discuss the methods that are important to understanding the linked-list implementation of a stack. Some of the stack operations are left as programming projects.

The `LinkedStack` Class

The `LinkedStack<T>` class implements the `StackADT<T>` interface, just as the `ArrayStack<T>` class from Chapter 3 does. Both provide the operations defined for a stack collection.

Because we are using a linked-list approach, there is no array in which we store the elements of the collection. Instead, we need only a single reference to the first node in the list. We will also maintain a count of the number of elements in the list. The header and class-level data of the `LinkedStack<T>` class are therefore:

```
package jsjf;
import jsjf.exceptions.*;
import java.util.Iterator;

/**
 * Represents a linked implementation of a stack.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class LinkedStack<T> implements StackADT<T>
{
    private int count;
    private LinearNode<T> top;
```

The `LinearNode<T>` class serves as the node class, containing a reference to the next `LinearNode<T>` in the list and a reference to the element stored in that node. Each node stores a generic type that is determined when the node is instantiated. In our `LinkedStack<T>` implementation, we simply use the same type for the node as used to define the stack. The `LinearNode<T>` class also contains methods to set and get the element values. The `LinearNode<T>` class is shown in Listing 4.4.

Note that the `LinearNode<T>` class is not tied to the implementation of a stack collection. It can be used in any linear linked-list implementation of a collection. We will use it for other collections as needed.

Using the `LinearNode<T>` class and maintaining a count of elements in the collection creates the implementation strategy depicted in Figure 4.8.

The constructor of the `LinkedStack<T>` class sets the count of elements to zero and sets the front of the list, represented by the variable `top`, to `null`. Note that because a linked-list implementation does not have to worry about capacity limitations, there is no need to create a second constructor as we did in the `ArrayStack<T>` class of Chapter 3.

KEY CONCEPT

A linked implementation of a stack adds and removes elements from one end of the linked list.

```
/**
 * Creates an empty stack.
 */
public LinkedStack()
{
    count = 0;
    top = null;
}
```

LISTING 4.4

```
package jsjf;

/**
 * Represents a node in a linked list.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
```


LISTING 4.4 *continued*

```
public class LinearNode<T>
{
    private LinearNode<T> next;
    private T element;

    /**
     * Creates an empty node.
     */
    public LinearNode()
    {
        next = null;
        element = null;
    }

    /**
     * Creates a node storing the specified element.
     * @param elem element to be stored
     */
    public LinearNode(T elem)
    {
        next = null;
        element = elem;
    }

    /**
     * Returns the node that follows this one.
     * @return reference to next node
     */
    public LinearNode<T> getNext()
    {
        return next;
    }

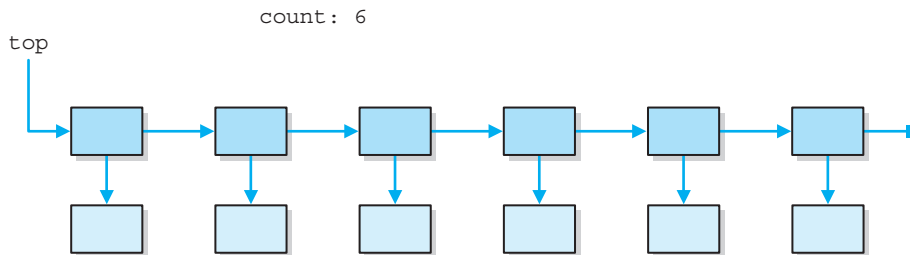
    /**
     * Sets the node that follows this one.
     * @param node node to follow this one
     */
    public void setNext(LinearNode<T> node)
    {
        next = node;
    }

    /**
     * Returns the element stored in this node.
     * @return element stored at the node
     */
}
```

LISTING 4.4 *continued*

```
public T getElement()
{
    return element;
}

/**
 * Sets the element stored in this node.
 * @param elem element to be stored at this node
 */
public void setElement(T elem)
{
    element = elem;
}
}
```

**FIGURE 4.8** A linked implementation of a stack collection

Because the nature of a stack is to allow elements to be added to, or removed from, only one end, we will only need to operate on one end of our linked list. We could choose to push the first element into the first position in the linked list, the second element into the second position, and so on. This would mean that the top of the stack would always be at the tail end of the list. However, if we consider the efficiency of this strategy, we realize that it would mean we would have to traverse the entire list on every push and every pop operation. Instead, we can choose to operate on the front of the list, making the front of the list the top of the stack. In this way, we do not have to traverse the list for either the push or the pop operation. Figure 4.9 illustrates this configuration for a stack containing four elements, A, B, C, and D, that have been pushed onto the stack in that order.

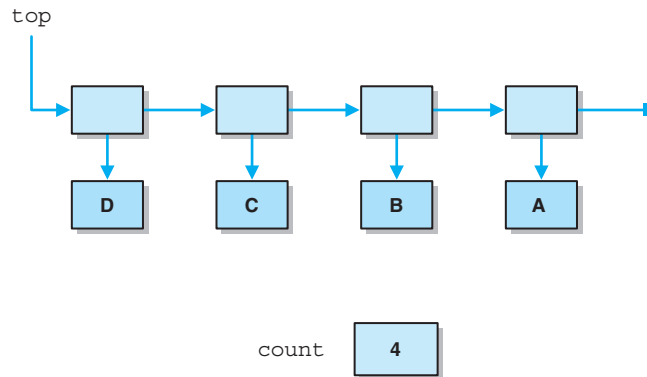


FIGURE 4.9 A linked implementation of a stack

Let's explore the implementation of the stack operations for the `LinkedStack` class.

The push Operation

Every time a new element is pushed onto the stack, a new `LinearNode` object must be created to store it in the linked list. To position the newly created node at the top of the stack, we must set its `next` reference to the current top of the stack and reset the `top` reference to point to the new node. We must also increment the `count` variable.

Implementing these steps results in the following code:

```
/**
 * Adds the specified element to the top of this stack.
 * @param element element to be pushed on stack
 */
public void push(T element)
{
    LinearNode<T> temp = new LinearNode<T>(element);

    temp.setNext(top);
    top = temp;
    count++;
}
```

Figure 4.10 shows the result of pushing the element E onto the stack depicted in Figure 4.9.

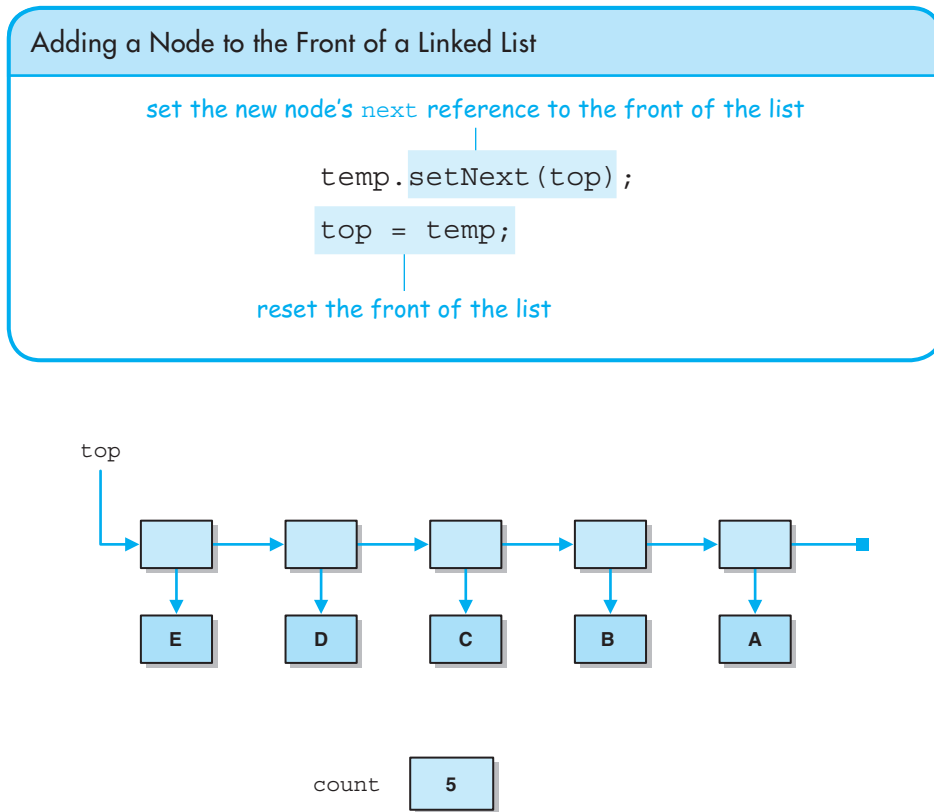


FIGURE 4.10 The stack after pushing element E

The `push` operation for the linked implementation of a stack consists of the following steps:

- Create a new node containing a reference to the object to be placed on the stack.
- Set the `next` reference of the new node to point to the current top of the stack (which will be null if the stack is empty).
- Set the `top` reference to point to the new node.
- Increment the count of elements in the stack.

All of these steps have time complexity $O(1)$ because they require only one processing step regardless of the number of elements already in the stack. Each of these steps would have to be accomplished once for each of the elements to be pushed. Thus, using this method, the `push` operation would be $O(1)$.

The `pop` Operation

The `pop` operation is implemented by returning a reference to the element currently stored at the top of the stack and adjusting the top reference to the new top of the stack. Before attempting to return any element, however, we must first ensure that there is at least one element to return. This operation can be implemented as follows:

```
/**
 * Removes the element at the top of this stack and returns a
 * reference to it.
 * @return element from top of stack
 * @throws EmptyCollectionException if the stack is empty
 */
public T pop() throws EmptyCollectionException
{
    if (isEmpty())
        throw new EmptyCollectionException("stack");

    T result = top.getElement();
    top = top.getNext();
    count--;

    return result;
}
```

If the stack is empty, as determined by the `isEmpty` method, an `EmptyCollectionException` is thrown. If there is at least one element to pop, it is stored in a temporary variable so that it can be returned. Then the reference to the top of the stack is set to the next element in the list, which is now the new top of the stack. The count of elements is decremented as well.

Figure 4.11 illustrates the result of a `pop` operation on the stack from Figure 4.10. Notice that this figure is identical to our original configuration in Figure 4.9. This illustrates the fact that the `pop` operation is the inverse of the `push` operation.

The `pop` operation for the linked implementation consists of the following steps:

- Make sure the stack is not empty.
- Set a temporary reference equal to the element on top of the stack.
- Set the `top` reference equal to the `next` reference of the node at the top of the stack.
- Decrement the count of elements in the stack.
- Return the element pointed to by the temporary reference.

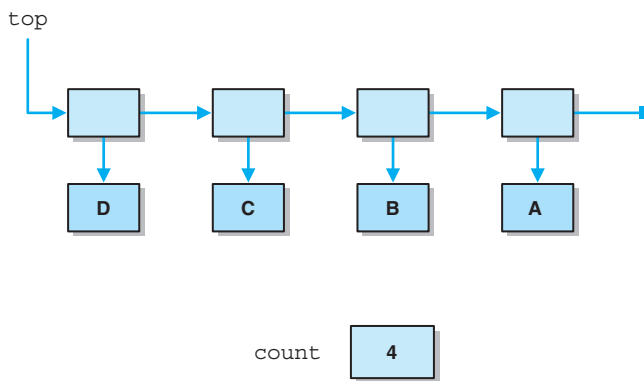


FIGURE 4.11 The stack after a `pop` operation

As with our previous examples, each of these operations consists of a single comparison or a simple assignment and is therefore $O(1)$. Thus, the `pop` operation for the linked implementation is $O(1)$.

Other Operations

Using a linked implementation, the `peek` operation is implemented by returning a reference to the element pointed to by the node pointed to by the `top` pointer. The `isEmpty` operation returns `true` if the count of elements is 0, and `false` otherwise. The `size` operation returns the count of elements in the stack. The `toString` operation can be implemented by simply traversing the linked list. These operations are left as programming projects.

Summary of Key Concepts

- Object reference variables can be used to create linked structures.
- A linked list is composed of objects that each point to the next object in the list.
- A linked list dynamically grows as needed and essentially has no capacity limitations.
- The order in which references are changed is crucial to maintaining a linked list.
- Dealing with the first node in a linked list often requires special handling.
- Objects that are stored in a collection should not contain any implementation details of the underlying data structure.
- The `java.util.Stack` class is derived from `Vector`, which gives a stack inappropriate operations.
- Any implementation of a collection can be used to solve a problem, as long as it validly implements the appropriate operations.
- A linked implementation of a stack adds elements to, and removes elements from, one end of the linked list.
- Recursive processing can be simulated using a stack to keep track of the appropriate data.

Summary of Terms

activation record An object that represents a method invocation.

doubly linked list A linked list in which each node has references to both the next node and the previous node in the list.

linked list A linked structure in which one object refers to the next, creating a linear ordering.

linked structure A data structure that uses object reference variables to create links between objects.

node A class that represents a single element in a linked structure.

program stack A stack of activation records used to keep track of method invocations during program execution.

sentinel node A node at the front or end of a linked list that serves as a marker and does not represent an element in the list.

Self-Review Questions

- SR 4.1 What do you understand by a linked structure?
- SR 4.2 What do you understand by a self-referential object?
- SR 4.3 How are the beginning and termination of a linked list indicated?
- SR 4.4 Describe in brief how the elements in a linked list are accessed.
- SR 4.5 What are the steps to be taken to insert a node at the beginning of a linked list?
- SR 4.6 What is the difference between deleting the first node in a linked list and deleting any of the other nodes in the middle of the list?
- SR 4.7 Describe the following data structure: *doubly linked list*.
- SR 4.8 What impact would the use of sentinel nodes or dummy nodes have on a doubly linked list implementation?
- SR 4.9 What are the advantages of using a linked implementation as opposed to an array implementation?
- SR 4.10 What are the advantages of using an array implementation as opposed to a linked implementation?
- SR 4.11 What are the advantages of using the `java.util.Stack` implementation of a stack?
- SR 4.12 What is the potential problem with the `java.util.Stack` implementation?

Exercises

- EX 4.1 Write Java statements to define a `Student` class that would have three private members: name, code, and a reference to another `Student` object.
- EX 4.2 What should be the basic operations required for a linked list of objects?
- EX 4.3 Write minimal Java statements to define a `StudentList` class that would store a list of students, as defined in Exercise 4.1.
- EX 4.4 Write Java statements to access the fifth student in the list described in Exercise 4.3.
- EX 4.5 Write an algorithm for the `traverse` method that will traverse a list. What is the time complexity of this algorithm?

- EX 4.6 Consider the list created in Exercise 4.3. Write Java statements for defining a method that would be traversing a list and displaying the content of the list.
- EX 4.7 Write Java statements for a method `search` that would be searching for a given string in the list created in Exercise 4.3.
- EX 4.8 What is the design flaw in the `Student` class described in Exercise 4.1? How should you rectify this flaw?

Programming Projects

- PP 4.1 Complete the implementation of the `LinkedList<T>` class by providing definitions for the `peek`, `size`, `isEmpty`, and `toString` methods.
- PP 4.2 Modify the `postfix` program from Chapter 3 so that it uses the `LinkedList<T>` class instead of the `ArrayStack<T>` class.
- PP 4.3 Create a new version of the `LinkedList<T>` class that makes use of a dummy record at the head of the list.
- PP 4.4 Create a simple graphical application that will enable a user to perform `push`, `pop`, and `peek` operations on a stack, and display the resulting stack (using `toString`) in a text area.
- PP 4.5 Design and implement an application that reads a sentence from the user and prints the sentence with the characters of each word backward. Use a stack to reverse the characters of each word.
- PP 4.6 Complete the solution to the iterative maze solver so that your solution marks the successful path.
- PP 4.7 The linked implementation in this chapter uses a `count` variable to keep track of the number of elements in the stack. Rewrite the linked implementation without a `count` variable.
- PP 4.8 There is a data structure called a drop-out stack that behaves like a stack in every respect except that if the stack size is n , then when the $n+1$ element is pushed, the first element is lost. Implement a drop-out stack using links.
- PP 4.9 Modify the maze problem in this chapter so that it can start from a user-defined starting position (other than 0, 0) and search for a user-defined ending point (other than row-1, column-1).

Answers to Self-Review Questions

- SRA 4.1 A linked structure is a data structure that uses object reference variables to create links between objects. Linked structures are the primary alternative to an array-based implementation of a collection.
- SRA 4.2 When the instance of a class defines as instance data a reference to another instance of the same class, it is called a self-referential object. Using only this one class, we can create a linked structure. One object instance contains a link to a second object instance. This second object also contains a link to a third object instance and so on.
- SRA 4.3 A separate reference variable is used to indicate the first node in the list. If this node is deleted, or if a new element is added in front of it, the front reference is carefully maintained. The list is terminated in a node whose next reference is null.
- SRA 4.4 When using linked lists, we maintain a pointer to the first element in the list. To access other elements, we access the first one and then follow the next pointer from that one to the next one, and so on. Special care is taken for dealing with the first node in the list so that the reference to the entire list is maintained appropriately.
- SRA 4.5 Adding a node at the beginning of the list requires resetting the reference to the entire list. So, two steps are taken to insert at the front: first, the next reference of the added node is set to point to the current first node in the list and second, the reference to the front of the list is reset to point to the newly added node.
- SRA 4.6 To delete the first node in a linked list, the reference to the front of the list need to be reset so that it points to the current second node in the list. If any other node is needed to be deleted, a separate reference to it should be set up before resetting the front reference.
- SRA 4.7 In a doubly linked list, two references are maintained: one to point to the first node in the list and another to point to the last node in the list. Each node in the list stores both a reference to the next element and a reference to the previous one. If we were to use sentinel nodes with a doubly linked list, we would place sentinel nodes on both ends of the list.

- SRA 4.8 It would take two dummy records in a doubly linked list, one at the front and one at the rear, to eliminate the special cases when dealing with the first and last nodes.
- SRA 4.9 A linked implementation allocates space only as it is needed and has a theoretical limit on the size of the hardware.
- SRA 4.10 An array implementation uses less space per object since it only has to store the object and not an extra pointer. However, the array implementation will allocate much more space than it needs initially.
- SRA 4.11 Because the `java.util.Stack` implementation is an extension of the `Vector` class, it can keep track of the positions of elements in the stack using an index and thus does not require each node to store an additional pointer. This implementation also allocates space only as it is needed, like the linked implementation.
- SRA 4.12 The `java.util.Stack` implementation is an extension of the `Vector` class and thus inherits a large number of operations that violate the basic assumptions of a stack.



Queues

5

A queue is another collection with which we are inherently familiar. A queue is a waiting line, such as a line of customers waiting in a bank for their opportunity to talk to a teller. In fact, in many countries the word *queue* is used habitually in this way. In such countries, a person might say, “join the queue” rather than “get in line.” Other examples of queues include a checkout line at the grocery store and cars waiting at a stoplight. In any queue, an item enters on one end and leaves from the other. Queues have a variety of uses in computer algorithms.

CHAPTER OBJECTIVES

- Examine queue processing.
- Demonstrate how a queue can be used to solve problems.
- Define a queue abstract data type.
- Examine various queue implementations.
- Compare queue implementations.

5.1 A Conceptual Queue

A *queue* is a linear collection whose elements are added on one end and removed from the other. Therefore, we say that queue elements are processed in a *first in, first out* (FIFO) manner. Elements are removed from a queue in the same order in which they are placed on the queue.

This is consistent with the general concept of a waiting line. When a customer arrives at a bank, he or she begins waiting at the end of the line. When a teller becomes available, the customer at the beginning of the line leaves the line to receive service. Eventually every customer who started out at the end of the line moves to the front of the line and exits. For any given set of people, the first person to get in line is the first person to leave it.

KEY CONCEPT

Queue elements are processed in a FIFO manner—the first element in is the first element out.

The processing of a queue is pictured in Figure 5.1. Usually a queue is depicted horizontally. One end is established as the *front* of the queue and the other as the *rear* of the queue. Elements go onto the rear of the queue and come off the front. Sometimes the front of the queue is called the *head* and the rear of the queue is called the *tail*.

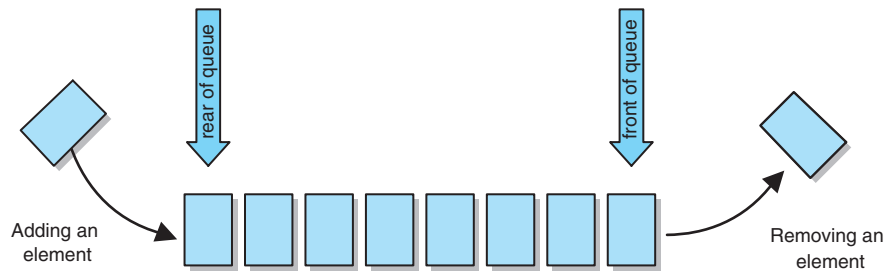


FIGURE 5.1 A conceptual view of a queue

Compare and contrast the processing of a queue to the LIFO (last in, first out) processing of a stack, which was discussed in Chapters 3 and 4. In a stack, the processing occurs at only one end of the collection. In a queue, processing occurs at both ends.

The operations defined for a queue ADT are listed in Figure 5.2. The term *enqueue* is used to refer to the process of adding a new element to the end of a queue. Likewise, *dequeue* is the process of removing the element at the front of a queue. The *first* operation enables the user to examine the element at the front of the queue without removing it from the collection.

Operation	Description
<code>enqueue</code>	Adds an element to the rear of the queue.
<code>dequeue</code>	Removes an element from the front of the queue.
<code>first</code>	Examines the element at the front of the queue.
<code>isEmpty</code>	Determines if the queue is empty.
<code>size</code>	Determines the number of elements on the queue.
<code>toString</code>	Returns a string representation of the queue.

FIGURE 5.2 The operations on a queue

Remember that naming conventions are not universal for collection operations. Sometimes `enqueue` is simply called `add`, `insert`, or `offer`. The `dequeue` operation is sometimes called `remove`, `poll`, or `serve`. The `first` operation is sometimes called `front` or `peek`.

Note that there is a general similarity between the operations on a queue and those on a stack. The `enqueue`, `dequeue`, and `first` operations correspond to the stack operations `push`, `pop`, and `peek`. As is true of a stack, there are no operations that allow the user to “reach into” the middle of a queue and reorganize or remove elements. If that type of processing is required, perhaps the appropriate collection to use is a list of some kind, such as those discussed in the next chapter.

5.2 Queues in the Java API

Unfortunately, the Java Collections API is not consistent in its implementations of collections. There are a couple of important differences between the way the stack and queue collections are implemented:

- The Java Collections API provides the `java.util.Stack` class that implements a stack collection. Instead of a queue class, a `Queue` interface is provided and is implemented by several classes, including the `LinkedList` class.
- The `java.util.Stack` class provides the traditional `push`, `pop`, and `peek` operations. The `Queue` interface does not implement the traditional `enqueue`, `dequeue`, and `first` operations. Instead, the `Queue` interface defines two alternatives for adding elements to, and removing elements from, a queue. These alternatives behave differently in terms of how the exceptional cases

are handled. One set provides a boolean return value, whereas the other throws an exception.

The `Queue` interface defines an `element` method that is the equivalent of our conceptual `first`, `front`, or `peek`. This method retrieves the element at the head of the queue but does not remove it.

The `Queue` interface provides two methods for adding an element to the queue: `add` and `offer`. The `add` operation ensures that the queue contains the given element. This operation will throw an exception if the given element cannot be added to the queue. The `offer` operation inserts the given element into this queue, returning `true` if the insertion is successful and `false` otherwise.

The `Queue` interface also provides two methods of removing an element from the queue: `poll` and `remove`. Just like the difference between the `add` and `offer` methods, the difference between `poll` and `remove` is in how the exceptional case is handled. In this instance, the exceptional case occurs when an attempt is made to remove an element from an empty queue. The `poll` method will return `null` if the queue is empty, whereas the `remove` method will throw an exception.

Queues have a wide variety of applications within computing. Whereas the principal purpose of a stack is to reverse order, the principal purpose of a queue is to preserve order. Before exploring various ways to implement a queue, let's examine some ways in which a queue can be used to solve problems.

5.3 Using Queues: Code Keys

A *Caesar cipher* is a simple approach to encoding messages by shifting each letter in a message along the alphabet by a constant amount k . For example, if k equals 3, then in an encoded message, each letter is shifted three characters forward: `a` is replaced with `d`, `b` with `e`, `c` with `f`, and so on. The end of the alphabet wraps back around to the beginning. Thus `w` is replaced with `z`, `x` with `a`, `y` with `b`, and `z` with `c`.

To decode the message, each letter is shifted the same number of characters backward. Therefore, if k equals 3, then the encoded message

```
vlpsolflwb iroorzv frpsohalwb
```

would be decoded into

```
simplicity follows complexity
```

Julius Caesar actually used this type of cipher in some of his secret government correspondence (hence the name). Unfortunately, the Caesar cipher is fairly easy

to break. There are only 26 possibilities for shifting the characters, and the code can be broken by trying various key values until one works.

This encoding technique can be improved by using a *repeating key*. Instead of shifting each character by a constant amount, we can shift each character by a different amount using a list of key values. If the message is longer than the list of key values, we just start using the key over again from the beginning. For example, if the key values are

3 1 7 4 2 5

then the first character is shifted by three, the second character by one, the third character by seven, and so on. After shifting the sixth character by five, we start using the key over again. The seventh character is shifted by three, the eighth by one, and so on.

Figure 5.3 shows the message “knowledge is power” encoded using this repeating key. Note that this encryption approach encodes the same letter into different characters, depending on where it occurs in the message (and thus on which key value is used to encode it). Conversely, the same character in the encoded message is decoded into different characters.

Encoded Message:	n	o	v	a	n	j	g	h	l		m	u		u	r	x	l	v
Key:	3	1	7	4	2	5	3	1	7		4	2		5	3	1	7	4
Decoded Message:	k	n	o	w	l	e	d	g	e		i	s		p	o	w	e	r

FIGURE 5.3 An encoded message using a repeating key

The program in Listing 5.1 uses a repeating key to encode and decode a message. The key of integer values is stored in a queue. After a key value is used, it is put back on the end of the queue so that the key continually repeats as needed for long messages. The key in this example uses both positive and negative values. Figure 5.4 illustrates the UML description of the `Codes` class. As we saw in Figure 3.7 in Chapter 3, the UML diagram illustrates the binding of the generic type `T` to an `Integer`. Unlike the earlier example, in this case we have two different bindings illustrated: one for the `LinkedList` class and one for the `Queue` interface.

KEY CONCEPT

A queue is a convenient collection for storing a repeating code key.

LISTING 5.1

```

import java.util.*;

/**
 * Codes demonstrates the use of queues to encrypt and decrypt messages.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class Codes
{
    /**
     * Encode and decode a message using a key of values stored in
     * a queue.
     */
    public static void main(String[] args)
    {
        int[] key = {5, 12, -3, 8, -9, 4, 10};
        Integer keyValue;
        String encoded = "", decoded = "";
        String message = "All programmers are playwrights and all " +
            "computers are lousy actors.";
        Queue<Integer> encodingQueue = new LinkedList<Integer>();
        Queue<Integer> decodingQueue = new LinkedList<Integer>();

        // load key queues

        for (int scan = 0; scan < key.length; scan++)
        {
            encodingQueue.add(key[scan]);
            decodingQueue.add(key[scan]);
        }

        // encode message

        for (int scan = 0; scan < message.length(); scan++)
        {
            keyValue = encodingQueue.remove();
            encoded += (char) (message.charAt(scan) + keyValue);
            encodingQueue.add(keyValue);
        }

        System.out.println ("Encoded Message:\n" + encoded + "\n");

        // decode message
    }
}

```

LISTING 5.1

continued

```

for (int scan = 0; scan < encoded.length(); scan++)
{
    keyValue = decodingQueue.remove();
    decoded += (char) (encoded.charAt(scan) - keyValue);
    decodingQueue.add(keyValue);
}

System.out.println ("Decoded Message:\n" + decoded);
}
}

```

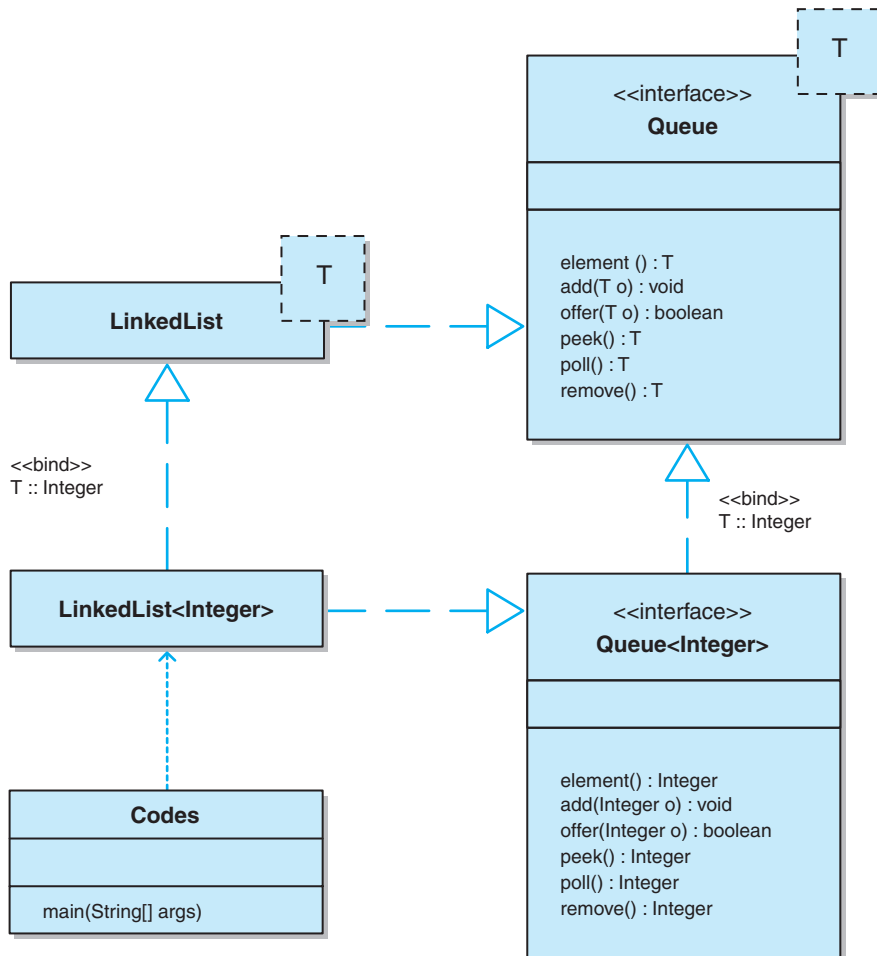


FIGURE 5.4 UML description of the Codes program

This program actually uses two copies of the key stored in two separate queues. The idea is that the person encoding the message has one copy of the key, and the person decoding the message has another. Two copies of the key are helpful in this program, because the decoding process needs to match up the first character of the message with the first value in the key.

Also, note that this program doesn't bother to wrap around the end of the alphabet. It encodes any character in the Unicode character set by shifting it to some other position in the character set. Therefore, we can encode any character, including uppercase letters, lowercase letters, and punctuation. Even spaces get encoded.

Using a queue to store the key makes it easy to repeat the key by putting each key value back onto the queue as soon as it is used. The nature of a queue keeps the key values in the proper order, and we don't have to worry about reaching the end of the key and starting over.

5.4 Using Queues: Ticket Counter Simulation

Let's look at another example using queues. Consider the situation in which you are waiting in line to purchase tickets at a movie theatre. In general, the more cashiers there are, the faster the line moves. The theatre manager wants to keep his customers happy, but he doesn't want to employ any more cashiers than necessary. Suppose the manager wants to keep the total time needed by a customer to less than seven minutes. Being able to simulate the effect of adding more cashiers during peak business hours enables the manager to plan more effectively. And, as we've discussed, a queue is the perfect collection for representing a waiting line.

Our simulated ticket counter will use the following assumptions:

- There is only one line and it is first come, first served (a queue).
- Customers arrive on average every 15 seconds.
- If there is a cashier available, processing begins immediately upon arrival.
- Processing a customer request takes on average two minutes (120 seconds) from the time the customer reaches a cashier.

First we can create a `Customer` class, as shown in Listing 5.2. A `Customer` object keeps track of the time the customer arrives and the time the customer departs after purchasing a ticket. The total time spent by the customer is therefore the departure time minus the arrival time. To keep things simple, our simulation will measure time in elapsed seconds, so a time value can be stored as a single integer. Our simulation will begin at time 0.

KEY CONCEPT

Simulations are often implemented using queues to represent waiting lines.

LISTING 5.2

```
/**
 * Customer represents a waiting customer.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class Customer
{
    private int arrivalTime, departureTime;

    /**
     * Creates a new customer with the specified arrival time.
     * @param arrives the arrival time
     */
    public Customer(int arrives)
    {
        arrivalTime = arrives;
        departureTime = 0;
    }

    /**
     * Returns the arrival time of this customer.
     * @return the arrival time
     */
    public int getArrivalTime()
    {
        return arrivalTime;
    }

    /**
     * Sets the departure time for this customer.
     * @param departs the departure time
     */
    public void setDepartureTime(int departs)
    {
        departureTime = departs;
    }

    /**
     * Returns the departure time of this customer.
     * @return the departure time
     */
}
```

LISTING 5.2 *continued*

```
public int getDepartureTime()
{
    return departureTime;
}

/**
 * Computes and returns the total time spent by this customer.
 * @return the total customer time
 */
public int totalTime()
{
    return departureTime - arrivalTime;
}
}
```

Our simulation will create a queue of customers and then see how long it takes to process those customers if there is only one cashier. Then we will process the same queue of customers with two cashiers. Then we will do it again with three cashiers. We continue this process for up to ten cashiers. At the end we compare the average times that it takes to process a customer.

Because of our assumption that customers arrive every 15 seconds (on average), we can preload a queue with customers. We will process 100 customers in this simulation.

The program shown in Listing 5.3 conducts our simulation. The outer loop determines how many cashiers are used in each pass of the simulation. For each pass, the customers are taken from the queue in turn and processed by a cashier. The total elapsed time is tracked, and at the end of each pass the average time is computed. Figure 5.5 shows the UML description of the `TicketCounter` and `Customer` classes.

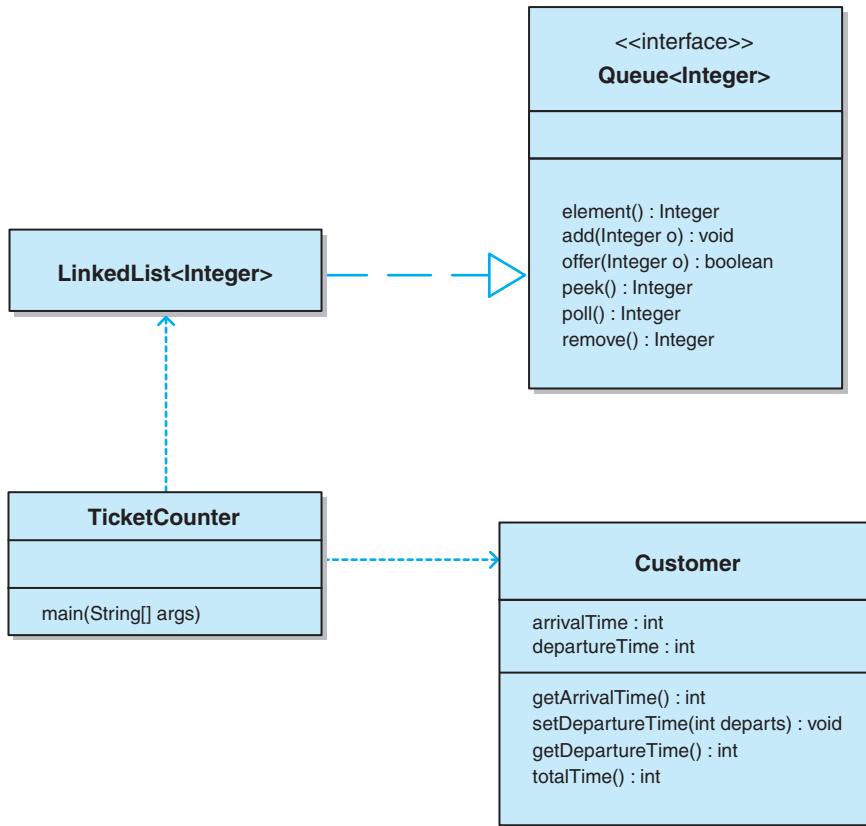


FIGURE 5.5 UML description of the TicketCounter program

LISTING 5.3

```

import java.util.*;

/**
 * TicketCounter demonstrates the use of a queue for simulating a line of customers.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class TicketCounter

```

LISTING 5.3 *continued*

```

{
    private final static int PROCESS = 120;
    private final static int MAX_CASHIERS = 10;
    private final static int NUM_CUSTOMERS = 100;

    public static void main(String[] args)
    {
        Customer customer;
        Queue<Customer> customerQueue = new LinkedList<Customer>();
        int[] cashierTime = new int[MAX_CASHIERS];
        int totalTime, averageTime, departs, start;

        // run the simulation for various number of cashiers

        for (int cashiers = 0; cashiers < MAX_CASHIERS; cashiers++)
        {

            // set each cashiers time to zero initially

            for (int count = 0; count < cashiers; count++)
                cashierTime[count] = 0;

            // load customer queue

            for (int count = 1; count <= NUM_CUSTOMERS; count++)
                customerQueue.add(new Customer(count * 15));
            totalTime = 0;

            // process all customers in the queue

            while (!(customerQueue.isEmpty()))
            {
                for (int count = 0; count <= cashiers; count++)
                {
                    if (!(customerQueue.isEmpty()))
                    {
                        customer = customerQueue.remove();
                        if (customer.getArrivalTime() > cashierTime[count])
                            start = customer.getArrivalTime();
                        else
                            start = cashierTime[count];
                        departs = start + PROCESS;
                        customer.setDepartureTime(departs);
                        cashierTime[count] = departs;
                        totalTime += customer.totalTime();
                    }
                }
            }
        }
    }
}

```

LISTING 5.3*continued*

```

    }
}

// output results for this simulation

averageTime = totalTime / NUM_CUSTOMERS;
System.out.println("Number of cashiers: " + (cashiers + 1));
System.out.println("Average time: " + averageTime + "\n");
}
}
}

```

The results of the simulation are shown in Figure 5.6. Note that with eight cashiers, the customers do not wait at all. The time of 120 seconds reflects only the time it takes to walk up and purchase the ticket. Increasing the number of cashiers to nine or ten or more will not improve the situation. Since the manager wants to keep the total average time to less than seven minutes (420 seconds), the simulation tells him that he should have six cashiers.

Number of Cashiers:	1	2	3	4	5	6	7	8	9	10
Average Time (sec):	5317	2325	1332	840	547	355	219	120	120	120

FIGURE 5.6 The results of the ticket counter simulation

5.5 A Queue ADT

As we did with stacks, we define a generic `QueueADT` interface that represents the queue operations, separating the general purpose of the operations from the variety of ways in which they could be implemented. A Java version of the `QueueADT` interface is shown in Listing 5.4, and its UML description is shown in Figure 5.7.

Note that in addition to the standard queue operations, we have included a `toString` method, just as we did with our stack collection. It is included for convenience and is not generally considered a classic operation on a queue.

LISTING 5.4

```
package jsjf;

/**
 * QueueADT defines the interface to a queue collection.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public interface QueueADT<T>
{
    /**
     * Adds one element to the rear of this queue.
     * @param element the element to be added to the rear of the queue
     */
    public void enqueue(T element);

    /**
     * Removes and returns the element at the front of this queue.
     * @return the element at the front of the queue
     */
    public T dequeue();

    /**
     * Returns without removing the element at the front of this queue.
     * @return the first element in the queue
     */
    public T first();

    /**
     * Returns true if this queue contains no elements.
     * @return true if this queue is empty
     */
    public boolean isEmpty();

    /**
     * Returns the number of elements in this queue.
     * @return the integer representation of the size of the queue
     */
    public int size();

    /**
     * Returns a string representation of this queue.
     * @return the string representation of the queue
     */
    public String toString();
}
```

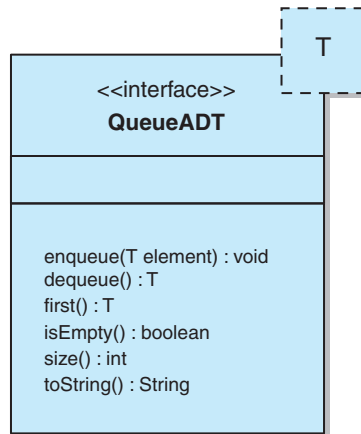


FIGURE 5.7 The `QueueADT` interface in UML

5.6 A Linked Implementation of a Queue

Because a queue is a linear collection, we can implement a queue as a linked list of `LinearNode` objects, as we did with stacks. The primary difference is that we will have to operate on both ends of the list. Therefore, in addition to a reference (called `head`) pointing to the first element in the list, we will keep track of a second reference (called `tail`) that points to the last element in the list. We will also use an integer variable called `count` to keep track of the number of elements in the queue.

Does it make a difference to which end of the list we add, or enqueue, elements and from which end of the list we remove, or dequeue, elements? If our linked list is singly linked, meaning that each node has only a pointer to the node behind it in the list, then yes, it does make a difference. In the case of the enqueue operation, it will not matter whether we add new elements to the head or the tail of the list. The processing steps will be very similar. If we add to the head of the list, then we will set the `next` pointer of the new node to point to the `head` of the list and will set the `head` variable to point to the new node. If we add to the `tail` of the list, then we will set the `next` pointer of the node at the `tail` of the list to point to the new node and will set the `tail` of the list to point to the new node. In both cases, all of these processing steps are $O(1)$, so the time complexity of the enqueue operation will be $O(1)$.

KEY CONCEPT

A linked implementation of a queue is facilitated by references to the first and last elements of the linked list.

The difference between our two choices, adding to the `head` or the `tail` of the list, occurs with the `dequeue` operation. If we enqueue at the `tail` of the list and dequeue from the `head` of the list, then to dequeue we simply set a temporary variable to point to the element at the `head` of the list and then set the `head` variable to the value of the next pointer of the first node. Both processing steps are $O(1)$, so the operation will be $O(1)$. However, if we enqueue at the `head` of the list and dequeue at the `tail` of the list, our processing steps become more interesting. In order to dequeue from the `tail` of the list, we must set a temporary variable to point to the element at the `tail` of the list and then set the `tail` pointer to point to the node before the current `tail`. Unfortunately, in a singly linked list, we cannot get to this node without traversing the list. Therefore, if we chose to enqueue at the `head` and dequeue at the `tail`, the dequeue operation will be $O(n)$ instead of $O(1)$ as it is with our other choice. Thus, we choose to enqueue at the `tail` and dequeue at the `head` of our singly linked list. Keep in mind that a doubly linked list would solve the problem of having to traverse the list, and thus it would not matter which end was which in a doubly linked implementation.

Figure 5.8 depicts this strategy for implementing a queue. It shows a queue that has had the elements A, B, C, and D added to the queue, or enqueued, in that order.

Remember that Figure 5.8 depicts the general case. We always have to be careful to maintain our references accurately in special cases. For an empty queue, the `head` and `tail` references are both null, and the `count` is zero. If there is exactly

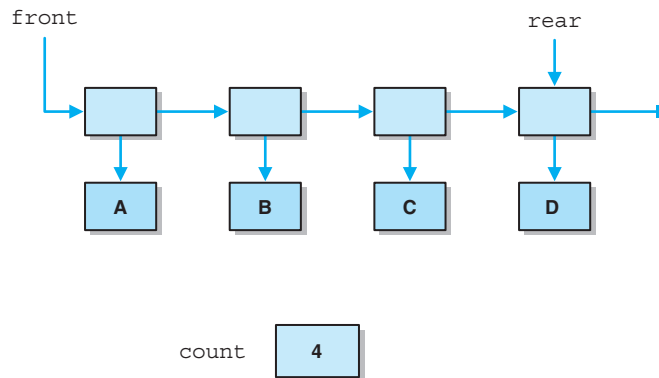


FIGURE 5.8 A linked implementation of a queue

one element in the queue, both the `head` and `tail` references point to the same object.

Let's explore the implementation of the queue operations using this linked-list approach. The header, class-level data, and constructors for our linked implementation of a queue are provided for context.

```
package jsjf;

import jsjf.exceptions.*;

/**
 * LinkedQueue represents a linked implementation of a queue.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class LinkedQueue<T> implements QueueADT<T>
{
    private int count;
    private LinearNode<T> head, tail;

    /**
     * Creates an empty queue.
     */
    public LinkedQueue()
    {
        count = 0;
        head = tail = null;
    }
}
```

The enqueue Operation

The `enqueue` operation requires that we put the new element on the tail of the queue. In the general case, that means setting the `next` reference of the current last element to the new one, and resetting the `tail` reference to the new last element. However, if the queue is currently empty, the `head` reference must also be set to the new (and only) element. This operation can be implemented as follows:

```

/**
 * Adds the specified element to the tail of this queue.
 * @param element the element to be added to the tail of the queue
 */
public void enqueue(T element)
{
    LinearNode<T> node = new LinearNode<T>(element);

    if (isEmpty())
        head = node;
    else
        tail.setNext(node);

    tail = node;
    count++;
}

```

Note that the next reference of the new node need not be explicitly set in this method because it has already been set to null in the constructor for the `LinearNode` class. The `tail` reference is set to the new node in either case and the `count` is incremented. Implementing the queue operations with sentinel nodes is left as an exercise. As we discussed earlier, this operation is $O(1)$.

Figure 5.9 shows the queue from Figure 5.8 after element E has been added.

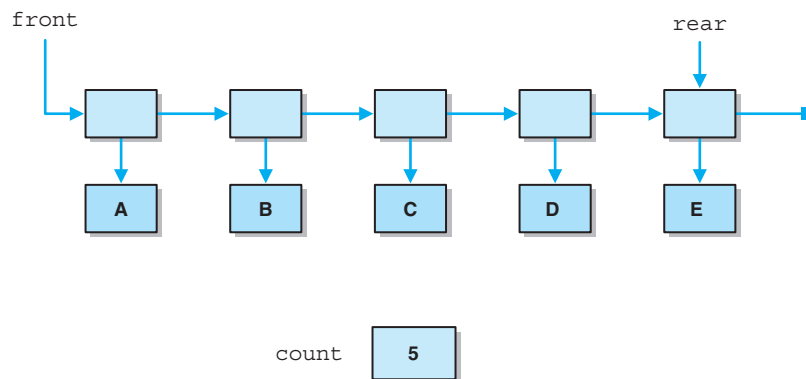


FIGURE 5.9 The queue after adding element E

The dequeue Operation

The first thing to do when implementing the dequeue operation is to ensure that there is at least one element to return. If not, an `EmptyCollectionException` is thrown. As we did with our stack collection in Chapters 3 and 4, it makes sense to employ a generic `EmptyCollectionException` to which we can pass a parameter specifying which collection we are dealing with. If there is at least one element in the queue, the first one in the list is returned, and the `head` reference is updated:

```
/**
 * Removes the element at the head of this queue and returns a
 * reference to it.
 * @return the element at the head of this queue
 * @throws EmptyCollectionException if the queue is empty
 */
public T dequeue() throws EmptyCollectionException
{
    if (isEmpty())
        throw new EmptyCollectionException("queue");

    T result = head.getElement();
    head = head.getNext();
    count--;

    if (isEmpty())
        tail = null;

    return result;
}
```

For the dequeue operation, we must consider the situation in which we are returning the only element in the queue. If, after removal of the head element, the queue is now empty, then the `tail` reference is set to null. Note that in this case, the `head` will be null because it was set equal to the next reference of the last element in the list. Again, as we discussed earlier, the dequeue operation for our implementation is $O(1)$.

KEY CONCEPT

The enqueue and dequeue operations work on opposite ends of the collection.

DESIGN FOCUS

The same goals of reuse that apply to other classes apply to exceptions as well. The `EmptyCollectionException` class is a good example of this. It is an exceptional case that will be the same for any collection that we create (such as attempting to perform an operation on the collection that cannot be performed if the collection is empty). Thus, creating a single exception with a parameter that enables us to designate which collection has thrown the exception is an excellent example of designing for reuse.

Figure 5.10 shows the result of a `dequeue` operation on the queue from Figure 5.9. The element A at the head of the queue is removed and returned to the user.

Note that, unlike the `push` and `pop` operations on a stack, the `dequeue` operation is not the inverse of `enqueue`. That is, Figure 5.10 is not identical to our original configuration shown in Figure 5.8, because the `enqueue` and `dequeue` operations are working on opposite ends of the collection.

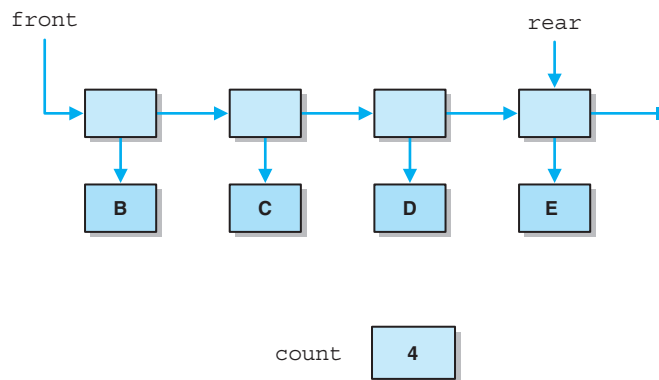


FIGURE 5.10 The queue after a `dequeue` operation

Other Operations

The remaining operations in the linked queue implementation are fairly straightforward and are similar to those in the stack collection. The `first` operation is implemented by returning a reference to the element at the head of the queue. The `isEmpty` operation returns `true` if the count of elements is 0, and `false` otherwise. The `size` operation simply returns the count of elements in the queue. Finally, the `toString` operation returns a string made up of the `toString` results of each individual element. These operations are left as programming projects.

5.7 Implementing Queues: With Arrays

One array-based strategy for implementing a queue is to fix one end of the queue (say, the front) at index 0 of the array. The elements are stored contiguously in the array. Figure 5.11 depicts a queue stored in this manner, assuming elements A, B, C, and D have been added to the queue in that order.



VideoNote

An array-based queue implementation.

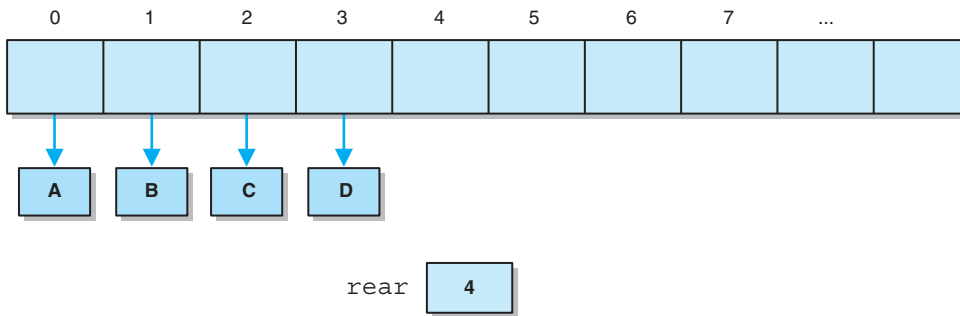


FIGURE 5.11 An array implementation of a queue

In a manner similar to the `top` variable in the `ArrayStack` implementation, the integer variable `rear` is used to indicate the next open cell in the array. Note that it also represents the number of elements in the queue.

This strategy assumes that the first element in the queue is always stored at index 0 of the array. Because queue processing affects both ends of the collection, this strategy will require that we shift the elements whenever an element is removed from the queue. This required shifting of elements would make the `dequeue` operation $O(n)$. Just as in our discussion of the complexity of our singly linked list implementation above, making a poor choice in our array implementation could lead to less than optimal efficiency.

Would it make a difference if we fixed the rear of the queue, instead of the front, at index 0 of the array? Keep in mind that when we enqueue an element onto the queue, we do so at the rear of the queue. This would mean that each `enqueue` operation would result in shifting all of the elements in the queue up one position in the array, making the `enqueue` operation $O(n)$.

KEY CONCEPT

Because queue operations modify both ends of the collection, fixing one end at index 0 requires that elements be shifted.

KEY CONCEPT

The shifting of elements in a noncircular array implementation creates an $O(n)$ complexity.

**DESIGN FOCUS**

It is important to note that this fixed array implementation strategy, which was very effective in our implementation of a stack, is not nearly as efficient for a queue. This is an important example of matching the data structure used to implement a collection with the collection itself. The fixed array strategy was efficient for a stack because all of the activity (adding and removing elements) was on one end of the collection and thus on one end of the array. With a queue, now that we are operating on both ends of the collection and order does matter, the fixed array implementation is much less efficient.

The key is to not fix either end. As elements are dequeued, the front of the queue will move further into the array. As elements are enqueued, the rear of the queue will also move further into the array. The challenge comes when the rear of the queue reaches the end of the array. Enlarging the array at this point is not a practical solution, and it does not make use of the now-empty space in the lower indexes of the array.

KEY CONCEPT

Treating arrays as circular eliminates the need to shift elements in an array queue implementation.

To make this solution work, we will use a *circular array* to implement the queue, defined in a class called `CircularArrayQueue`. A circular array is not a new construct—it is just a way to think about the array used to store the collection. Conceptually, the array is used as a circle, whose last index is followed by the first index. A circular array storing a queue is shown in Figure 5.12.

Two integer values are used to represent the front and rear of the queue. These values change as elements are added and removed. Note that the value of `front` represents the location where the first element in the queue is stored, and the value of `rear` represents the next available slot in the array (not where the last element is stored). Using `rear` in this manner is consistent with our other array implementation. Note, however, that the value of `rear` no longer represents the number of elements in the queue. We will use a separate integer value to keep a count of the elements.

When the rear of the queue reaches the end of the array, it “wraps around” to the front of the array. The elements of the queue can therefore straddle the end of the array, as shown in Figure 5.13, which assumes the array can store 100 elements.

Using this strategy, once an element has been added to the queue, it stays in one location in the array until it is removed with a `dequeue` operation. No elements need to be shifted as elements are added or removed. This approach requires, however, that we carefully manage the values of `front` and `rear`.

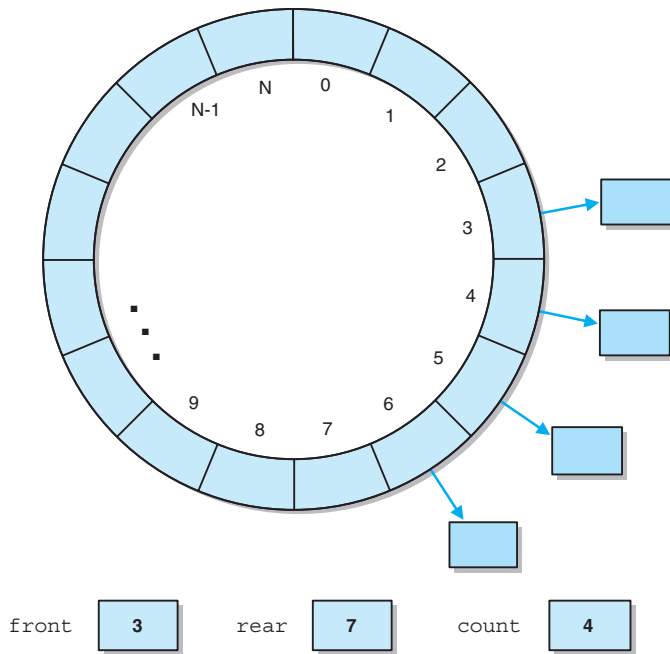


FIGURE 5.12 A circular array implementation of a queue

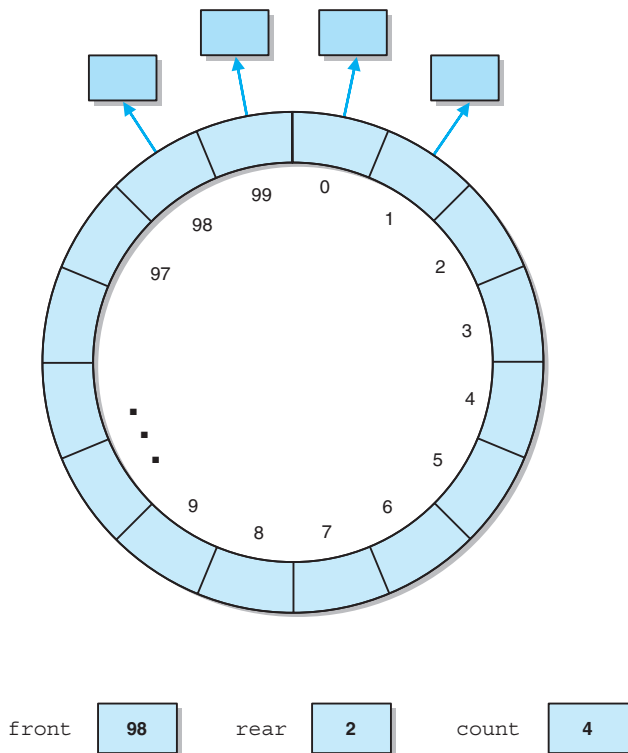


FIGURE 5.13 A queue straddling the end of a circular array

Let's look at another example. Figure 5.14 shows a circular array (drawn linearly) with a capacity of ten elements. Initially it is shown after elements A through H have been enqueued. It is then shown after the first four elements (A through D) have been dequeued. Finally, it is shown after elements I, J, K, and L have been enqueued, which causes the queue to wrap around the end of the array.

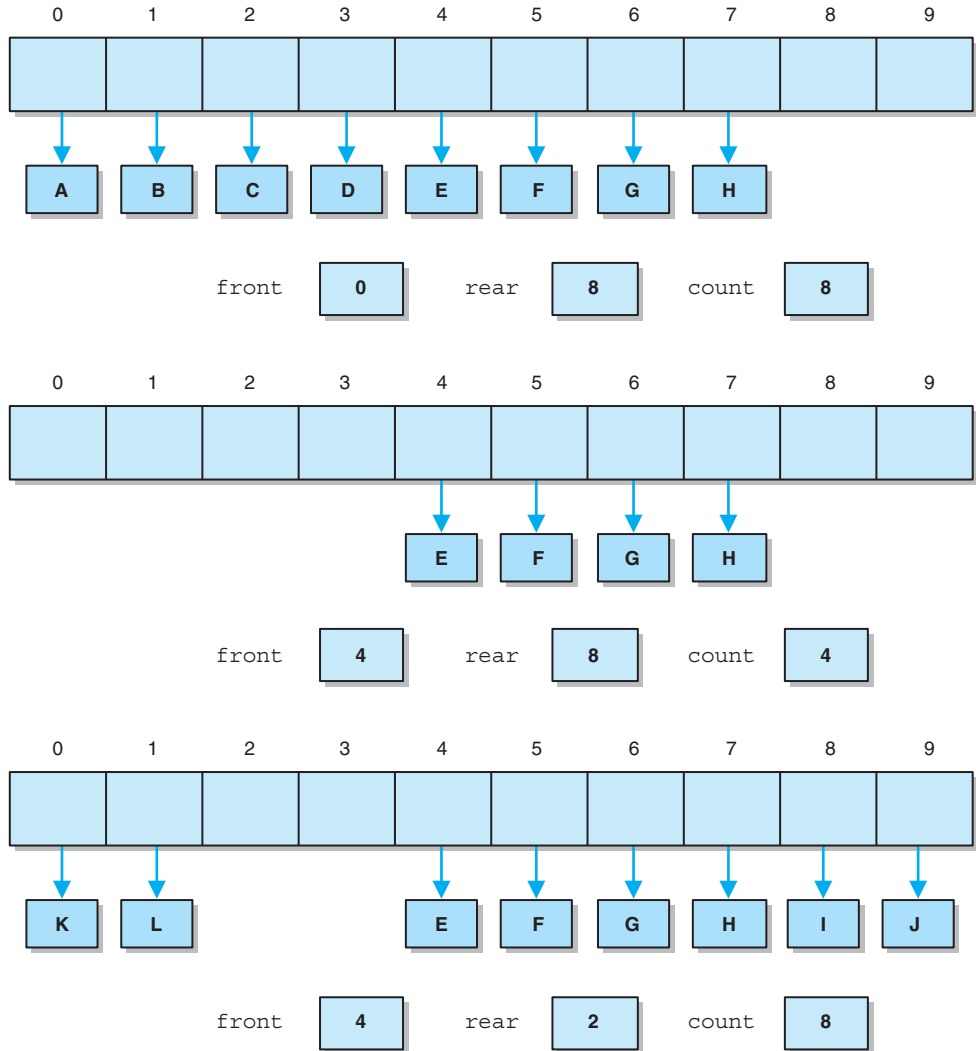


FIGURE 5.14 Changes in a circular array implementation of a queue

The header, class-level data, and constructors for our circular array implementation of a queue are provided for context:

```
package jsjf;

package jsjf.exceptions.*;

/**
 * CircularArrayQueue represents an array implementation of a queue in
 * which the indexes for the front and rear of the queue circle back to 0
 * when they reach the end of the array.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class CircularArrayQueue<T> implements QueueADT<T>
{
    private final static int DEFAULT_CAPACITY = 100;
    private int front, rear, count;
    private T[] queue;

    /**
     * Creates an empty queue using the specified capacity.
     * @param initialCapacity the initial size of the circular array queue
     */
    public CircularArrayQueue (int initialCapacity)
    {
        front = rear = count = 0;
        queue = (T[]) (new Object[initialCapacity]);
    }

    /**
     * Creates an empty queue using the default capacity.
     */
    public CircularArrayQueue()
    {
        this(DEFAULT_CAPACITY);
    }
}
```

The enqueue Operation

In general, after an element is enqueued, the value of `rear` is incremented. But when an enqueue operation fills the last cell of the array (at the largest index), the value of `rear` must be set to 0, indicating that the next element should be stored at index 0. The appropriate update to the value of `rear` can be accomplished in

one calculation by using the remainder operator (%). Recall that the remainder operator returns the remainder after dividing the first operand by the second. Therefore, if `queue` is the name of the array storing the queue, the following line of code will update the value of `rear` appropriately:

```
rear = (rear+1) % queue.length;
```

Let's try this calculation, assuming we have an array of size 10. If `rear` is currently 5, it will be set to $6\%10$, or 6. If `rear` is currently 9, it will be set to $10\%10$, or 0. Try this calculation using various situations to see that it works no matter how big the array is.

Given this strategy, the enqueue operation can be implemented as follows:

```
/**
 * Adds the specified element to the rear of this queue, expanding
 * the capacity of the queue array if necessary.
 * @param element the element to add to the rear of the queue
 */
public void enqueue(T element)
{
    if (size() == queue.length)
        expandCapacity();

    queue[rear] = element;
    rear = (rear+1) % queue.length;

    count++;
}
```

Circular Increment

increment regularly

```
rear = (rear + 1) % queue.length;
```

wrap back to 0 if appropriate

Note that this implementation strategy will still allow the array to reach capacity. As with any array-based implementation, all cells in the array may become filled. This implies that the rear of the queue has “caught up” to the front of the queue. To add another element, the array would have to be enlarged. Keep in mind, however, that the elements of the existing array must be copied into the new array in their proper order in the queue, which is not necessarily the order in which they appear in the current array. This makes the private `expandCapacity` method slightly more complex than the one we used for stacks:

```
/**
 * Creates a new array to store the contents of this queue with
 * twice the capacity of the old one.
 */
private void expandCapacity()
{
    T[] larger = (T[]) (new Object[queue.length * 2]);

    for (int scan = 0; scan < count; scan++)
    {
        larger[scan] = queue[front];
        front = (front + 1) % queue.length;
    }

    front = 0;
    rear = count;
    queue = larger;
}
```

The dequeue Operation

Likewise, after an element is dequeued, the value of `front` is incremented. After enough dequeue operations, the value of `front` will reach the last index of the array. After removal of the element at the largest index, the value of `front` must be set to 0 instead of being incremented. The same calculation we used to set the value of `rear` in the enqueue operation can be used to set the value of `front` in the dequeue operation:

```
/**
 * Removes the element at the front of this queue and returns a
 * reference to it.
 * @return the element removed from the front of the queue
 * @throws EmptyCollectionException if the queue is empty
 */
```

```

public T dequeue() throws EmptyCollectionException
{
    if (isEmpty())
        throw new EmptyCollectionException("queue");

    T result = queue[front];
    queue[front] = null;
    front = (front+1) % queue.length;

    count--;

    return result;
}

```

Other Operations

Operations such as `toString` become a bit more complicated using this approach, because the elements are not stored starting at index 0 and may wrap around the end of the array. These methods have to take the current situation into account. All of the other operations for a circular array queue are left as programming projects.

5.8 Double-Ended Queues (Deque)

A deque, or double-ended queue, is an extension of the concept of a queue that allows adding, removing, and viewing elements from both ends of the queue. As mentioned in Chapter 3, the Java API provides the `Deque` interface, which, like the `Queue` interface, is implemented by the `LinkedList` class. Just like the `Queue` interface, the `Deque` interface provides two versions of each operation: one that will throw an exception and one that will return a `boolean`.

Interestingly, the `Deque` interface also provides implementations of the basic stack operations `push`, `pop`, and `peek`. In fact, Oracle now recommends that the `Deque` interface be used in place of the `java.util.stack` class.

Summary of Key Concepts

- Queue elements are processed in a FIFO manner—the first element in is the first element out.
- A queue is a convenient collection for storing a repeating code key.
- Simulations are often implemented using queues to represent waiting lines.
- A linked implementation of a queue is facilitated by references to the first and last elements of the linked list.
- The enqueue and dequeue operations work on opposite ends of the collection.
- Because queue operations modify both ends of the collection, fixing one end at index 0 requires that elements be shifted.
- The shifting of elements in a noncircular array implementation creates an $O(n)$ complexity.
- Treating arrays as circular eliminates the need to shift elements in an array queue implementation.

Summary of Terms

Caesar cipher A simple message encoding technique in which letters are shifted along the alphabet by a constant amount.

circular array An array that is treated as circular, meaning that incrementing the last index value in the array wraps around back to the first element.

dequeue A queue operation in which an element is removed from the front of the queue.

enqueue A queue operation in which an element is added to the rear of the queue.

FIFO (1) First in, first out; (2) A description of a collection in which the first element added will be the first element removed.

queue A linear collection whose elements are added on one end and removed from the other.

repeating key A list of integer values used to shift letters by varying amounts in an improved version of a Caesar cipher.

Self-Review Questions

- SR 5.1 Describe the queue data structure and processing of elements in it.
- SR 5.2 What are the purposes of the enqueue and dequeue operations in a queue?

- SR 5.3 What are the queue operations provided in the Queue interface of the Java Collections API?
- SR 5.4 What are the advantages of using a linked implementation of a queue as opposed to an array implementation?
- SR 5.5 Why is shifting of elements required in a linear array-based implementation of a queue?
- SR 5.6 Why is a fixed array implementation strategy for a queue inefficient?
- SR 5.7 How is the problem of shifting elements solved?

Exercises

- EX 5.1 Hand trace a queue `x` through the following operations:
- ```
X.enqueue(new Integer(14));
X.enqueue(new Integer(11));
Object Y = X.dequeue();
X.enqueue(new Integer(18));
X.enqueue(new Integer(12));
X.enqueue(new Integer(15));
X.enqueue(new Integer(13));
Object Y = X.dequeue();
X.enqueue(new Integer(14));
X.enqueue(new Integer(19));
```
- EX 5.2 Given the queue `x` that results from Exercise 5.1, what would be the result of each of the following?
- `Y = X.dequeue();`  
`Z = X.dequeue();`
  - `X.first();`
  - `X.enqueue(25);`
  - `X.first();`
- EX 5.3 What would be the time complexity of the `size` operation for each of the implementations if there were not a `count` variable?
- EX 5.4 Under what circumstances could the `head` and `tail` references for the linked implementation or the `front` and `rear` references of the array implementation be equal?
- EX 5.5 Hand trace the ticket counter problem for 22 customers and 4 cashiers. Graph the total process time for each person. What can you surmise from these results?

- EX 5.6 Compare and contrast the `enqueue` method of the `LinkedListQueue` class and the `push` method of the `LinkedListStack` class from Chapter 4.
- EX 5.7 Describe two different ways in which the `isEmpty` method of the `LinkedListQueue` class could be implemented.
- EX 5.8 Name five everyday examples of a queue other than those discussed in this chapter.
- EX 5.9 Explain why the array implementation of a stack does not require elements to be shifted, but the noncircular array implementation of a queue does.
- EX 5.10 Suppose the `count` variable was not used in the `CircularArrayQueue` class. Explain how you could use the values of `front` and `rear` to compute the number of elements in the list.

## Programming Projects

- PP 5.1 Complete the implementation of the `LinkedListQueue` class presented in this chapter. Specifically, complete the implementations of the `first`, `isEmpty`, `size`, and `toString` methods.
- PP 5.2 Complete the implementation of the `CircularArrayQueue` class described in this chapter, including all methods.
- PP 5.3 Write a version of the `CircularArrayQueue` class that grows the list in the direction opposite to the direction in which the version described in this chapter grows the list.
- PP 5.4 All of the implementations in this chapter use a `count` variable to keep track of the number of elements in the queue. Rewrite the linked implementation without a `count` variable.
- PP 5.5 All of the implementations in this chapter use a `count` variable to keep track of the number of elements in the queue. Rewrite the circular array implementation without a `count` variable.
- PP 5.6 A data structure called a deque is closely related to a queue. The name *deque* stands for “double-ended queue.” The difference between the two is that with a deque, you can insert, remove, or view from either end of the queue. Implement a deque using arrays.
- PP 5.7 Implement the deque from Programming Project 5.6 using links. (*Hint:* Each node will need a `next` reference and a `previous` reference.)

- PP 5.8 Create a graphical application that provides buttons to enqueue and dequeue elements from a queue, a text field to accept a string as input for enqueue, and a text area to display the contents of the queue after each operation.
- PP 5.9 Create a system using a stack and a queue to test whether a given string is a palindrome (that is, whether the characters read the same both forward and backward).
- PP 5.10 Create a system to simulate vehicles at an intersection. Assume that there is one lane going in each of four directions, with stoplights facing each direction. Vary the arrival average of vehicles in each direction and the frequency of the light changes to view the “behavior” of the intersection.

### Answers to Self-Review Questions

- SRA 5.1 A queue is a linear collection whose elements are added on one end and removed from the other. The queue elements are processed in a first in, first out (FIFO) manner. Elements are removed from a queue in the same order in which they are placed on the queue.
- SRA 5.2 The `enqueue` operation adds an element to the end of the queue and the `dequeue` operation removes an element from the front of the queue.
- SRA 5.3 The `Queue` interface has the *element* method that returns the first element. It has `add` and `offer` methods for inserting elements and the *poll* and *remove* methods for removing an element from the queue.
- SRA 5.4 A linked implementation of queue allocates space only as it is needed whereas, the array implementation will allocate much more space than it needs initially.
- SRA 5.5 Since queue operations modify both ends of the collection, fixing one end at index 0 requires that elements be shifted.
- SRA 5.6 A fixed array implementation is inefficient because shifting of elements in a noncircular array implementation creates an  $O(n)$  complexity.
- SRA 5.7 Treating arrays as circular eliminates the need to shift elements in an array queue implementation.



# Lists

# 6

The concept of a list is familiar to all of us. You may make to-do lists, lists of items to buy at the grocery store, and lists of friends to invite to a party. You might number the items in a list or keep them in alphabetical order. In other situations you may simply keep the items in a particular order that makes the most sense to you. This chapter explores the concept of a list collection and some ways in which such collections can be managed.

## CHAPTER OBJECTIVES

- Examine various types of list collections.
- Demonstrate how lists can be used to solve problems.
- Define a list abstract data type.
- Examine and compare list implementations.

## 6.1 A List Collection

Let's begin by differentiating between a linked list and the concept of a list collection. As we've seen in previous chapters, a linked list is an implementation strategy that uses references to create links between objects. We used linked lists in Chapters 4 and 5 to help us implement stack and queue collections, respectively.

A list collection, on the other hand, is a conceptual notion—the idea of keeping things organized in a linear list. Just like stacks and queues, a list can be implemented using linked lists or arrays. A list collection has no inherent capacity; it can grow as large as needed.

Both stacks and queues are linear structures and might be thought of as lists, but elements can be added and removed only on the ends. List collections are more general; elements can be added and removed in the interior of the list as well as on the ends.

Furthermore, there are three types of list collections:

- *Ordered lists*, whose elements are ordered by some inherent characteristic of the elements
- *Unordered lists*, whose elements have no inherent order but are ordered by their placement in the list
- *Indexed lists*, whose elements can be referenced using a numeric index

### KEY CONCEPT

List collections can be categorized as ordered, unordered, or indexed.

### KEY CONCEPT

The elements of an ordered list have an inherent relationship defining their order.

An ordered list is based on some particular characteristic of the elements in the list. For example, you may keep a list of people ordered alphabetically by name, or you may keep an inventory list ordered by part number. The list is sorted on the basis of some key value. Any element added to an ordered list has a proper location in the list, given its key value and the key values of the elements already in the list. Figure 6.1 shows a conceptual view of an ordered list, in which the elements are ordered by an integer key value. Adding a value to the list involves finding the new element's proper, sorted position among the existing elements.

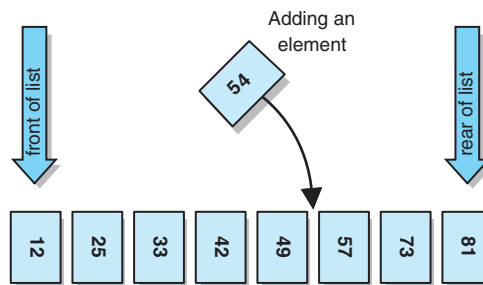


FIGURE 6.1 A conceptual view of an ordered list



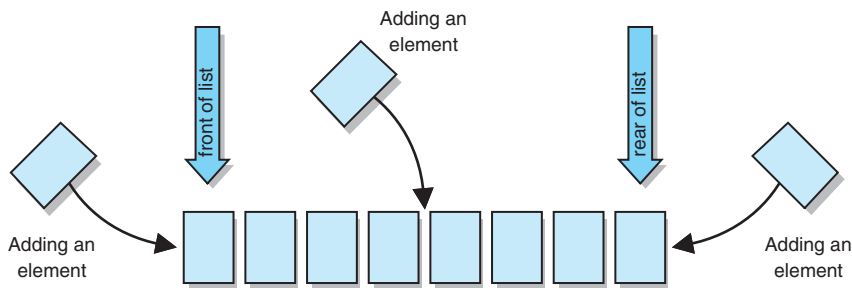
VideoNote

List categories.

The placement of elements in an unordered list is not based on any inherent characteristic of the elements. Don't let the name mislead you. The elements in an unordered list are kept in a particular order, but that order is not based on the elements themselves. The client using the list determines the order of the elements. Figure 6.2 shows a conceptual view of an unordered list. A new element can be put at the front or rear of the list, or it can be inserted after a particular element already in the list.

### KEY CONCEPT

The elements of an unordered list are kept in whatever order the client chooses.

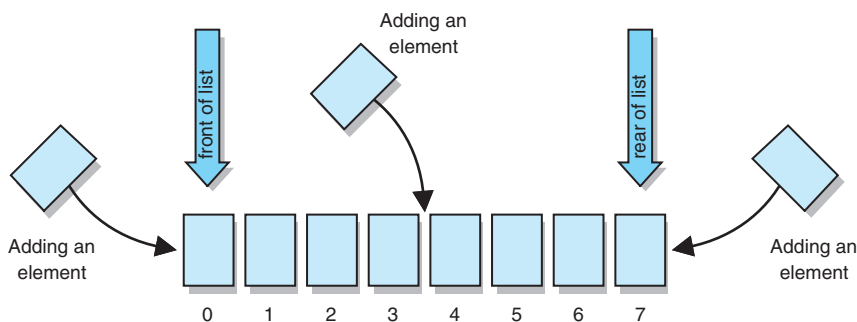


**FIGURE 6.2** A conceptual view of an unordered list

An indexed list is similar to an unordered list in that there is no inherent relationship among the elements that determines their order in the list. The client using the list determines the order of the elements. However, in addition, each element can be referenced by a numeric index that begins at 0 at the front of the list and continues contiguously until the end of the list. Figure 6.3 shows a conceptual view of an indexed list. A new element can be inserted into the list at any position, including at the front or rear of the list. Every time a change occurs in the list, the indexes are adjusted to stay in order and contiguous.

### KEY CONCEPT

An indexed list maintains a contiguous numeric index range for its elements.



**FIGURE 6.3** A conceptual view of an indexed list

Note the primary difference between an indexed list and an array: An indexed list keeps its indexes contiguous. If an element is removed, the positions of other elements “collapse” to eliminate the gap. When an element is inserted, the indexes of other elements are shifted to make room.

## DESIGN FOCUS

Is it possible that a list could be both an ordered list and an indexed list? Possible perhaps, but not very meaningful. If a list were both ordered and indexed, what would happen if a client application attempted to add an element at a particular index or to change an element at a particular index such that it is not in the proper order? Which rule would have precedence, index position or order?

## 6.2 Lists in the Java Collections API

The list classes provided in the Java API primarily support the concept of an indexed list. To some extent, they overlap with the concept of an unordered list. Note, though, that the Java API does not have any classes that directly implement an ordered list as described above.

### KEY CONCEPT

The Java API does not provide a class that implements an ordered list.

You’re probably already familiar with the `ArrayList` class from the Java API. It is a favorite among Java programmers, because it provides a quick way to manage a set of objects. Its counterpart, the `LinkedList` class, provides the same basic functionality with, as the name implies, an underlying linked implementation. Both store elements defined by a generic parameter `E`.

Both `ArrayList` and `LinkedList` implement the `java.util.List` interface. Some of the methods in the `List` interface are shown in Figure 6.4.

| Method                                 | Description                                           |
|----------------------------------------|-------------------------------------------------------|
| <code>add(E element)</code>            | Adds an element to the end of the list.               |
| <code>add(int index, E element)</code> | Inserts an element at the specified index.            |
| <code>get(int index)</code>            | Returns the element at the specified index.           |
| <code>remove(int index)</code>         | Removes the element at the specified index.           |
| <code>remove(E object)</code>          | Removes the first occurrence of the specified object. |
| <code>set(int index, E element)</code> | Replaces the element at the specified index.          |
| <code>size()</code>                    | Returns the number of elements in the list.           |

FIGURE 6.4 Some methods in the `java.util.List` interface

Before looking at our own implementation of lists, let's look at a couple of examples that use lists provided by the Java API.

## 6.3 Using Unordered Lists: Program of Study

The list of courses a student takes in order to fulfill degree requirements is sometimes called a program of study. Let's look at an example that manages a simplified program of study. We'll use the `LinkedList` class from the Java API, adding some unordered list operations, to manage the list of courses.

Listing 6.1 contains a `main` method that creates a `ProgramOfStudy` object and uses it to manage a few specific courses. It first adds a few initial courses, one after the other, to the end of the list. Then a second CS course is inserted into the list after the existing CS course. Then a specific THE course is found, and its grade is updated. Finally, a GER course is replaced by a FRE course.

After manipulating the list of courses in these specific ways, the `main` method prints the entire `ProgramOfStudy` object and then saves it to disk so that it can be retrieved and modified further at a later time.

### LISTING 6.1

```
import java.io.IOException;

/**
 * Demonstrates the use of a list to manage a set of objects.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class POSTester
{
 /**
 * Creates and populates a Program of Study. Then saves it using
 * serialization.
 */
 public static void main(String[] args) throws IOException
 {
 ProgramOfStudy pos = new ProgramOfStudy();
```



LISTING 6.1 *continued*

```

pos.addCourse(new Course("CS", 101, "Introduction to Programming", "A-"));
pos.addCourse(new Course("ARCH", 305, "Building Analysis", "A"));
pos.addCourse(new Course("GER", 210, "Intermediate German"));
pos.addCourse(new Course("CS", 320, "Computer Architecture"));
pos.addCourse(new Course("THE", 201, "The Theatre Experience"));

Course arch = pos.find("CS", 320);
pos.addCourseAfter(arch, new Course("CS", 321, "Operating Systems"));

Course theatre = pos.find("THE", 201);
theatre.setGrade("A-");

Course german = pos.find("GER", 210);
pos.replace(german, new Course("FRE", 110, "Beginning French", "B+"));

System.out.println(pos);

pos.save("ProgramOfStudy");
}
}

```

The `ProgramOfStudy` class is shown in Listing 6.2 and the `Course` class is shown in Listing 6.3. First note that the instance variable called `list` is declared to be of type `List<Course>`, which refers to the interface. In the constructor, a new `LinkedList<Course>` object is instantiated. If desired, this could be changed to an `ArrayList<Course>` object without any other changes to the class.

The methods `addCourse`, `find`, `addCourseAfter`, and `replace` perform the various core operations needed to update the program of study. They essentially add unordered list operations to the basic list operations provided by the `LinkedList` class.

The `iterator` method returns an `Iterator` object. This method was not used in the `ProgramOfStudyTester` program, but it is a key operation. Iterators are discussed in detail in Chapter 7.

Finally, the `save` and `load` methods are used to write the `ProgramOfStudy` object to a file and to read it back in, respectively. Unlike text-based I/O operations we've seen in previous examples, this one uses a process called *serialization* to read and write the object as a binary stream. So with just a few lines of code, an object can be stored with its current state completely intact. In this case, that means all courses currently stored in the Program of Study list are stored as part of the object.

Note that the `ProgramOfStudy` and `Course` classes implement the `Serializable` interface. In order for an object to be saved using *serialization*, its class must implement `Serializable`. There are no methods in the `Serializable` interface—it is used simply to indicate that the object may be converted to a serialized representation. The `ArrayList` and `LinkedList` classes implement `Serializable`.

### Serializable

```
public class Course implements Serializable
```

indicates that this class can be serialized  
The `Serializable` interface contains no methods.

### LISTING 6.2

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
```

## LISTING 6.2

*continued*

```

/**
 * Represents a Program of Study, a list of courses taken and planned, for an
 * individual student.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class ProgramOfStudy implements Iterable<Course>, Serializable
{
 private List<Course> list;

 /**
 * Constructs an initially empty Program of Study.
 */
 public ProgramOfStudy()
 {
 list = new LinkedList<Course>();
 }

 /**
 * Adds the specified course to the end of the course list.
 *
 * @param course the course to add
 */
 public void addCourse(Course course)
 {
 if (course != null)
 list.add(course);
 }

 /**
 * Finds and returns the course matching the specified prefix and number.
 *
 * @param prefix the prefix of the target course
 * @param number the number of the target course
 * @return the course, or null if not found
 */
 public Course find(String prefix, int number)
 {
 for (Course course : list)
 if (prefix.equals(course.getPrefix()) &&
 number == course.getNumber())
 return course;
 return null;
 }
}

```

## LISTING 6.2

*continued*

```
/**
 * Adds the specified course after the target course. Does nothing if
 * either course is null or if the target is not found.
 *
 * @param target the course after which the new course will be added
 * @param newCourse the course to add
 */
public void addCourseAfter(Course target, Course newCourse)
{
 if (target == null || newCourse == null)
 return;

 int targetIndex = list.indexOf(target);
 if (targetIndex != -1)
 list.add(targetIndex + 1, newCourse);
}

/**
 * Replaces the specified target course with the new course. Does nothing if
 * either course is null or if the target is not found.
 *
 * @param target the course to be replaced
 * @param newCourse the new course to add
 */
public void replace(Course target, Course newCourse)
{
 if (target == null || newCourse == null)
 return;

 int targetIndex = list.indexOf(target);
 if (targetIndex != -1)
 list.set(targetIndex, newCourse);
}

/**
 * Creates and returns a string representation of this Program of Study.
 *
 * @return a string representation of the Program of Study
 */
public String toString()
{
 String result = "";
 for (Course course : list)
```

LISTING 6.2 *continued*

```

 result += course + "\n";
 return result;
}

/**
 * Returns an iterator for this Program of Study.
 *
 * @return an iterator for the Program of Study
 */
public Iterator<Course> iterator()
{
 return list.iterator();
}

/**
 * Saves a serialized version of this Program of Study to the specified
 * file name.
 *
 * @param fileName the file name under which the POS will be stored
 * @throws IOException
 */
public void save(String fileName) throws IOException
{
 FileOutputStream fos = new FileOutputStream(fileName);
 ObjectOutputStream oos = new ObjectOutputStream(fos);
 oos.writeObject(this);
 oos.flush();
 oos.close();
}

/**
 * Loads a serialized Program of Study from the specified file.
 *
 * @param fileName the file from which the POS is read
 * @return the loaded Program of Study
 * @throws IOException
 * @throws ClassNotFoundException
 */
public static ProgramOfStudy load(String fileName)
throws IOException, ClassNotFoundException
{

```

**LISTING 6.2** *continued*

```
 FileInputStream fis = new FileInputStream(fileName);
 ObjectInputStream ois = new ObjectInputStream(fis);
 ProgramOfStudy pos = (ProgramOfStudy) ois.readObject();
 ois.close();

 return pos;
 }
}
```

**LISTING 6.3**

```
import java.io.Serializable;

/**
 * Represents a course that might be taken by a student.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class Course implements Serializable
{
 private String prefix;
 private int number;
 private String title;
 private String grade;

 /**
 * Constructs the course with the specified information.
 *
 * @param prefix the prefix of the course designation
 * @param number the number of the course designation
 * @param title the title of the course
 * @param grade the grade received for the course
 */
 public Course(String prefix, int number, String title, String grade)
 {
 this.prefix = prefix;
 }
}
```

**LISTING 6.3** *continued*

```
 this.number = number;
 this.title = title;
 if (grade == null)
 this.grade = "";
 else
 this.grade = grade;
 }

 /**
 * Constructs the course with the specified information, with no grade
 * established.
 *
 * @param prefix the prefix of the course designation
 * @param number the number of the course designation
 * @param title the title of the course
 */
 public Course(String prefix, int number, String title)
 {
 this(prefix, number, title, "");
 }

 /**
 * Returns the prefix of the course designation.
 *
 * @return the prefix of the course designation
 */
 public String getPrefix()
 {
 return prefix;
 }

 /**
 * Returns the number of the course designation.
 *
 * @return the number of the course designation
 */
 public int getNumber()
 {
 return number;
 }
}
```

**LISTING 6.3***continued*

```
/**
 * Returns the title of this course.
 *
 * @return the prefix of the course
 */
public String getTitle()
{
 return title;
}

/**
 * Returns the grade for this course.
 *
 * @return the grade for this course
 */
public String getGrade()
{
 return grade;
}

/**
 * Sets the grade for this course to the one specified.
 *
 * @param grade the new grade for the course
 */
public void setGrade(String grade)
{
 this.grade = grade;
}

/**
 * Returns true if this course has been taken if a grade has been
 * received.
 *
 * @return true if this course has been taken and false otherwise
 */
public boolean taken()
{
 return !grade.equals("");
}
```



## LISTING 6.3

*continued*

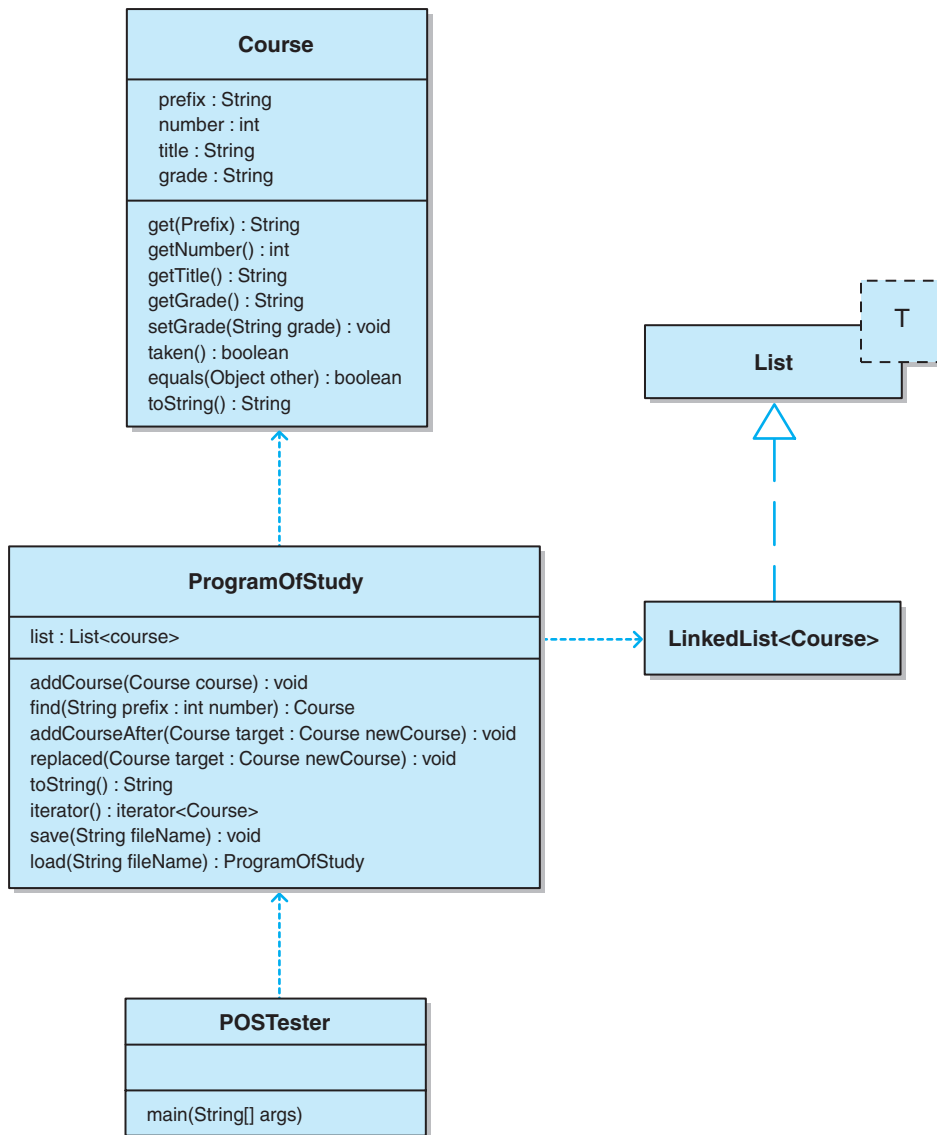
```

/**
 * Determines if this course is equal to the one specified, based on the
 * course designation (prefix and number).
 *
 * @return true if this course is equal to the parameter
 */
public boolean equals(Object other)
{
 boolean result = false;
 if (other instanceof Course)
 {
 Course otherCourse = (Course) other;
 if (prefix.equals(otherCourse.getPrefix()) &&
 number == otherCourse.getNumber())
 result = true;
 }
 return result;
}

/**
 * Creates and returns a string representation of this course.
 *
 * @return a string representation of the course
 */
public String toString()
{
 String result = prefix + " " + number + ": " + title;
 if (!grade.equals(""))
 result += " [" + grade + "];"
 return result;
}
}

```

A UML class diagram that describes the relationships among the classes in the ProgramOfStudy example is shown in Figure 6.5.



**FIGURE 6.5** UML description of the ProgramOfStudy program

## 6.4 Using Indexed Lists: Josephus

Flavius Josephus was a Jewish historian of the first century. Legend has it that he was one of a group of 41 Jewish rebels who decided to kill themselves rather than surrender to the Romans, who had them trapped. They decided to form a circle and to kill every third person until no one was left. Josephus, not wanting to die, calculated where he needed to stand so that he would be the last one alive. Thus was born a class of problems referred to as the Josephus problem. These problems involve finding the order of events when events in a list are not taken in order but, rather, are taken every  $i^{\text{th}}$  element in a cycle until none remains.

### KEY CONCEPT

The Josephus problem is a classic computing problem that is appropriately solved with indexed lists.

For example, suppose that we have a list of seven elements numbered from 1 to 7:

1 2 3 4 5 6 7

If we were to remove every third element from the list, the first element to be removed would be number 3, leaving the list

1 2 4 5 6 7

The next element to be removed would be number 6, leaving the list

1 2 4 5 7

The elements are thought of as being in a continuous cycle, so when we reach the end of the list, we continue counting at the beginning. Therefore, the next element to be removed would be number 2, leaving the list

1 4 5 7

The next element to be removed would be number 7, leaving the list

1 4 5

The next element to be removed would be number 5, leaving the list

1 4

The next-to-last element to be removed would be number 1, leaving the number 4 as the last element on the list.

Listing 6.4 illustrates a generic implementation of the Josephus problem, allowing the user to input the number of items in the list and the gap between elements. Initially, a list is filled with integers representing the soldiers. Each element is then removed from the list, one at a time, by computing the next index position in the list to be removed.

The one complication in this process is computation of the next index position to be removed. This is particularly interesting because the list collapses on itself

as elements are removed. For example, the element number 6 from our previous example should be the second element removed from the list. However, once element 3 has been removed from the list, element 6 is no longer in its original position. Instead of being at index position 5 in the list, it is now at index position 4.

#### LISTING 6.4

```
import java.util.*;

/**
 * Demonstrates the use of an indexed list to solve the Josephus problem.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class Josephus
{
 /**
 * Continue around the circle eliminating every nth soldier
 * until all of the soldiers have been eliminated.
 */
 public static void main(String[] args)
 {
 int numPeople, skip, targetIndex;
 List<String> list = new ArrayList<String>();
 Scanner in = new Scanner(System.in);

 // get the initial number of soldiers
 System.out.print("Enter the number of soldiers: ");
 numPeople = in.nextInt();
 in.nextLine();

 // get the number of soldiers to skip
 System.out.print("Enter the number of soldiers to skip: ");
 skip = in.nextInt();

 // load the initial list of soldiers
 for (int count = 1; count <= numPeople; count++)
 {
 list.add("Soldier " + count);
 }
 targetIndex = skip;
 System.out.println("The order is: ");
 }
}
```

**LISTING 6.4** *continued*

```

// Treating the list as circular, remove every nth element
// until the list is empty
while (!list.isEmpty())
{
 System.out.println(list.remove(targetIndex));
 if (list.size() > 0)
 targetIndex = (targetIndex + skip) % list.size();
}
}
}

```

## 6.5 A List ADT

Now let's explore our own implementation of a list collection. We'll go beyond what the Java API provides and include full implementations of unordered and ordered lists.

### KEY CONCEPT

Many common operations can be defined for all list types. The differences between them stem from how elements are added.

There is a set of operations that is common to both ordered and unordered lists. These common operations are shown in Figure 6.6. They include operations to remove and examine elements, as well as classic operations such as `isEmpty` and `size`. The `contains` operation is also supported by both list types, which enables the user to determine whether a list contains a particular element.

| Operation                | Description                                           |
|--------------------------|-------------------------------------------------------|
| <code>removeFirst</code> | Removes the first element from the list.              |
| <code>removeLast</code>  | Removes the last element from the list.               |
| <code>remove</code>      | Removes a particular element from the list.           |
| <code>first</code>       | Examines the element at the front of the list.        |
| <code>last</code>        | Examines the element at the rear of the list.         |
| <code>contains</code>    | Determines if the list contains a particular element. |
| <code>isEmpty</code>     | Determines if the list is empty.                      |
| <code>size</code>        | Determines the number of elements on the list.        |

**FIGURE 6.6** The common operations on a list

## Adding Elements to a List

The differences between ordered and unordered lists generally center on how elements are added to the list. In an ordered list, we need only specify the new element to add. Its position in the list is based on its key value. This operation is shown in Figure 6.7.

| Operation | Description                  |
|-----------|------------------------------|
| add       | Adds an element to the list. |

**FIGURE 6.7** The operation particular to an ordered list

An unordered list supports three variations of the add operation. Elements can be added to the front of the list, to the rear of the list, or after a particular element that is already in the list. These operations are shown in Figure 6.8.

| Operation  | Description                                                     |
|------------|-----------------------------------------------------------------|
| addToFront | Adds an element to the front of the list.                       |
| addToRear  | Adds an element to the rear of the list.                        |
| addAfter   | Adds an element after a particular element already in the list. |

**FIGURE 6.8** The operations particular to an unordered list

Conceptually, the operations particular to an indexed list make use of its ability to reference elements by their index. A new element can be inserted into the list at a particular index, or it can be added to the rear of the list without specifying an index at all. Note that if an element is inserted or removed, the elements at higher indexes are either shifted up to make room or shifted down to close the gap. Alternatively, the element at a particular index can be set, which overwrites the element currently at that index and therefore does not cause other elements to shift.

We can capitalize on the fact that both ordered lists and unordered lists share a common set of operations. These operations need to be defined only once. Therefore, we will define three list interfaces: one with the common operations and two with the operations particular to each list type. Inheritance can be used with interfaces, just as it can with classes. The interfaces of the particular list types extend the common list definition. This relationship among the interfaces is shown in Figure 6.9.

Listings 6.5 through 6.7 show the Java interfaces corresponding to the UML diagram in Figure 6.9.

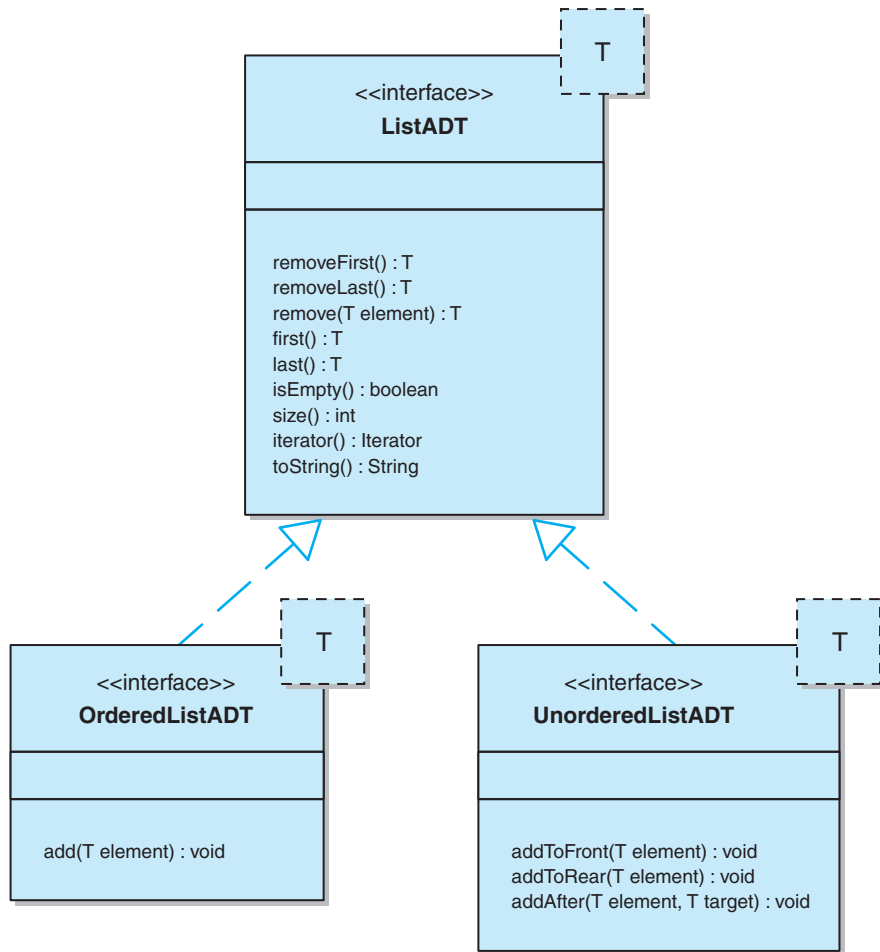


FIGURE 6.9 Using inheritance to define list interfaces

### LISTING 6.5

```
package jsjf;
import java.util.Iterator;
```

```
/**
 * ListADT defines the interface to a general list collection. Specific
 * types of lists will extend this interface to complete the
 * set of necessary operations.
 */
```

**LISTING 6.5***continued*

```
* @author Lewis and Chase
* @version 4.0
*/
public interface ListADT<T> extends Iterable<T>
{
 /**
 * Removes and returns the first element from this list.
 *
 * @return the first element from this list
 */
 public T removeFirst();

 /**
 * Removes and returns the last element from this list.
 *
 * @return the last element from this list
 */
 public T removeLast();

 /**
 * Removes and returns the specified element from this list.
 *
 * @param element the element to be removed from the list
 */
 public T remove(T element);

 /**
 * Returns a reference to the first element in this list.
 *
 * @return a reference to the first element in this list
 */
 public T first();

 /**
 * Returns a reference to the last element in this list.
 *
 * @return a reference to the last element in this list
 */
 public T last();
}
```



**LISTING 6.5***continued*

```
/**
 * Returns true if this list contains the specified target element.
 *
 * @param target the target that is being sought in the list
 * @return true if the list contains this element
 */
public boolean contains(T target);

/**
 * Returns true if this list contains no elements.
 *
 * @return true if this list contains no elements
 */
public boolean isEmpty();

/**
 * Returns the number of elements in this list.
 *
 * @return the integer representation of number of elements in this list
 */
public int size();

/**
 * Returns an iterator for the elements in this list.
 *
 * @return an iterator over the elements in this list
 */
public Iterator<T> iterator();

/**
 * Returns a string representation of this list.
 *
 * @return a string representation of this list
 */
public String toString();
}
```

**LISTING 6.6**

```
package jsjf;

/**
 * OrderedListADT defines the interface to an ordered list collection. Only
 * Comparable elements are stored, kept in the order determined by
 * the inherent relationship among the elements.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public interface OrderedListADT<T> extends ListADT<T>
{
 /**
 * Adds the specified element to this list at the proper location
 *
 * @param element the element to be added to this list
 */
 public void add(T element);
}
```

**LISTING 6.7**

```
package jsjf;

/**
 * UnorderedListADT defines the interface to an unordered list collection.
 * Elements are stored in any order the user desires.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public interface UnorderedListADT<T> extends ListADT<T>
{
 /**
 * Adds the specified element to the front of this list.
 *
 * @param element the element to be added to the front of this list
 */
 public void addToFront(T element);
}
```

## LISTING 6.7

*continued*

```

/**
 * Adds the specified element to the rear of this list.
 *
 * @param element the element to be added to the rear of this list
 */
public void addToRear(T element);

/**
 * Adds the specified element after the specified target.
 *
 * @param element the element to be added after the target
 * @param target the target is the item that the element will be added
 * after
 */
public void addAfter(T element, T target);
}

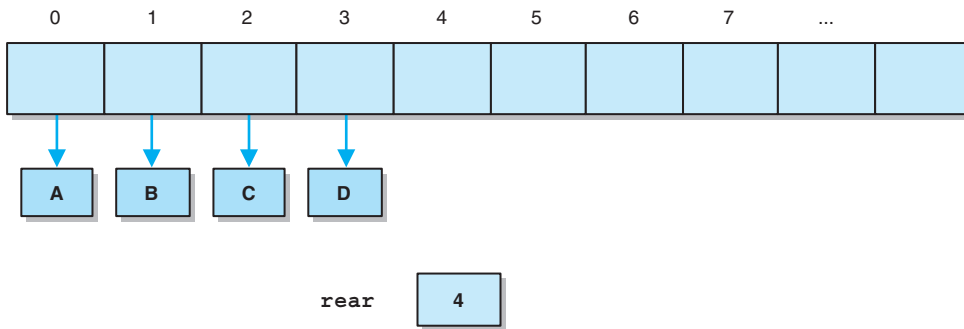
```

## 6.6 Implementing Lists with Arrays

As we've seen in previous chapters, an array-based implementation of a collection can fix one end of the list at index 0 and shift elements as needed. This is similar to our array-based implementation of a stack from Chapter 3. We dismissed that approach for queue in Chapter 5 because its operations `add` and `remove` elements from both ends. General lists also add and remove from either end, but they insert and remove in the middle of the list, as well. So shifting of elements cannot be avoided. A circular array approach could be used, but that will not eliminate the need to shift elements when adding or removing elements from the middle of the list.

Figure 6.10 depicts an array implementation of a list with the front of the list fixed at index 0. The integer variable `rear` represents the number of elements in the list and the next available slot for adding an element to the rear of the list.

Note that Figure 6.10 applies to both ordered and unordered lists. First we will explore the common operations. After Figure 6.10 we show the header and class-level data of the `ArrayList` class.



**FIGURE 6.10** An array implementation of a list

```

/**
 * ArrayList represents an array implementation of a list. The front of
 * the list is kept at array index 0. This class will be extended
 * to create a specific kind of list.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public abstract class ArrayList<T> implements ListADT<T>, Iterable<T>
{
 private final static int DEFAULT_CAPACITY = 100;
 private final static int NOT_FOUND = -1;
 protected int rear;
 protected T[] list;
 protected int modCount;

 /**
 * Creates an empty list using the default capacity.
 */
 public ArrayList()
 {
 this(DEFAULT_CAPACITY);
 }

 /**
 * Creates an empty list using the specified capacity.
 *
 * @param initialCapacity the size of the array list
 */
}

```

```

public ArrayList(int initialCapacity)
{
 rear = 0;
 list = (T[]) (new Object[initialCapacity]);
 modCount = 0;
}

```

The `ArrayList` class implements the `ListADT` interface defined earlier. It also implements the `Iterable` interface. That interface, and the `modCount` variable, are discussed in Chapter 7.

## The remove Operation

This variation of the `remove` operation requires that we search for the element passed in as a parameter and remove it from the list if it is found. Then, elements at higher indexes in the array are shifted down in the list to fill in the gap. Consider what happens if the element to be removed is the first element in the list. In this case, there is a single comparison to find the element, followed by  $n-1$  shifts to shift the elements down to fill the gap. On the opposite extreme, what happens if the element to be removed is the last element in the list? In this case, we would require  $n$  comparisons to find the element, and none of the remaining elements would need to be shifted. As it turns out, this implementation of the `remove` operation will always require exactly  $n$  comparisons and shifts, and thus the operation is  $O(n)$ . Note that if we were to use a circular array implementation, it would only improve the performance of the special case when the element to be removed is the first element. This operation can be implemented as follows:

```

/**
 * Removes and returns the specified element.
 *
 * @param element the element to be removed and returned from the list
 * @return the removed element
 * @throws ElementNotFoundException if the element is not in the list
 */
public T remove(T element)
{
 T result;
 int index = find(element);

```

```

 if (index == NOT_FOUND)
 throw new ElementNotFoundException("ArrayList");
 result = list[index];
 rear--;

 // shift the appropriate elements
 for (int scan=index; scan < rear; scan++)
 list[scan] = list[scan+1];
 list[rear] = null;
 modCount++;
 return result;
}

```

The `remove` method makes use of a method called `find`, which finds the element in question, if it exists in the list, and returns its index. The `find` method returns a constant called `NOT_FOUND` if the element is not in the list. The `NOT_FOUND` constant is equal to `-1` and is defined in the `ArrayList` class. If the element is not found, a `NoSuchElementException` is generated. If it is found, the elements at higher indexes are shifted down, the `rear` value is updated, and the element is returned.

The `find` method supports the implementation of a public operation on the list, rather than defining a new operation. Therefore, the `find` method is declared with private visibility. The `find` method can be implemented as follows:

```

/**
 * Returns the array index of the specified element, or the
 * constant NOT_FOUND if it is not found.
 *
 * @param target the target element
 * @return the index of the target element, or the
 * NOT_FOUND constant
 */
private int find(T target)
{
 int scan = 0;
 int result = NOT_FOUND;
 if (!isEmpty())
 while (result == NOT_FOUND && scan < rear)
 if (target.equals(list[scan]))
 result = scan;
 else
 scan++;
 return result;
}

```

Note that the `find` method relies on the `equals` method to determine whether the target has been found. It's possible that the object passed into the method is an exact copy of the element being sought. In fact, it may be an alias of the element in the list. However, if the parameter is a separate object, it may not contain all aspects of the element being sought. Only the key characteristics on which the `equals` method is based are important.

The logic of the `find` method could have been incorporated into the `remove` method, though it would have made the `remove` method somewhat complicated. When appropriate, such support methods should be defined to keep each method readable. Furthermore, in this case, the `find` support method is useful in implementing the `contains` operation, as we will now explore.

## DESIGN FOCUS

The overriding of the `equals` method and the implementation of the `Comparable` interface are excellent examples of the power of object-oriented design. We can create implementations of collections that can handle classes of objects that have not yet been designed as long as those objects provide a definition of equality and/or a method of comparison between objects of the class.

## DESIGN FOCUS

Separating out private methods such as the `find` method in the `ArrayList` class provides multiple benefits. First, it simplifies the definition of the already complex `remove` method. Second, it allows us to use the `find` method to implement the `contains` operation as well as the `addAfter` method for an `ArrayUnorderedList`. Notice that the `find` method does not throw an `ElementNotFound` exception. It simply returns a value (`-1`), signifying that the element was not found. In this way, the calling routine can decide how to handle the fact that the element was not found. In the `remove` method, that means throwing an exception. In the `contains` method, that means returning `false`.

## The contains Operation

The purpose of the `contains` operation is to determine whether a particular element is currently contained in the list. As we discussed, we can use the `find` support method to create a fairly straightforward implementation:

```
/**
 * Returns true if this list contains the specified element.
 *
 * @param target the target element
 * @return true if the target is in the list, false otherwise
 */
public boolean contains(T target)
{
 return (find(target) != NOT_FOUND);
}
```

If the target element is not found, the `contains` method returns false. If it is found, it returns true. A carefully constructed `return` statement ensures the proper return value. Because this method is performing a linear search of our list, our worst case will be that the element we are searching for is not in the list. This case would require  $n$  comparisons. We would expect this method to require, on average,  $n/2$  comparisons, which results in the operation being  $O(n)$ .

## The `add` Operation for an Ordered List

The `add` operation is the only way an element can be added to an ordered list. No location is specified in the call because the elements themselves determine their order. Very much like the `remove` operation, the `add` operation requires a combination of comparisons and shifts: comparisons to find the correct location in the list and then shifts to open a position for the new element. Looking at the two extremes, if the element to be added to the list belongs at the front of the list, that will require one comparison, and then the other  $n-1$  elements in the list will need to be shifted. If the element to be added belongs at the rear of the list, this will require  $n$  comparisons, and none of the other elements in the list will need to be shifted. Like the `remove` operation, the `add` operation requires  $n$  comparisons and shifts each time it is executed, and thus the operation is  $O(n)$ . The `add` operation can be implemented as follows:

```
/**
 * Adds the specified Comparable element to this list, keeping
 * the elements in sorted order.
 *
 * @param element the element to be added to the list
 */
```



```

public void add(T element)
{
 if (!(element instanceof Comparable))
 throw new NonComparableElementException("OrderedList");

 Comparable<T> comparableElement = (Comparable<T>)element;

 if (size() == list.length)
 expandCapacity();
 int scan = 0;

 // find the insertion location
 while (scan < rear && comparableElement.compareTo(list[scan]) > 0)
 scan++;

 // shift existing elements up one
 for (int shift=rear; shift > scan; shift--)
 list[shift] = list[shift-1];

 // insert element
 list[scan] = element;
 rear++;
 modCount++;
}

```

**KEY CONCEPT**

Only `Comparable` objects can be stored in an ordered list.

Note that only `Comparable` objects can be stored in an ordered list. If the element isn't `Comparable`, an exception is thrown. If it is `Comparable` but cannot be validly compared to the elements in the list, a `ClassCastException` will result, when the `compareTo` method is invoked.

Recall that the `Comparable` interface defines the `compareTo` method that returns a negative integer, zero, or positive integer value if the executing object is less than, equal to, or greater than the parameter, respectively.

The unordered and indexed versions of a list do not require that the elements they store be `Comparable`. It is a testament to the utility of object-oriented programming that the various classes that implement these list variations can exist in harmony despite these differences.

## Operations Particular to Unordered Lists

The `addToFront` and `addToRear` operations are similar to operations from other collections and are therefore left as programming projects. Keep in mind that the `addToFront` operation must shift the current elements in the list first, to make room at index 0 for the new element. Thus we know that the `addToFront` operation will be  $O(n)$  because it requires  $n-1$  elements to be shifted. Like the push operation on a stack, the `addToRear` operation will be  $O(1)$ .

## The `addAfter` Operation for an Unordered List

The `addAfter` operation accepts two parameters: one that represents the element to be added and one that represents the target element that determines the placement of the new element. The `addAfter` method must first find the target element, shift the elements at higher indexes to make room, and then insert the new element after it. Very much like the `remove` operation and the `add` operation for ordered lists, the `addAfter` method requires a combination of  $n$  comparisons and shifts and will be  $O(n)$ .

```
/**
 * Adds the specified element after the specified target element.
 * Throws an ElementNotFoundException if the target is not found.
 *
 * @param element the element to be added after the target element
 * @param target the target that the element is to be added after
 */
public void addAfter(T element, T target)
{
 if (size() == list.length)
 expandCapacity();

 int scan = 0;

 // find the insertion point
 while (scan < rear && !target.equals(list[scan]))
 scan++;

 if (scan == rear)
 throw new ElementNotFoundException("UnorderedList");

 scan++;

 // shift elements up one
```

```

 for (int shift=rear; shift > scan; shift--)
 list[shift] = list[shift-1];

 // insert element

 list[scan] = element;
 rear++;
 modCount++;
}

```

## 6.7 Implementing Lists with Links

As we have seen with other collections, the use of a linked list is often another convenient way to implement a linear collection. The common operations that apply for ordered and unordered lists, as well as the particular operations for each type, can be implemented with techniques similar to the ones we have used before. We will examine a couple of the more interesting operations but will leave most of them as programming projects.

First, the class header, class-level data, and constructor for our `LinkedList` class are provided for context:

```

/**
 * LinkedList represents a linked implementation of a list.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public abstract class LinkedList<T> implements ListADT<T>, Iterable<T>
{
 protected int count;
 protected LinearNode<T> head, tail;
 protected int modCount;

 /**
 * Creates an empty list.
 */
 public LinkedList()
 {
 count = 0;
 head = tail = null;
 modCount = 0;
 }
}

```

## The remove Operation

The `remove` operation is part of the `LinkedList` class shared by both implementations: unordered and ordered lists. The `remove` operation consists of making sure that the list is not empty, finding the element to be removed, and then handling one of four cases: the element to be removed is the only element in the list, the element to be removed is the first element in the list, the element to be removed is the last element in the list, or the element to be removed is in the middle of the list. In all cases, the `count` is decremented by one. Unlike the `remove` operation for the array version, the linked version does not require elements to be shifted to close the gap. However, given that the worst case still requires  $n$  comparisons to determine that the target element is not in the list, the `remove` operation is still  $O(n)$ . An implementation of the `remove` operation follows.

```
/**
 * Removes the first instance of the specified element from this
 * list and returns it. Throws an EmptyCollectionException
 * if the list is empty. Throws a ElementNotFoundException if the
 * specified element is not found in the list.
 *
 * @param targetElement the element to be removed from the list
 * @return a reference to the removed element
 * @throws EmptyCollectionException if the list is empty
 * @throws ElementNotFoundException if the target element is not found
 */
public T remove(T targetElement) throws EmptyCollectionException,
 ElementNotFoundException
{
 if (isEmpty())
 throw new EmptyCollectionException("LinkedList");

 boolean found = false;
 LinearNode<T> previous = null;
 LinearNode<T> current = head;

 while (current != null && !found)
 if (targetElement.equals(current.getElement()))
 found = true;
 else
 {
 previous = current;
 current = current.getNext();
 }
}
```

```
 if (!found)
 throw new ElementNotFoundException("LinkedList");

 if (size() == 1) // only one element in the list
 head = tail = null;
 else if (current.equals(head)) // target is at the head
 head = current.getNext();
 else if (current.equals(tail)) // target is at the tail
 {
 tail = previous;
 tail.setNext(null);
 }
 else // target is in the middle
 previous.setNext(current.getNext());

 count--;
 modCount++;

 return current.getElement();
}
```

## Summary of Key Concepts

- List collections can be categorized as ordered, unordered, or indexed.
- The elements of an ordered list have an inherent relationship defining their order.
- The elements of an unordered list are kept in whatever order the client chooses.
- An indexed list maintains a contiguous numeric index range for its elements.
- The Java API does not provide a class that implements an ordered list.
- Many common operations can be defined for all list types. The differences between them stem from how elements are added.
- Interfaces can be used to derive other interfaces. The child interface contains all abstract methods of the parent.
- An interface name can be used to declare an object reference variable. An interface reference can refer to any object of any class that implements the interface.
- Interfaces enable us to make polymorphic references in which the method that is invoked is based on the particular object being referenced at the time.
- The Josephus problem is a classic computing problem that is appropriately solved with indexed lists.
- Only `Comparable` objects can be stored in an ordered list.

## Summary of Terms

**indexed list** A list whose elements can be referenced using a numeric index.

**Josephus problem** A classic computing problem whose goal is to find the order in which elements are selected from a list by taking every  $i^{\text{th}}$  element cyclically until none remains.

**natural ordering** An expression of the ordering criteria used to determine whether one object comes before another, often implemented using the `compareTo` method.

**ordered list** A list whose elements are ordered in terms of some inherent characteristic of the elements.

**serialization** A technique for representing an object as a stream of binary digits, which allows objects to be read and written from files with their state maintained.

**unordered list** A list whose elements have no inherent order but are ordered by their placement in the list.

## Self-Review Questions

- SR 6.1 Define: an ordered list, an unordered list, and an indexed list.
- SR 6.2 What is the primary difference between an indexed list and an array?
- SR 6.3 What is the purpose of the `set` and `size` methods that are part of the Java Collections API framework?
- SR 6.4 Why do the `ArrayList` and `LinkedList` classes implement the `Serializable` interface?
- SR 6.5 Why does an ordered list require that the elements they store be `Comparable`?
- SR 6.6 An array-based list of integers has 16 elements stored in it. What would be the value of the variables `front` and `rear`?
- SR 6.7 Why is the time to increase the capacity of the array on an `add` operation considered negligible for the `ArrayList` implementation?
- SR 6.8 Why is a circular array implementation not as attractive as an implementation for a list as it was for a queue?

## Exercises

- EX 6.1 Hand trace an ordered list `x` through the following operations.

```
X.add(new Integer(20));
X.add(new Integer(35));
Object Y = X.first();
X.add(new Integer(15));
X.add(new Integer(10));
X.add(new Integer(25));
Object Y = X.removeLast();
Object Y = X.remove(new Integer(35));
X.add(new Integer(45));
```

- EX 6.2 Given the resulting list `x` from Exercise 6.1, what would be the result of each of the following?

```
a. X.first();
b. z = X.contains(new Integer(15));
 X.first();
c. Y = X.remove(new Integer(10));
 X.first();
```

- EX 6.3 Consider the class `ProgramOfStudy` discussed in the chapter (Listing 6.2). Write a method `addCourseBefore` that would add the specified course before the target course.

- EX 6.4 What would happen if a programmer erroneously changes the code snippet `"list.add(targetIndex + 1, newCourse);"` to `"list.set(targetIndex + 1, newCourse);"`?
- EX 6.5 Compare and contrast ordered, unordered and indexed lists in terms of adding elements in them.
- EX 6.6 Hand trace an unordered list through the following operations.
- ```
X.addToFront(new Integer(20));
X.addToRear(new Integer(35));
Object Y = X.first();
X.addAfter(new Integer(15), new Integer(20));
X.addToFront(new Integer(10));
X.addToRear(new Integer(25));
Object Y = X.removeLast();
Object Y = X.remove(new Integer(35));
X.addAfter(new Integer(45), new Integer(15));
```
- EX 6.7 Discuss the benefits provided by the `private` method `find` in the `ArrayList` class. What is the logic behind keeping this method as `private`?

Programming Projects

- PP 6.1 Implement a stack using a `LinkedList` object to store the stack elements.
- PP 6.2 Implement a stack using an `ArrayList` object to store the stack elements.
- PP 6.3 Implement a queue using a `LinkedList` object to store the queue elements.
- PP 6.4 Implement a queue using an `ArrayList` object to store the queue elements.
- PP 6.5 Implement the Josephus problem using a queue, and compare the performance of that algorithm to the `ArrayList` implementation from this chapter.
- PP 6.6 Implement an `OrderedList` using a `LinkedList` object to store the list elements.
- PP 6.7 Implement an `OrderedList` using an `ArrayList` object to store the list elements.
- PP 6.8 Complete the implementation of the `ArrayList` class.
- PP 6.9 Complete the implementation of the `ArrayOrderedList` class.

- PP 6.10 Complete the implementation of the `ArrayUnorderedList` class.
- PP 6.11 Write an implementation of the `LinkedList` class.
- PP 6.12 Write an implementation of the `LinkedListOrderedList` class.
- PP 6.13 Write an implementation of the `LinkedListUnorderedList` class.
- PP 6.14 Create an implementation of a doubly linked `DoubleOrderedList` class. You will need to create a `DoubleNode` class, a `DoubleList` class, and a `DoubleIterator` class.
- PP 6.15 Create a graphical application that provides a button for `add` and `remove` from an ordered list, a text field to accept a string as input for `add`, and a text area to display the contents of the list after each operation.
- PP 6.16 Create a graphical application that provides a button for `addToFront`, `addToRear`, `addAfter`, and `remove` from an unordered list. Your application must provide a text field to accept a string as input for any of the `add` operations. The user should be able to select the element to be added after, and to select the element to be removed.
- PP 6.17 Modify the `Course` class from this chapter so that it implements the `Comparable` interface. Order the courses first by department and then by course number. Then write a program that uses an ordered list to maintain a list of courses.

Answers to Self-Review Questions

- SRA 6.1 An ordered list is a collection of objects ordered by value. An unordered list is a collection of objects with no inherent order. An indexed list is a collection of objects with no inherent order that are ordered by index value.
- SRA 6.2 An indexed list keeps its indexes contiguous. If an element is removed, the positions of other elements “collapse” to eliminate the gap. When an element is inserted, the indexes of other elements are shifted to make room. The complexity of the algorithm remains $O(1)$. In an array, the shifting of elements is required, which increases the complexity of the algorithm to $O(n)$.
- SRA 6.3 The `set` method replaces the element at the specified index and the `size` method returns the number of elements in the list.
- SRA 6.4 In order for an object to be saved using serialization, its class must implement `Serializable`. There are no methods in the

`Serializable` interface—it is used simply to indicate that the object may be converted to a serialized representation.

Serialization is a technique for representing an object as a stream of binary digits, which allows objects to be read and written from files with their state maintained. Therefore, the `ArrayList` and `LinkedList` classes implement the `Serializable` interface to enable its objects to be saved in a file and retrieved later, as a stream of binary digits.

- SRA 6.5 An ordered list is a list whose elements are ordered in terms of some inherent characteristic of the elements. The `Comparable` interface has a `compareTo` method, implementing which imparts the ability of natural ordering of the objects. In other words, it sets an ordering criterion that is used to determine whether one object comes before another.
- SRA 6.6 The variable `front` would store the value 0 and the variable `rear` would store the value 16.
- SRA 6.7 Averaged over the total number of insertions into the list, the time to enlarge the array has little effect on the total time.
- SRA 6.8 The circular array implementation of a queue improved the efficiency of the `dequeue` operation from $O(n)$ to $O(1)$ because it eliminated the need to shift elements in the array. That is not the case for a list, because we can add or remove elements anywhere in the list, not just at the front or the rear.

This page is intentionally left blank.



Iterators

7

We mentioned iterators in Chapter 6 in our discussion of lists, but didn't explore them in any detail. They are important enough to deserve their own chapter. Conceptually, they provide a standard way to access each element of a collection in turn, which is a common operation. And their implementation in the Java API has some interesting nuances that are worth exploring carefully.

CHAPTER OBJECTIVES

- Define an iterator and explore its use.
- Discuss the `Iterator` and `Iterable` interfaces.
- Explore the concept of fail-fast collections.
- Use iterators in various situations.
- Explore implementation options related to iterators.

7.1 What's an Iterator?

KEY CONCEPT

An iterator is an object that provides a way to access each element in a collection in turn.

An *iterator* is an object that allows the user to acquire and use each element in a collection one at a time. It works in conjunction with a collection but is a separate object. An iterator is a mechanism for helping implement a collection.

Embracing the concept of an iterator consistently over the implementation of multiple collections makes it much easier to process and manage those collections and the elements they contain. The Java API has a consistent approach to iterators that are implemented by nearly all collections in the class library. We will follow this approach in our own implementations.

Iterators are implemented in the Java API using two primary interfaces:

- `Iterator` – used to define an object that can be used as an iterator.
- `Iterable` – used to define a collection from which an iterator can be extracted.

KEY CONCEPT

A collection is often defined as `Iterable`, which means it provides an `Iterator` when needed.

A collection is `Iterable`, which commits it to providing an `Iterator` when requested. For example, a `LinkedList` is `Iterable`, which means it provides a method called `iterator` that can be called to get an iterator over of the elements in the list. The names of the interfaces make it fairly easy to keep them straight.

The abstract methods defined in these two interfaces are shown in Figures 7.1 and 7.2. Both interfaces operate on a generic type, which is denoted by `E` in these figures.

The `Iterable` interface has only one method, called `iterator`, that returns an `Iterator` object. When you create the collection, you commit to the element type, which is used to define the elements in the iterator.

Method	Description
<code>boolean hasNext()</code>	Returns true if the iteration has more elements.
<code>E next()</code>	Returns the next element in the iteration.
<code>void remove()</code>	Removes the last element returned by the iteration from the underlying collection.

FIGURE 7.1 The methods in the `Iterator` interface

Method	Description
<code>Iterator<E> iterator()</code>	Returns an iterator over a set of elements of type <code>E</code> .

FIGURE 7.2 The methods in the `Iterable` interface

The `Iterator` interface contains three methods. The first two, `hasNext` and `next`, can be used in concert to access the elements in turn. For example, if `myList` is an `ArrayList` of `Book` objects, you could use the following code to print all books in the list.

```
Iterator<Book> itr = myList.iterator();
while (itr.hasNext())
    System.out.println(itr.next());
```

In this example, the first line calls the `iterator` method of the collection to obtain the `Iterator<Book>` object. Then a call to the `hasNext` method of the iterator is used as the condition of the `while` loop. Inside the loop, the `next` method of the iterator is called to get the next book. When the iteration is exhausted, the loop terminates.

The `remove` operation of the `Iterator` interface is provided as a convenience to allow you to remove an element from a collection while iterating over it. The `remove` method is considered an optional operation, and not all iterators implement it.

Now, you've probably realized that you could access the elements of a collection using a `for-each` loop as we've done in the past. The following code does the same thing that the previous `while` loop accomplishes.

```
for (Book book : myList)
    System.out.println(book);
```

The `for-each` code is cleaner and shorter than the `while` loop code, and it will often be the technique you'll want to use. But you should be aware that both of these examples are using iterators. Java provides the `for-each` construct specifically to simplify the processing of iterators. Behind the scenes, the `for-each` code is translated into code that explicitly calls the iterator methods.

In fact, you can use a `for-each` loop only on an `Iterable` collection. Most of the collections in the Java API are `Iterable`, and you can define your own collection objects to be `Iterable` as well.

So why would you ever use an explicit iterator with a `while` loop instead of the cleaner `for-each` loop? Well, there are two basic reasons. First, you may not want to process all elements in the iteration. If you're looking for a particular element, for example, and do not wish to process them all, you may choose to use an explicit iterator. (You could break out of the loop, but that may not be as clean.)

You may also choose to use an explicit iterator if you want to call the iterator's `remove` method. The `for-each` loop does not provide explicit access to the iterator, so the only way you could do it would be to call the `remove` method of the collection, and that would cause a completely separate traversal of the collection data structure in order to reach the element (again) to remove it.

KEY CONCEPT

The optional `remove` method of an iterator makes it possible to remove an element without having to traverse the collection again.

Other Iterator Issues

We should note a couple of other issues related to iterators before we continue. First, there is no assumption about the order in which an `Iterator` object delivers the elements from the collection. In the case of a list, there is a linear order to the elements, so the iterator would probably follow that order. In other cases, an iterator may follow a different order that makes sense for that collection and its underlying data structures. Read the API documentation carefully before making any assumptions about how an iterator delivers its elements.

KEY CONCEPT

You should make no assumptions about the order in which an iterator delivers elements unless it is explicitly stated.

Second, you should be aware that there is an intimate relationship between an iterator and its collection. An iterator references elements that are still stored in the collection. Therefore, while an iterator is in use, there are at least two objects with references to the element objects. Because of this relationship, the structure of the underlying collection must not be modified while an iterator on that collection is actively being used.

Embracing this assumption, most of the iterators provided by collections in the Java API are implemented to be *fail-fast*, which means that they will throw a `ConcurrentModificationException` if the collection is modified while an iterator is active. The idea is that the iterator will fail quickly and cleanly, rather than permitting a problem to be introduced that won't be discovered until some unknown point in the future.

KEY CONCEPT

Most iterators are fail-fast and will throw an exception if the collection is modified while an iterator is active.

7.2 Using Iterators: Program of Study Revisited

In Chapter 6 we examined a program that created a program of study for a student, consisting of a list of the courses the student has taken and is planning to take. Recall that a `Course` object stores course information such as the number and title, as well as the grade the student received if she or he has already taken the course.

The `ProgramOfStudy` class maintains an unordered list of `Course` objects. In Chapter 6 we examined various aspects of this class. Now we will focus on aspects of it that pertain to iterators. The `ProgramOfStudy` class is reprinted in Listing 7.1 for convenience.

Note first that the `ProgramOfStudy` class implements the `Iterable` interface using the `Course` class as the generic type. As discussed in the previous section, that commits this class to implementing the `iterator` method, which returns an `Iterator` object for the program of study. In this implementation, the `iterator` method simply returns the `Iterator` object obtained from the `LinkedList` object that stores the courses.

Thus a `ProgramOfStudy` object is `Iterable`, and the `LinkedList` it uses to store the `Course` objects is `Iterable` as well. We'll see both in use.

LISTING 7.1

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

/**
 * Represents a Program of Study, a list of courses taken and planned, for an
 * individual student.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class ProgramOfStudy implements Iterable<Course>, Serializable
{
    private List<Course> list;

    /**
     * Constructs an initially empty Program of Study.
     */
    public ProgramOfStudy()
    {
        list = new LinkedList<Course>();
    }

    /**
     * Adds the specified course to the end of the course list.
     *
     * @param course the course to add
     */
    public void addCourse(Course course)
    {
        if (course != null)
            list.add(course);
    }

    /**
     * Finds and returns the course matching the specified prefix and number.
     *
     * @param prefix the prefix of the target course
     */
}
```


LISTING 7.1 *continued*

```

    * @param number the number of the target course
    * @return the course, or null if not found
    */
public Course find(String prefix, int number)
{
    for (Course course : list)
        if (prefix.equals(course.getPrefix()) &&
            number == course.getNumber())
            return course;

    return null;
}

/**
 * Adds the specified course after the target course. Does nothing if
 * either course is null or if the target is not found.
 *
 * @param target the course after which the new course will be added
 * @param newCourse the course to add
 */
public void addCourseAfter(Course target, Course newCourse)
{
    if (target == null || newCourse == null)
        return;

    int targetIndex = list.indexOf(target);
    if (targetIndex != -1)
        list.add(targetIndex + 1, newCourse);
}

/**
 * Replaces the specified target course with the new course. Does nothing if
 * either course is null or if the target is not found.
 *
 * @param target the course to be replaced
 * @param newCourse the new course to add
 */
public void replace(Course target, Course newCourse)
{
    if (target == null || newCourse == null)
        return;

    int targetIndex = list.indexOf(target);
    if (targetIndex != -1)
        list.set(targetIndex, newCourse);
}

```

LISTING 7.1

continued

```
/**
 * Creates and returns a string representation of this Program of Study.
 *
 * @return a string representation of the Program of Study
 */
public String toString()
{
    String result = "";
    for (Course course : list)
        result += course + "\n";
    return result;
}

/**
 * Returns an iterator for this Program of Study.
 *
 * @return an iterator for the Program of Study
 */
public Iterator<Course> iterator()
{
    return list.iterator();
}

/**
 * Saves a serialized version of this Program of Study to the specified
 * file name.
 *
 * @param fileName the file name under which the POS will be stored
 * @throws IOException
 */
public void save(String fileName) throws IOException
{
    FileOutputStream fos = new FileOutputStream(fileName);
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(this);
    oos.flush();
    oos.close();
}

/**
 * Loads a serialized Program of Study from the specified file.
 *
 * @param fileName the file from which the POS is read
 * @return the loaded Program of Study
 * @throws IOException
 * @throws ClassNotFoundException
 */
```

LISTING 7.1 *continued*

```

public static ProgramOfStudy load(String fileName) throws IOException,
    ClassNotFoundException
{
    FileInputStream fis = new FileInputStream(fileName);
    ObjectInputStream ois = new ObjectInputStream(fis);
    ProgramOfStudy pos = (ProgramOfStudy) ois.readObject();
    ois.close();

    return pos;
}

```

Consider the `toString` method in the `ProgramOfStudy` class. It uses a for-each loop on the linked list to scan through the list and append the description of each course to the overall description. It can do this only because the `LinkedList` class is `Iterable`.

The `find` method of `ProgramOfStudy` is similar in that it uses a for-each loop to scan through the list of `Course` objects. In this case, however, the `return` statement jumps out of the loop (and the method) as soon as the target course is found.

Printing Certain Courses

Now let's examine a driver program that exercises our program of study in a new way. Listing 7.2 contains a `main` method that first reads a previously created `ProgramOfStudy` object stored in a file. (Recall that the `ProgramOfStudy` class uses serialization to store the list of courses.) Then, after printing the entire list, it prints only those courses that have been taken and in which the student received a grade of A or A-.

Note that a for-each loop is used to examine each course and print only those with high grades. That loop iterates over the `ProgramOfStudy` object called `pos`. This is possible only because the `ProgramOfStudy` class is `Iterable`.

LISTING 7.2

```

import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

```

LISTING 7.2 *continued*

```
/**
 * Demonstrates the use of an Iterable object (and the technique for reading
 * a serialized object from a file).
 *
 * @author Lewis and Chase
 */
public class POSGrades
{
    /**
     * Reads a serialized Program of Study, then prints all courses in which
     * a grade of A or A- was earned.
     */
    public static void main(String[] args) throws Exception
    {
        ProgramOfStudy pos = ProgramOfStudy.load("ProgramOfStudy");

        System.out.println(pos);

        System.out.println("Classes with Grades of A or A-\n");

        for (Course course : pos)
        {
            if (course.getGrade().equals("A") || course.getGrade().equals("A-"))
                System.out.println(course);
        }
    }
}
```

Removing Courses

Listing 7.3 contains yet another driver program. This example removes, from a program of study, any course that doesn't already have a grade. After an existing `ProgramOfStudy` object is read from a file and printed, each course is examined in turn, and if it has no grade, it is removed from the list.

This time, however, a for-each loop is not used to iterate over the `Course` objects. Instead, the `iterator` method of the `ProgramOfStudy` object is called explicitly, which returns an `Iterator` object. Then, using the `hasNext` and `next` methods of the iterator, a while loop is used to iterate over the courses. An explicit iterator is used in this case because of the `remove` operation. To remove a `Course` object, we call the `remove` method of the iterator. If we had done this in a for-each loop, we would have triggered a `ConcurrentModificationException`, as discussed in the first section of this chapter.

LISTING 7.3

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Iterator;

/**
 * Demonstrates the use of an explicit iterator.
 *
 * @author Lewis and Chase
 */
public class POSClear
{
    /**
     * Reads a serialized Program of Study, then removes all courses that
     * don't have a grade.
     */
    public static void main(String[] args) throws Exception
    {
        ProgramOfStudy pos = ProgramOfStudy.load("ProgramOfStudy");

        System.out.println(pos);

        System.out.println("Removing courses with no grades.\n");

        Iterator<Course> itr = pos.iterator();
        while (itr.hasNext())
        {
            Course course = itr.next();
            if (!course.taken())
                itr.remove();
        }

        System.out.println(pos);

        pos.save("ProgramOfStudy");
    }
}
```

7.3 Implementing Iterators: With Arrays

In Chapter 6 we explored the implementation of an array-based list. One thing we didn't show then was the implementation of the iterator for our own `ArrayList` class. Let's explore it now.

Listing 7.4 contains the `ArrayListIterator` class. It's defined as a private class, and therefore would actually be an inner class, part of the `ArrayList` class from Chapter 6. This is an appropriate use for an inner class, which has an intimate relationship with its outer class.

The `ArrayListIterator` class maintains two integers: one for the index of the current element in the iteration, and one to keep track of the number of modifications made through the iterator. The constructor sets `current` to 0 (the first element in the array) and sets the `iteratorModCount` to be equal to the `modCount` of the collection itself.

The `modCount` variable is an integer defined in the outer `ArrayList` class. If you go back to Chapter 6, you'll see that anytime the collection was modified (such as something being added to the collection), the `modCount` was incremented. So when a new iterator is created, its modification count is set equal to the count of the collection itself. If those two values get out of synch (because the collection was updated), then the iterator will throw a `ConcurrentModificationException`.

The `hasNext` method checks the modification count and then returns true if there are still elements to process, which in this case is true if the `current` iterator index is less than the `rear` counter. Recall that the `rear` counter is maintained by the outer collection class.

KEY CONCEPT

An iterator class is often implemented as an inner class of the collection to which it belongs.

KEY CONCEPT

An iterator checks the modification count to ensure that it stays consistent with the modification count from the collection when it was created.

LISTING 7.4

```
/**
 * ArrayListIterator iterator over the elements of an ArrayList.
 */
private class ArrayListIterator implements Iterator
{
    int iteratorModCount;
    int current;

    /**
     * Sets up this iterator using the specified modCount.
     */
}
```

LISTING 7.4 *continued*

```

    * @param modCount the current modification count for the ArrayList
    */
public ArrayListIterator()
{
    iteratorModCount = modCount;
    current = 0;
}

/**
 * Returns true if this iterator has at least one more element
 * to deliver in the iteration.
 *
 * @return true if this iterator has at least one more element to deliver
 *         in the iteration
 * @throws ConcurrentModificationException if the collection has changed
 *         while the iterator is in use
 */
public boolean hasNext() throws ConcurrentModificationException
{
    if (iteratorModCount != modCount)
        throw new ConcurrentModificationException();

    return (current < rear);
}

/**
 * Returns the next element in the iteration. If there are no
 * more elements in this iteration, a NoSuchElementException is
 * thrown.
 *
 * @return the next element in the iteration
 * @throws NoSuchElementException if an element not found exception occurs
 * @throws ConcurrentModificationException if the collection has changed
 */
public T next() throws ConcurrentModificationException
{
    if (!hasNext())
        throw new NoSuchElementException();

    current++;

    return list[current - 1];
}

```

LISTING 7.4 *continued*

```
/**
 * The remove operation is not supported in this collection.
 *
 * @throws UnsupportedOperationException if the remove method is called
 */
public void remove() throws UnsupportedOperationException
{
    throw new UnsupportedOperationException();
}
```

The next method returns the next element in the iteration and increments the current index value. If the next method is invoked and there are no elements left to process, then a `NoSuchElementException` is thrown.

In this implementation of the iterator, the remove operation is not supported (remember, it's considered optional). If this method is called, then an `UnsupportedOperationException` is thrown.

7.4 Implementing Iterators: With Links

Similarly, an iterator for a collection using links can also be defined. Like the `ArrayListIterator` class, the `LinkedListIterator` class (see Listing 7.5) is implemented as a private inner class. The `LinkedList` outer class maintains its own `modCount` that must stay in synch with the iterator's stored value.

In this iterator, though, the value of `current` is a pointer to a `LinearNode` instead of an integer index value. The `hasNext` method, therefore, simply confirms that `current` is pointing to a valid node. The `next` method returns the element at the current node and moves the `current` reference to the next node. As with our `ArrayListIterator`, the remove method is not supported.

LISTING 7.5

```
/**
 * LinkedListIterator represents an iterator for a linked list of linear nodes.
 */
private class LinkedListIterator implements Iterator<T>
{
```


LISTING 7.5 *continued*

```

private int iteratorModCount; // the number of elements in the collection
private LinearNode<T> current; // the current position

/**
 * Sets up this iterator using the specified items.
 *
 * @param collection the collection the iterator will move over
 * @param size       the integer size of the collection
 */
public LinkedListIterator()
{
    current = head;
    iteratorModCount = modCount;
}

/**
 * Returns true if this iterator has at least one more element
 * to deliver in the iteration.
 *
 * @return true if this iterator has at least one more element to deliver
 *         in the iteration
 * @throws ConcurrentModificationException if the collection has changed
 *         while the iterator is in use
 */
public boolean hasNext() throws ConcurrentModificationException
{
    if (iteratorModCount != modCount)
        throw new ConcurrentModificationException();

    return (current != null);
}

/**
 * Returns the next element in the iteration. If there are no
 * more elements in this iteration, a NoSuchElementException is
 * thrown.
 *
 * @return the next element in the iteration
 * @throws NoSuchElementException if the iterator is empty
 */
public T next() throws ConcurrentModificationException
{
    if (!hasNext())
        throw new NoSuchElementException();
}

```

LISTING 7.5 *continued*

```
        T result = current.getElement();
        current = current.getNext();
        return result;
    }

    /**
     * The remove operation is not supported.
     *
     * @throws UnsupportedOperationException if the remove operation is called
     */
    public void remove() throws UnsupportedOperationException
    {
        throw new UnsupportedOperationException();
    }
}
```

Summary of Key Concepts

- An iterator is an object that provides a way to access each element in a collection in turn.
- A collection is often defined as `Iterable`, which means it provides an `Iterator` when needed.
- The optional `remove` method of an iterator makes it possible to remove an element without having to traverse the collection again.
- Most iterators are fail-fast and will throw an exception if the collection is modified while an iterator is active.
- You should make no assumptions about the order in which an iterator delivers elements unless it is explicitly stated.
- An iterator class is often implemented as an inner class of the collection to which it belongs.
- An iterator checks the modification count to ensure that it stays consistent with the modification count from the collection when it was created.

Summary of Terms

iterator An object that allows the user to acquire and use each element in the collection one at a time.

fail-fast An iterator that throws an exception if its collection is modified in any way except through the iterator itself.

Self-Review Questions

- SR 7.1 What is an iterator?
- SR 7.2 What does the `Iterable` interface represent?
- SR 7.3 What does the `Iterator` interface represent?
- SR 7.4 What is the relationship between a for-each loop and iterators?
- SR 7.5 Why might you need to use an explicit iterator instead of a for-each loop?
- SR 7.6 What does it mean for an iterator to be fail-fast?
- SR 7.7 How is the fail-fast characteristic implemented?

Exercises

- EX 7.1 Write a for-each loop that prints all elements in a collection of `Item` objects called `inventory`. What is required for that loop to work?
- EX 7.2 Write a while loop that uses an explicit iterator to accomplish the same thing as Exercise 7.1.
- EX 7.3 Write a for-each loop that calls the `addBonus` method on each `SalaryAccount` object in a collection called `payroll`. What is required for that loop to work?
- EX 7.4 Write a while loop that uses an explicit iterator to accomplish the same thing as Exercise 7.3.

Answers to Self-Review Questions

- SRA 7.1 An iterator is an object that is used to process each element in a collection one at a time.
- SRA 7.2 The `Iterable` interface is implemented by a collection to formally commit to providing an iterator when it is needed.
- SRA 7.3 The `Iterator` interface is implemented by an interface and provides methods for checking for, accessing, and removing elements.
- SRA 7.4 A for-each loop can be used only with collections that implement the `Iterable` interface. It is a syntactic simplification that can also be accomplished using an iterator explicitly.
- SRA 7.5 You may need to use an explicit iterator rather than a for-each loop if you don't plan on processing all elements in a collection or if you may use the iterator's `remove` method.
- SRA 7.6 A fail-fast iterator will fail quickly and cleanly if the underlying collection has been modified by something other than the iterator itself.
- SRA 7.7 An iterator notes the modification count of the collection when it is created and, on subsequent operations, makes sure that that value hasn't changed. If it has, the iterator throws a `ConcurrentModificationException`.

This page is intentionally left blank.



Recursion

8

Recursion is a powerful programming technique that provides elegant solutions to certain problems. It is particularly helpful in the implementation of various data structures and in the process of searching and sorting data. This chapter provides an introduction to recursive processing. It contains an explanation of the basic concepts underlying recursion and then explores the use of recursion in programming.

CHAPTER OBJECTIVES

- Explain the underlying concepts of recursion.
- Examine recursive methods and unravel their processing steps.
- Define *infinite recursion* and discuss ways to avoid it.
- Explain when recursion should and should not be used.
- Demonstrate the use of recursion to solve problems.

8.1 Recursive Thinking

We know that one method can call another method to help it accomplish its goal. Similarly, a method can also call itself to help accomplish its goal. *Recursion* is a programming technique in which a method calls itself to fulfill its overall purpose.

KEY CONCEPT

Recursion is a programming technique in which a method calls itself. A key to being able to program recursively is to be able to think recursively.

Before we get into the details of how we use recursion in a program, we need to explore the general concept of recursion. The ability to think recursively is essential to being able to use recursion as a programming technique.

In general, recursion is the process of defining something in terms of itself. For example, consider the following definition of the word *decoration*:

decoration: n. any ornament or adornment used to decorate something

The word *decorate* is used to define the word *decoration*. You may recall your grade-school teacher telling you to avoid such recursive definitions when explaining the meaning of a word. However, in many situations, recursion is an appropriate way to express an idea or definition. For example, suppose we want to formally define a list of one or more numbers, separated by commas. Such a list can be defined recursively either as a number or as a number followed by a comma followed by a list. This definition can be expressed as follows:

A list is a: number
or a: number comma list

This recursive definition of a list defines each of the following lists of numbers:

24, 88, 40, 37
96, 43
14, 64, 21, 69, 32, 93, 47, 81, 28, 45, 81, 52, 69
70

No matter how long a list is, the recursive definition describes it. A list of one element, such as in the last example, is defined completely by the first (nonrecursive) part of the definition. For any list longer than one element, the recursive part of the definition (the part that refers to itself) is used as many times as necessary, until the last element is reached. The last element in the list is always defined by the nonrecursive part of this definition. Figure 8.1 shows how one particular list of numbers corresponds to the recursive definition of *list*.

Infinite Recursion

Note that this definition of a list contains one option that is recursive and one option that is not. The part of the definition that is not recursive is called the *base*

case. If all options had a recursive component, then the recursion would never end. For example, if the definition of a list were simply “a number followed by a comma followed by a list,” then no list could ever end. This problem is called *infinite recursion*. It is similar to an infinite loop, except that the “loop” occurs in the definition itself.

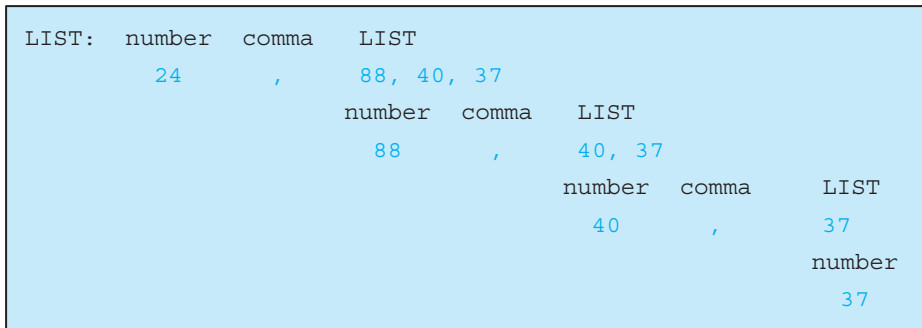


FIGURE 8.1 Tracing the recursive definition of a list

As in the infinite loop problem, a programmer must be careful to design algorithms so that they avoid infinite recursion. Any recursive definition must have a base case that does not result in a recursive option. The *base case* of the list definition is a single number that is not followed by anything. In other words, when the last number in the list is reached, the base case option terminates the recursive path.

KEY CONCEPT

Any recursive definition must have a nonrecursive part, called the base case, that permits the recursion to eventually end.

Recursion in Math

Let’s look at an example of recursion in mathematics. The value referred to as $N!$ (which is pronounced *N factorial*) is defined for any positive integer N as the product of all integers between 1 and N , inclusive. Therefore,

$$3! = 3 * 2 * 1 = 6$$

and

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

Mathematical formulas are often expressed recursively. The definition of $N!$ can be expressed recursively as

$$1! = 1$$

$$N! = N * (N-1)! \text{ for } N > 1$$

KEY CONCEPT

Mathematical problems and formulas are often expressed recursively.

The base case of this definition is $1!$, which is defined to be 1. All other values of $N!$ (for $N > 1$) are defined recursively as N times the value $(N-1)!$. The recursion is that the factorial function is defined in terms of the factorial function.

COMMON ERROR

A common error made by programmers new to recursion is to provide an incomplete base case. The reason why the base case for the factorial problem ($N = 1$) works is that factorial is defined only for positive integers. A common error would be to set a base case of $N = 1$ when there is some possibility that N could be less than 1. It is important to account for all of the possibilities: $N > 1$, $N = 1$, and $N < 1$.

Using this definition, $50!$ is equal to $50 * 49!$. And $49!$ is equal to $49 * 48!$. And $48!$ is equal to $48 * 47!$. This process continues until we get to the base case of 1. Because $N!$ is defined only for positive integers, this definition is complete and will always conclude with the base case.

The next section describes how recursion is accomplished in programs.

8.2 Recursive Programming

Let's use a simple mathematical operation to demonstrate the concepts of recursive programming. Consider the process of summing the values between 1 and N , inclusive, where N is any positive integer. The sum of the values from 1 to N can be expressed as N plus the sum of the values from 1 to $N-1$. That sum can be expressed similarly, as shown in Figure 8.2.

KEY CONCEPT

Each recursive call to a method creates new local variables and parameters.

$$\begin{aligned} \sum_{i=1}^N i &= N + \sum_{i=1}^{N-1} i = N + N-1 + \sum_{i=1}^{N-2} i \\ &= N + N-1 + N-2 + \sum_{i=1}^{N-3} i \\ &= N + N-1 + N-2 + \cdots + 2 + 1 \end{aligned}$$

FIGURE 8.2 The sum of the numbers 1 through N , defined recursively

For example, the sum of the values between 1 and 20 is equal to 20 plus the sum of the values between 1 and 19. Continuing this approach, the sum of the values between 1 and 19 is equal to 19 plus the sum of the values between 1 and 18. This may sound like a strange way to think about this problem, but it is a straightforward example that can be used to demonstrate how recursion is programmed.

In Java, as in many other programming languages, a method can call itself. Each call to the method creates a new environment in which to work. That is, all local variables and parameters are newly defined with their own unique data space every time the method is called. Each parameter is given an initial value based on the new call. Each time a method terminates, processing returns to the method that called it (which may be an earlier invocation of the same method). These rules are no different from those governing any “regular” method invocation.

A recursive solution to the summation problem is defined by the following recursive method called `sum`:

```
// This method returns the sum of 1 to num

public int sum(int num)
{
    int result;
    if (num == 1)
        result = 1;
    else
        result = num + sum(num-1);
    return result;
}
```

Note that this method essentially embodies our recursive definition that the sum of the numbers between 1 and N is equal to N plus the sum of the numbers between 1 and $N-1$. The `sum` method is recursive because `sum` calls itself. The parameter passed to `sum` is decremented each time `sum` is called, until it reaches the base case of 1. Recursive methods usually contain an `if-else` statement, with one of the branches representing the base case.

Recursive Call

```
public int sum(int num) — calling a method within itself
{                                     with a different parameter value
    ...
    result = num + sum(num-1);
    ...
}
```

KEY CONCEPT

A careful trace of recursive processing can provide insight into the way it is used to solve a problem.

Suppose the `main` method calls `sum`, passing it an initial value of 1, which is stored in the parameter `num`. Because `num` is equal to 1, the result of 1 is returned to `main`, and no recursion occurs.

Now let's trace the execution of the `sum` method when it is passed an initial value of 2. Because `num` does not equal 1, `sum` is called again with an argument of `num-1`, or 1. This is a new call to the method `sum`, with a new parameter `num` and a new local variable `result`. Because this `num` is equal to 1 in this invocation, the result of 1 is returned without further recursive calls. Control returns to the first version of `sum` that was invoked. The return value of 1 is added to the initial value of `num` in that call to `sum`, which is 2. Therefore, `result` is assigned the value 3, which is returned to the `main` method. The method called from `main` correctly calculates the sum of the integers from 1 to 2 and returns the result of 3.

The base case in the summation example is when `num` equals 1, at which point no further recursive calls are made. The recursion begins to fold back into the earlier versions of the `sum` method, returning the appropriate value each time. Each return value contributes to the computation of the sum at the higher level. Without the base case, infinite recursion would result. Because each call to a method requires additional memory space, infinite recursion often results in a run-time error, indicating that memory has been exhausted.

Trace the `sum` function with different initial values of `num` until this processing becomes familiar. Figure 8.3 illustrates the recursive calls when `main` invokes `sum` to determine the sum of the integers from 1 to 4. Each box represents a copy of

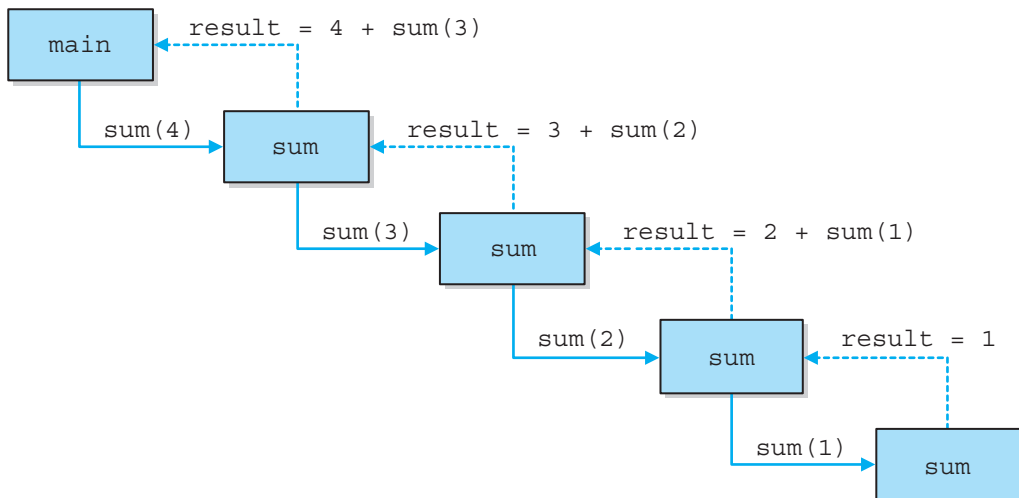


FIGURE 8.3 Recursive calls to the `sum` method

the method as it is invoked, indicating the allocation of space to store the formal parameters and any local variables. Invocations are shown as solid lines, and returns are shown as dotted lines. The return value `result` is shown at each step. The recursive path is followed completely until the base case is reached; then the calls begin to return their result up through the chain.

Recursion versus Iteration

Of course, there is an iterative solution to the summation problem we just explored:

```
sum = 0;
for (int number = 1; number <= num; number++)
    sum += number;
```

This solution is certainly more straightforward than the recursive version. If you recall our discussion from Chapter 2, we also learned that the sum of the numbers from 1 to `N` can be computed in a single step:

```
sum = num * (num+1) / 2;
```

It is important to know when recursion provides an appropriate solution to a problem. We used the summation problem to demonstrate recursion because it is a simple problem to understand, not because one would use recursion to solve it under normal conditions. Recursion has the overhead of multiple method invocations and, in this case, presents a more complicated solution than its iterative or computational counterparts.

A programmer must learn when to use recursion and when not to use it. Determining which approach is best is another important software engineering decision that depends on the problem being solved. All problems can be solved in an iterative manner, but in some cases the iterative version is much more complicated. For some problems, recursion enables us to create relatively short, elegant programs.

KEY CONCEPT

Recursion is the most elegant and appropriate way to solve some problems, but for others it is less intuitive than an iterative solution.

Direct versus Indirect Recursion

Direct recursion occurs when a method invokes itself, such as when `sum` calls `sum`. *Indirect recursion* occurs when a method invokes another method, eventually resulting in the original method being invoked again. For example, if method `m1` invokes method `m2`, and `m2` invokes method `m1`, we can say that `m1` is indirectly recursive. The amount of indirection could be several levels deep, as when `m1`

invokes `m2`, which invokes `m3`, which invokes `m4`, which invokes `m1`. Figure 8.4 depicts a situation that involves indirect recursion. Method invocations are shown with solid lines, and returns are shown with dotted lines. The entire invocation path is followed, and then the recursion unravels following the return path.

Indirect recursion requires paying just as much attention to base cases as direct recursion does. Furthermore, indirect recursion can be more difficult to trace because of the intervening method calls. Therefore, extra care is warranted when designing or evaluating indirectly recursive methods. Ensure that the indirection is truly necessary and that it is clearly explained in documentation.

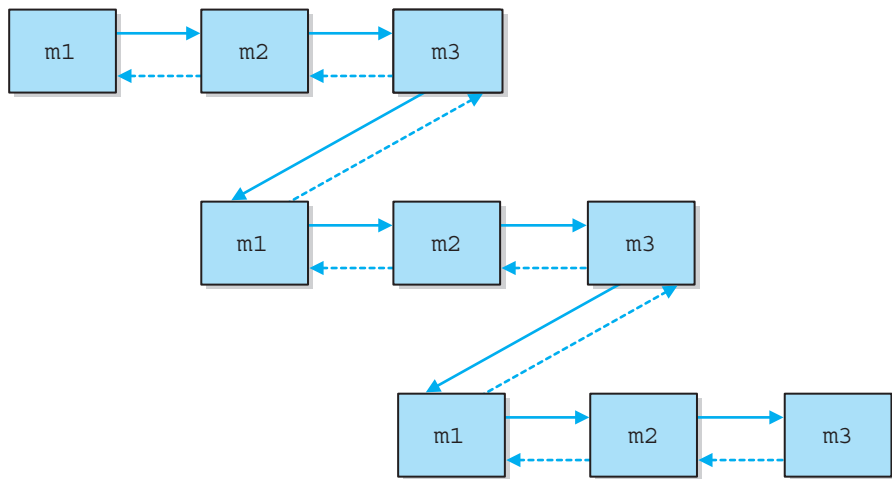


FIGURE 8.4 Indirect recursion

8.3 Using Recursion

The following sections describe problems that we solve using a recursive technique. For each one, we examine exactly how recursion plays a role in the solution and how a base case is used to terminate the recursion. As you explore these examples, consider how complicated a nonrecursive solution for each problem would be.

Traversing a Maze

As we discussed in Chapter 4, solving a maze involves a great deal of trial and error: following a path, backtracking when you cannot go farther, and trying other, untried options. Such activities often are handled nicely using recursion. In Chapter 4,

we solved this problem iteratively, using a stack to keep track of our potential moves. However, we can also solve this problem recursively by using the run-time stack to keep track of our progress. The `MazeTester` program shown in Listing 8.1 creates a `Maze` object and attempts to traverse it.

LISTING 8.1

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

/**
 * MazeTester uses recursion to determine if a maze can be traversed.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class MazeTester
{
    /**
     * Creates a new maze, prints its original form, attempts to
     * solve it, and prints out its final form.
     */
    public static void main(String[] args) throws FileNotFoundException
    {
        Scanner scan = new Scanner(System.in);
        System.out.print("Enter the name of the file containing the maze: ");
        String filename = scan.nextLine();

        Maze labyrinth = new Maze(filename);

        System.out.println(labyrinth);

        MazeSolver solver = new MazeSolver(labyrinth);
        if (solver.traverse(0, 0))
            System.out.println("The maze was successfully traversed!");
        else
            System.out.println("There is no possible path.");
        System.out.println(labyrinth);
    }
}
```

The Maze class, shown in Listing 8.2, uses a two-dimensional array of integers to represent the maze. The maze is loaded from a file. The goal is to move from the top-left corner (the entry point) to the bottom-right corner (the exit point). Initially, a 1 indicates a clear path, and a 0 indicates a blocked path. As the maze is solved, these array elements are changed to other values to indicate attempted paths and, ultimately, a successful path through the maze if one exists. Figure 8.5 shows the UML illustration of this solution.

LISTING 8.2

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

/**
 * Maze represents a maze of characters. The goal is to get from the
 * top left corner to the bottom right, following a path of 1's. Arbitrary
 * constants are used to represent locations in the maze that have been TRIED
 * and that are part of the solution PATH.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class Maze
{
    private static final int TRIED = 2;
    private static final int PATH = 3;
    private int numberOfRows, numberOfColumns;
    private int[][] grid;

    /**
     * Constructor for the Maze class. Loads a maze from the given file.
     * Throws a FileNotFoundException if the given file is not found.
     *
     * @param filename the name of the file to load
     * @throws FileNotFoundException if the given file is not found
     */
    public Maze(String filename) throws FileNotFoundException
    {
        Scanner scan = new Scanner(new File(filename));
        numberOfRows = scan.nextInt();
        numberOfColumns = scan.nextInt();

        grid = new int[numberOfRows][numberOfColumns];
        for (int i = 0; i < numberOfRows; i++)
            for (int j = 0; j < numberOfColumns; j++)
```

LISTING 8.2*continued*

```
        grid[i][j] = scan.nextInt();
    }

    /**
     * Marks the specified position in the maze as TRIED
     *
     * @param row the index of the row to try
     * @param col the index of the column to try
     */
    public void tryPosition(int row, int col)
    {
        grid[row][col] = TRIED;
    }

    /**
     * Return the number of rows in this maze
     *
     * @return the number of rows in this maze
     */
    public int getRows()
    {
        return grid.length;
    }

    /**
     * Return the number of columns in this maze
     *
     * @return the number of columns in this maze
     */
    public int getColumns()
    {
        return grid[0].length;
    }

    /**
     * Marks a given position in the maze as part of the PATH
     *
     * @param row the index of the row to mark as part of the PATH
     * @param col the index of the column to mark as part of the PATH
     */
    public void markPath(int row, int col)
    {
```


LISTING 8.2 *continued*

```

        grid[row][col] = PATH;
    }

    /**
     * Determines if a specific location is valid. A valid location
     * is one that is on the grid, is not blocked, and has not been TRIED.
     *
     * @param row the row to be checked
     * @param column the column to be checked
     * @return true if the location is valid
     */
    public boolean validPosition(int row, int column)
    {
        boolean result = false;

        // check if cell is in the bounds of the matrix
        if (row >= 0 && row < grid.length &&
            column >= 0 && column < grid[row].length)

            // check if cell is not blocked and not previously tried
            if (grid[row][column] == 1)
                result = true;

        return result;
    }

    /**
     * Returns the maze as a string.
     *
     * @return a string representation of the maze
     */
    public String toString()
    {
        String result = "\n";
        for (int row=0; row < grid.length; row++)
        {
            for (int column=0; column < grid[row].length; column++)
                result += grid[row][column] + " ";
            result += "\n";
        }
        return result;
    }
}

```

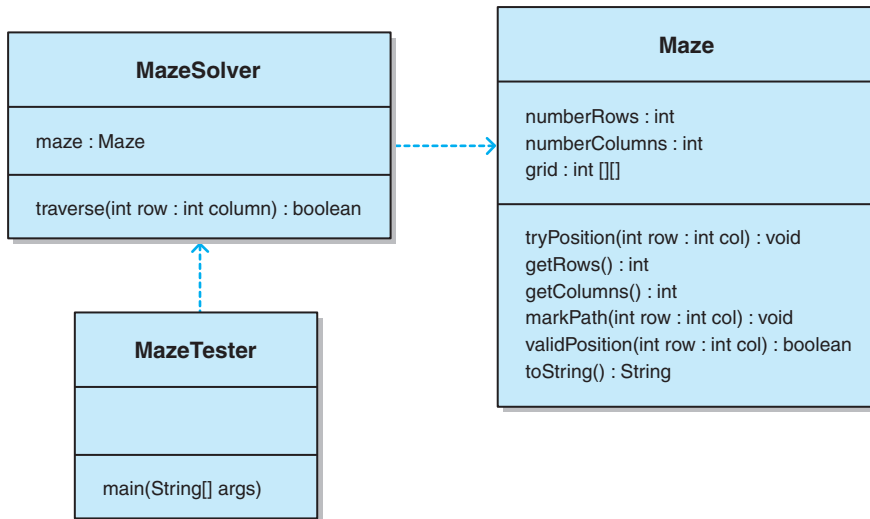


FIGURE 8.5 UML description of the maze-solving program

The only valid moves through the maze are in the four primary directions: down, right, up, and left. No diagonal moves are allowed. Listing 8.3 shows the `MazeSolver` class.

LISTING 8.3

```

/**
 * MazeSolver attempts to recursively traverse a Maze. The goal is to get from the
 * given starting position to the bottom right, following a path of 1's. Arbitrary
 * constants are used to represent locations in the maze that have been TRIED
 * and that are part of the solution PATH.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class MazeSolver
{
    private Maze maze;

    /**
     * Constructor for the MazeSolver class.
     */
}
  
```

LISTING 8.3 *continued*

```

public MazeSolver(Maze maze)
{
    this.maze = maze;
}

/**
 * Attempts to recursively traverse the maze. Inserts special
 * characters indicating locations that have been TRIED and that
 * eventually become part of the solution PATH.
 *
 * @param row row index of current location
 * @param column column index of current location
 * @return true if the maze has been solved
 */
public boolean traverse(int row, int column)
{
    boolean done = false;

    if (maze.validPosition(row, column))
    {
        maze.tryPosition(row, column);    // mark this cell as tried
        if (row == maze.getRows()-1 && column == maze.getColumns()-1)
            done = true;    // the maze is solved
        else
        {
            done = traverse(row+1, column);    // down
            if (!done)
                done = traverse(row, column+1);    // right
            if (!done)
                done = traverse(row-1, column);    // up
            if (!done)
                done = traverse(row, column-1);    // left
        }
        if (done)    // this location is part of the final path
            maze.markPath(row, column);
    }

    return done;
}
}

```

Let's think this through recursively. The maze can be traversed successfully if it can be traversed successfully from position (0, 0). Therefore, the maze can be traversed successfully if it can be traversed successfully from any position adjacent to (0, 0)—namely, position (1, 0), position (0, 1), position (−1, 0), or position (0, −1). Picking a potential next step, say (1, 0), we find ourselves in the same type of situation as before. To traverse the maze successfully from the new current position, we must successfully traverse it from an adjacent position. At any point, some of the adjacent positions may be invalid, may be blocked, or may represent a possible successful path. We continue this process recursively. If the base case position is reached, the maze has been traversed successfully.

The recursive method in the `MazeSolver` class is called `traverse`. It returns a `boolean` value that indicates whether a solution was found. First the method determines whether a move to the specified row and column is valid. A move is considered valid if it stays within the grid boundaries and if the grid contains a 1 in that location, indicating that a move in that direction is not blocked. The initial call to `traverse` passes in the upper-left location (0, 0).

If the move is valid, the grid entry is changed from a 1 to a 2, marking this location as visited so that we don't retrace our steps later. Then the `traverse` method determines whether the maze has been completed by having reached the bottom-right location. Therefore, there are actually three possibilities of the base case for this problem that will terminate any particular recursive path:

- An invalid move because the move is out of bounds or blocked
- An invalid move because the move has been tried before
- A move that arrives at the final location

If the current location is not the bottom-right corner, we search for a solution in each of the primary directions. First, we look down by recursively calling the `traverse` method and passing in the new location. The logic of the `traverse` method starts all over again using this new position. Either a solution is ultimately found by first attempting to move down from the current location, or it is not found. If it's not found, we try moving right. If that fails, we try moving up. Finally, if no other direction has yielded a correct path, we try moving left. If no direction from the current location yields a correct solution, then there is no path from this location, and `traverse` returns `false`. If the very first invocation of the `traverse` method returns `false`, then there is no possible path through this maze.

If a solution is found from the current location, then the grid entry is changed to a 3. The first 3 is placed in the bottom-right corner. The next 3 is placed in the location that led to the bottom-right corner, and so on until the final 3 is placed in the upper-left corner. Therefore, when the final maze is printed, 0 still indicates a blocked path, 1 indicates an open path that was never tried, 2 indicates a path

that was tried but failed to yield a correct solution, and 3 indicates a part of the final solution of the maze.

Here are a sample maze input file and its corresponding output:

```

5 5
1 0 0 0 0
1 1 1 1 0
0 1 0 0 0
1 1 1 1 0
0 1 0 1 1

3 0 0 0 0
3 3 1 1 0
0 3 0 0 0
1 3 3 3 0
0 2 0 3 3

```

Note that there are several opportunities for recursion in each call to the `traverse` method. Any or all of them might be followed, depending on the maze configuration. Although there may be many paths through the maze, the recursion terminates when a path is found. Carefully trace the execution of this code, while following the maze array to see how the recursion solves the problem. Then consider the difficulty of producing a nonrecursive solution.

The Towers of Hanoi

The *Towers of Hanoi* puzzle was invented in the 1880s by Edouard Lucas, a French mathematician. It has become a favorite among computer scientists because its solution is an excellent demonstration of recursive elegance.

The puzzle consists of three upright pegs (towers) and a set of disks with holes in the middle so that they slide onto the pegs. Each disk has a different diameter. Initially, all of the disks are stacked on one peg in order of size such that the largest disk is on the bottom, as shown in Figure 8.6.



FIGURE 8.6 The Towers of Hanoi puzzle

The goal of the puzzle is to move all of the disks from their original (first) peg to the destination (third) peg. We can use the “extra” peg as a temporary place to put disks, but we must obey the following three rules:

- We can move only one disk at a time.
- We cannot place a larger disk on top of a smaller disk.
- All disks must be on some peg except for the disk that is in transit between pegs.

These rules imply that we must move smaller disks “out of the way” in order to move a larger disk from one peg to another. Figure 8.7 shows the step-by-step solution for the Towers of Hanoi puzzle using three disks. To move all three disks from the first peg to the third peg, we first have to get to the point where the smaller two disks are out of the way on the second peg so that the largest disk can be moved from the first peg to the third peg.

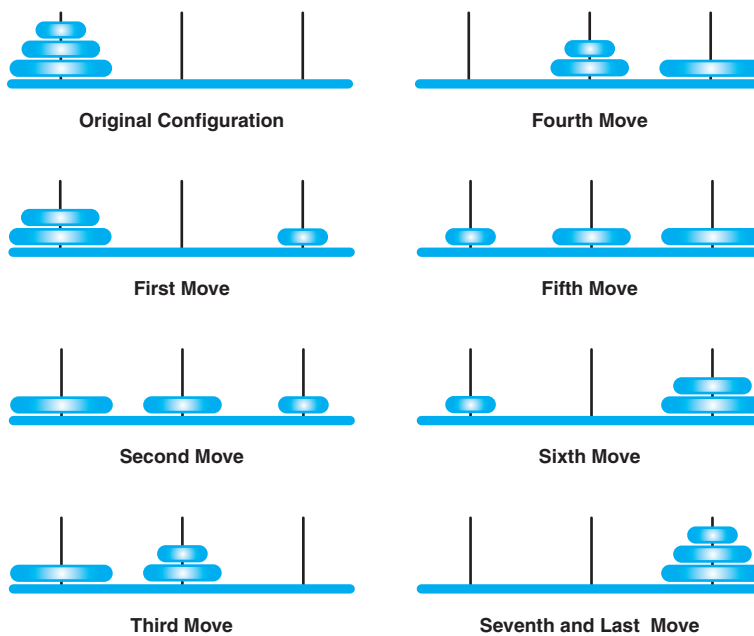


FIGURE 8.7 A solution to the three-disk Towers of Hanoi puzzle

The first three moves shown in Figure 8.7 can be thought of as “moving the smaller disks out of the way.” The fourth move puts the largest disk in its final place. The last three moves put the smaller disks in their final place on top of the largest one.

Let's use this idea to form a general strategy. To move a stack of N disks from the original peg to the destination peg:

- Move the topmost $N-1$ disks from the original peg to the extra peg.
- Move the largest disk from the original peg to the destination peg.
- Move the $N-1$ disks from the extra peg to the destination peg.

This strategy lends itself nicely to a recursive solution. The step to move the $N-1$ disks out of the way is the same problem all over again: moving a stack of disks. For this subtask, though, there is one less disk, and our destination peg is what we were originally calling the extra peg. An analogous situation occurs after we have moved the largest disk, and we have to move the original $N-1$ disks again.

The base case for this problem occurs when we want to move a “stack” that consists of only one disk. That step can be accomplished directly and without recursion.

The program in Listing 8.4 creates a `TowersOfHanoi` object and invokes its `solve` method. The output is a step-by-step list of instructions that describes how the disks should be moved to solve the puzzle. This example uses four disks, which is specified by a parameter to the `TowersOfHanoi` constructor.

LISTING 8.4

```
/**
 * SolveTowers uses recursion to solve the Towers of Hanoi puzzle.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class SolveTowers
{
    /**
     * Creates a TowersOfHanoi puzzle and solves it.
     */
    public static void main(String[] args)
    {
        TowersOfHanoi towers = new TowersOfHanoi(4);
        towers.solve();
    }
}
```

The `TowersOfHanoi` class, shown in Listing 8.5, uses the `solve` method to make an initial call to `moveTower`, the recursive method. The initial call indicates that all of the disks should be moved from peg 1 to peg 3, using peg 2 as the extra position.

The `moveTower` method first considers the base case (a “stack” of one disk). When that occurs, it calls the `moveOneDisk` method, which prints a single line describing that particular move. If the stack contains more than one disk, we call `moveTower` again to get the $N-1$ disks out of the way, then move the largest disk, then move the $N-1$ disks to their final destination with yet another call to `moveTower`.

LISTING 8.5

```
/**
 * TowersOfHanoi represents the classic Towers of Hanoi puzzle.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class TowersOfHanoi
{
    private int totalDisks;

    /**
     * Sets up the puzzle with the specified number of disks.
     *
     * @param disks the number of disks
     */
    public TowersOfHanoi(int disks)
    {
        totalDisks = disks;
    }

    /**
     * Performs the initial call to moveTower to solve the puzzle.
     * Moves the disks from tower 1 to tower 3 using tower 2.
     */
    public void solve()
    {
        moveTower(totalDisks, 1, 3, 2);
    }
}
```


LISTING 8.5 *continued*

```

/**
 * Moves the specified number of disks from one tower to another
 * by moving a subtower of n-1 disks out of the way, moving one
 * disk, then moving the subtower back. Base case of 1 disk.
 *
 * @param numDisks the number of disks to move
 * @param start    the starting tower
 * @param end      the ending tower
 * @param temp     the temporary tower
 */
private void moveTower(int numDisks, int start, int end, int temp)
{
    if (numDisks == 1)
        moveOneDisk(start, end);
    else
    {
        moveTower(numDisks-1, start, temp, end);
        moveOneDisk(start, end);
        moveTower(numDisks-1, temp, end, start);
    }
}

/**
 * Prints instructions to move one disk from the specified start
 * tower to the specified end tower.
 *
 * @param start the starting tower
 * @param end   the ending tower
 */
private void moveOneDisk(int start, int end)
{
    System.out.println("Move one disk from " + start + " to " + end);
}
}

```

Note that the parameters to `moveTower` describing the pegs are switched around as needed to move the partial stacks. This code follows our general strategy and uses the `moveTower` method to move all partial stacks. Trace the code carefully for a stack of three disks to understand the processing. Figure 8.8 shows the UML diagram for this problem.

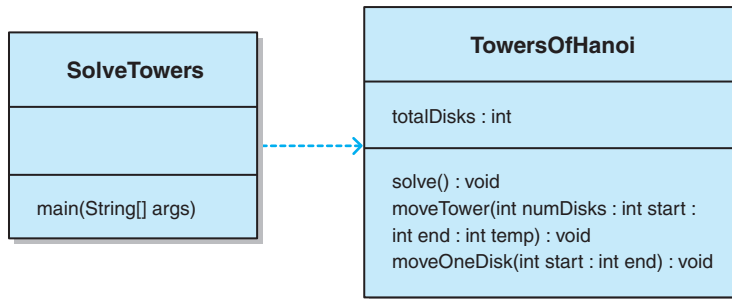


FIGURE 8.8 UML description of Towers of Hanoi puzzle solution

8.4 Analyzing Recursive Algorithms

In Chapter 2, we explored the concept of analyzing an algorithm to determine its complexity (usually its time complexity) and expressed it in terms of a growth function. The growth function gave us the order of the algorithm, which can be used to compare it to other algorithms that accomplish the same task.

When analyzing a loop, we determined the order of the body of the loop and multiplied it by the number of times the loop was executed. Analyzing a recursive algorithm uses similar thinking. Determining the order of a recursive algorithm is a matter of determining the order of the recursion (the number of times the recursive definition is followed) and multiplying that by the order of the body of the recursive method.

Consider the recursive method presented in Section 8.2 that computes the sum of the integers from 1 to some positive value. We reprint it here for convenience:

```
// This method returns the sum of 1 to num

public int sum (int num)
{
    int result;
    if (num == 1)
        result = 1;
    else
        result = num + sum (num-1);
    return result;
}
```

KEY CONCEPT

The order of a recursive algorithm can be determined using techniques similar to those used in analyzing iterative processing.

The size of this problem is naturally expressed as the number of values to be summed. Because we are summing the integers from 1 to `num`, the number of values to be summed is `num`. The operation of interest is the act of adding two values together. The body of the recursive method performs one addition operation and therefore is $O(1)$. Each time the recursive method is invoked, the value of `num` is decreased by 1. Therefore, the recursive method is called `num` times, so the order of the recursion is $O(n)$. Thus, because the body is $O(1)$ and the recursion is $O(n)$, the order of the entire algorithm is $O(n)$.

We will see that in some algorithms the recursive step operates on half as much data as the previous call, thus creating an order of recursion of $O(\log n)$. If the body of the method is $O(1)$, then the whole algorithm is $O(\log n)$. If the body of the method is $O(n)$, then the whole algorithm is $O(n \log n)$.

Now consider the Towers of Hanoi puzzle. The size of the puzzle is naturally the number of disks, and the processing operation of interest is the step of moving one disk from one peg to another. Each call to the recursive method `moveTower` results in one disk being moved. Unfortunately, except for the base case, each recursive call results in calling itself *twice more*, and each call operates on a stack of disks that is only one less than the stack that is passed in as the parameter. Thus, calling `moveTower` with 1 disk results in 1 disk being moved, calling `moveTower` with 2 disks results in 3 disks being moved, calling `moveTower` with 3 disks results in 7 disks being moved, calling `moveTower` with 4 disks results in 15 disks being moved, and so on. Looking at it another way, if $f(n)$ is the growth function for this problem, then

$$\begin{aligned} f(n) &= 1 \text{ when } n \text{ is equal to } 1 \\ \text{for } n > 1, \\ f(n) &= 2 * (f(n - 1) + 1) \\ &= 2^n - 1 \end{aligned}$$

Contrary to its short and elegant implementation, the solution to the Towers of Hanoi puzzle is terribly inefficient. To solve the puzzle with a stack of n disks, we have to make $2^n - 1$ individual disk moves. Therefore, the Towers of Hanoi algorithm is $O(2^n)$. This order is an example of exponential complexity. As the number of disks increases, the number of required moves increases exponentially.

Legend has it that priests of Brahma are working on this puzzle in a temple at the center of the world. They are using 64 gold disks, moving them between pegs of pure diamond. The downside is that when the priests finish the puzzle, the world will end. The upside is that even if they move one disk every second of every day, it will take them over 584 billion years to complete it. That's with a puzzle of only 64 disks! It is certainly an indication of just how intractable exponential algorithm complexity is.

KEY CONCEPT

The Towers of Hanoi solution has exponential complexity, which is very inefficient, yet the code is remarkably short and elegant.



VideoNote

Analyzing recursive algorithms

Summary of Key Concepts

- Recursion is a programming technique in which a method calls itself. A key to being able to program recursively is to be able to think recursively.
- Any recursive definition must have a nonrecursive part, called the base case, that permits the recursion to eventually end.
- Mathematical problems and formulas are often expressed recursively.
- Each recursive call to a method creates new local variables and parameters.
- A careful trace of recursive processing can provide insight into the way it is used to solve a problem.
- Recursion is the most elegant and appropriate way to solve some problems, but for others it is less intuitive than an iterative solution.
- The order of a recursive algorithm can be determined using techniques similar to those used in analyzing iterative processing.
- The Towers of Hanoi solution has exponential complexity, which is very inefficient, yet the code is incredibly short and elegant.

Summary of Terms

base case The part of an operation's definition that is not recursive.

direct recursion The type of recursion in which a method invokes itself directly (as opposed to indirect recursion).

indirect recursion The type of recursion in which a method calls another method, which may call yet another, and so on until the original method is called (as opposed to direct recursion).

infinite recursion The problem that occurs when a base case is never reached or not defined for an operation.

recursion A programming technique in which a method calls itself to fulfill its overall purpose.

Towers of Hanoi A classic computing puzzle in which the goal is to move disks from one tower to another under specific rules.

Self-Review Questions

SR 8.1 What is recursion?

SR 8.2 What is infinite recursion?

- SR 8.3 When is a base case needed for recursive processing?
- SR 8.4 Is recursion necessary?
- SR 8.5 When should recursion be avoided?
- SR 8.6 What is indirect recursion?
- SR 8.7 Explain the general approach to solving the Towers of Hanoi puzzle. How is it related to recursion?

Exercises

- EX 8.1 Write a recursive definition for the greatest common divisor (gcd) of two numbers.
- EX 8.2 Write a recursive method that determines if its parameter is a prime number.
- EX 8.3 Write a recursive method `sumOfArray` that calculates the sum of integers stored in an array within a given range. The method takes three parameters: the array, the lower index of the range, and the upper index of the range.
- EX 8.4 You have provided a recursive definition for the greatest common divisor of two numbers in Exercise 8.1. The greatest common divisor (gcd), of two or more non-zero integers is the largest positive integer that divides the numbers without a remainder. Write a recursive method `gcd` that would return greatest common divisor of two numbers.
- EX 8.5 Modify and extend the method that calculates the sum of the integers between 1 and N shown in this chapter. Write a method `sumOfSquares` that would return the sum of squares of N integers, from 1 through N.
- EX 8.6 Write a recursive method that returns the value of N! (N factorial) using the definition given in this chapter. Explain why you would not normally use recursion to solve this problem.
- EX 8.7 Write a recursive method to reverse a string. Explain why you would not normally use recursion to solve this problem.
- EX 8.8 A palindrome is a word, phrase, number, or other sequence of units that can be read the same way in either direction. For example, the words *civic*, *madam*, *deed*, etc. are palindromes. Write a recursive method that determines if its `String` argument is a palindrome.
- EX 8.9 Annotate the lines of output of the `SolveTowers` program in this chapter to show the recursive steps.
- EX 8.10 Write a recursive method `combination` that returns the combination of n things taken r at a time where n and r both are nonnegative integers.

- EX 8.11 Write a recursive method that writes the sum of digits of a positive decimal integer.
- EX 8.12 Determine and explain the order of your solution to Exercise 8.5.
- EX 8.13 Determine and explain the order of your solution to Exercise 8.6.
- EX 8.14 Determine the order of the recursive maze solution presented in this chapter.

Programming Projects

- PP 8.1 Design and implement a program that implements Euclid's algorithm for finding the greatest common divisor of two positive integers. The greatest common divisor is the largest integer that divides both values without producing a remainder. In a class called `DivisorCalc`, define a static method called `gcd` that accepts two integers, `num1` and `num2`. Create a driver to test your implementation. The recursive algorithm is defined as follows:

```
gcd (num1, num2) is num2 if num2 <= num1 and num2
divides num1
gcd (num1, num2) is gcd (num2, num1) if num1 < num2
gcd (num1, num2) is gcd (num2, num1%num2) otherwise
```

- PP 8.2 Modify the `Maze` class so that it prints out the path of the final solution as it is discovered, without storing it.
- PP 8.3 Design and implement a program that traverses a 3D maze.
- PP 8.4 Design and implement a recursive program that solves the Nonattacking Queens problem. That is, write a program to determine how eight queens can be positioned on an eight-by-eight chessboard so that none of them is in the same row, column, or diagonal as any other queen. There are no other chess pieces on the board.
- PP 8.5 In the language of an alien race, all words take the form of Blurbs. A Blurb is a Whoozit followed by one or more Whatzits. A Whoozit is the character 'x' followed by zero or more 'y's. A Whatzit is a 'q' followed by either a 'z' or a 'd' followed by a Whoozit. Design and implement a recursive program that generates random Blurbs in this alien language.
- PP 8.6 Design and implement a recursive program to determine whether a string is a valid Blurb as defined in the previous project description.

- PP 8.7 Design and implement a recursive program to determine and print the Nth line of Pascal's Triangle, as shown below. Each interior value is the sum of the two values above it. (*Hint*: Use an array to store the values on each line.)

```

          1
         1 1
        1 2 1
       1 3 3 1
      1 4 6 4 1
     1 5 10 10 5 1
    1 6 15 20 15 6 1
   1 7 21 35 35 21 7 1
  1 8 28 56 70 56 28 8 1

```

- PP 8.8 Design and implement a graphical version of the Towers of Hanoi puzzle. Allow the user to set the number of disks used in the puzzle. The user should be able to interact with the puzzle in two main ways. The user can move the disks from one peg to another using the mouse, in which case the program should ensure that each move is legal. The user can also watch a solution take place as an animation, with pause/resume buttons. Permit the user to control the speed of the animation.

Answers to Self-Review Questions

- SRA 8.1 Recursion is a programming technique in which a method calls itself, solving a smaller version of the problem each time, until the terminating condition is reached.
- SRA 8.2 Infinite recursion occurs when there is no base case that serves as a terminating condition, or when the base case is improperly specified. The recursive path is followed forever. In a recursive program, infinite recursion often results in an error that indicates that available memory has been exhausted.
- SRA 8.3 A base case is always needed to terminate recursion and begin the process of returning through the calling hierarchy. Without the base case, infinite recursion results.
- SRA 8.4 Recursion is not necessary. Every recursive algorithm can be written in an iterative manner. However, some problem solutions are much more elegant and straightforward when they are written recursively.

- SRA 8.5 Avoid recursion when the iterative solution is simpler and more easily understood and programmed. Recursion has the overhead of multiple method calls and is not always intuitive.
- SRA 8.6 Indirect recursion occurs when a method calls another method, which calls another method, and so on until one of the called methods invokes the original method. Indirect recursion is usually more difficult to trace than direct recursion, in which a method calls itself.
- SRA 8.7 The Towers of Hanoi puzzle of N disks is solved by moving $N-1$ disks out of the way onto an extra peg, moving the largest disk to its destination, then moving the $N-1$ disks from the extra peg to the destination. This solution is inherently recursive because we can use the same process to move the whole substack of $N-1$ disks.

This page is intentionally left blank.



Searching and Sorting

9

Two common tasks in the world of software development are searching for a particular element within a group and sorting a group of elements into a particular order. There are a variety of algorithms that can be used to accomplish these tasks, and the differences among them are worth exploring carefully. These topics go hand in hand with the study of collections and data structures.

CHAPTER OBJECTIVES

- Examine the linear search and binary search algorithms.
- Examine several sort algorithms.
- Discuss the complexity of these algorithms.

9.1 Searching

Searching is the process of finding a designated *target element* within a group of items, or determining that the target does not exist within the group. The group of items to be searched is sometimes called the *search pool*.

KEY CONCEPT

Searching is the process of finding a designated target within a group of items or determining that the target doesn't exist.

This section examines two common approaches to searching: a linear search and a binary search. Later in this book, other search techniques are presented that use the characteristics of particular data structures to facilitate the search process.

Our goal is to perform the search as efficiently as possible. In terms of algorithm analysis, we want to minimize the number of comparisons we have to make to find the target. In general, the more items there are in the search pool, the more comparisons it will take to find the target. Thus, the size of the problem is defined by the number of items in the search pool.

KEY CONCEPT

An efficient search minimizes the number of comparisons made.

To be able to search for an object, we must be able to compare one object to another. Our implementations of these algorithms search an array of `Comparable` objects. Therefore, the elements involved must implement the `Comparable` interface and be comparable to each other. We might attempt to accomplish this restriction in the header for the `Searching` class in which all of our searching methods are located by doing something like

```
public class Searching<T extends Comparable<T>>
```

The net effect of this generic declaration is that we can instantiate the `Searching` class with any class that implements the `Comparable` interface. Recall that the `Comparable` interface contains one method, `compareTo`, which is designed to return an integer that is less than zero, equal to zero, or greater than zero (respectively) if the object is less than, equal to, or greater than the object to which it is being compared. Therefore, any class that implements the `Comparable` interface defines the relative order of any two objects of that class.

Declaring the `Searching` class in this manner, however, will cause us to have to instantiate the class anytime we want to use one of the search methods. This is awkward at best for a class that contains nothing but service methods. A better solution would be to declare all of the methods as static and generic. Let's first remind ourselves about the concept of static methods, and then we will explore generic static methods.

Static Methods

A *static method* (also called a *class method*) can be invoked through the class name (all the methods of the `Math` class are static methods, for example). You don't have to instantiate an object of the class to invoke a static method. For example, the `sqrt` method is called through the `Math` class as follows:

```
System.out.println("Square root of 27: " + Math.sqrt(27));
```

A method is made static by using the `static` modifier in the method declaration. As we have seen, the `main` method of a Java program must be declared with the `static` modifier; this is so that `main` can be executed by the interpreter without instantiating an object from the class that contains `main`.

Because static methods do not operate in the context of a particular object, they cannot reference instance variables, which exist only in an instance of a class. The compiler will issue an error if a static method attempts to use a nonstatic variable. A static method can, however, reference static variables, because static variables exist independent of specific objects. Therefore, the `main` method can access only static and local variables.

The methods in the `Math` class perform basic computations based on values passed as parameters. There is no object state to maintain in these situations; therefore, there is no good reason to force us to create an object in order to request these services.

KEY CONCEPT

A method is made static by using the `static` modifier in the method declaration.

Generic Methods

In a manner similar to what we have done in creating generic classes, we can also create generic methods. That is, instead of creating a class that refers to a generic type parameter, we can create an individual method that does so. A generic parameter applies only to that method.

To create a generic method, we insert a generic declaration in the header of the method immediately preceding the return type.

```
public static <T extends Comparable<T>> boolean  
    linearSearch(T[] data, int min, int max, T target)
```

Now that method, including the return type and the types of the parameters, can make use of the generic type parameter. It makes sense that the generic declaration has to come before the return type so that, although this example doesn't do so, the generic type can be used in the return type.

Generic Method

generic type parameter applies to this method

```
public static <T extends Comparable<T>> boolean
    linearSearch(T[] data, int min, int max, T target)
```

generic type can be used in
method parameters and return type

Now that we can create a generic static method, we do not need to instantiate the `Searching` class each time we need one of the methods. Instead, we can simply invoke the static method using the class name and including our type to replace the generic type. For example, an invocation of the `linearSearch` method to search an array of `Strings` might look like this:

```
Searching.linearSearch(targetarray, min, max, target);
```

Note that it is not necessary to specify the type to replace the generic type. The compiler will infer the type from the arguments provided. Thus, for this line of code, the compiler will replace the generic type `T` with whatever the element type is for `targetarray` and the type of `target`.

Linear Search

If the search pool is organized into a list of some kind, one straightforward way to perform the search is to start at the beginning of the list and compare each value in turn to the target element. Eventually, we will either find the target or come to the end of the list and conclude that the target doesn't exist in the group. This approach is called a *linear search* because it begins at one end and scans the search pool in a linear manner. This process is depicted in Figure 9.1.

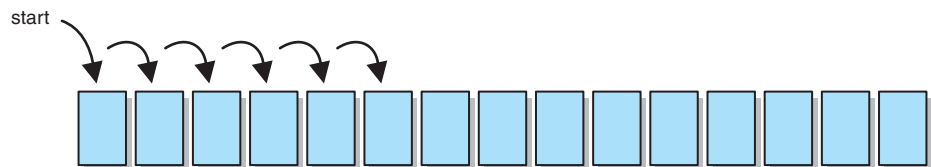


FIGURE 9.1 A linear search

The following method implements a linear search. It accepts the array of elements to be searched, the beginning and ending index for the search, and the target value sought. The method returns a `boolean` value that indicates whether or not the target element was found.

```
/**
 * Searches the specified array of objects using a linear search
 * algorithm.
 *
 * @param data    the array to be searched
 * @param min     the integer representation of the minimum value
 * @param max     the integer representation of the maximum value
 * @param target  the element being searched for
 * @return       true if the desired element is found
 */
public static <T>
    boolean linearSearch(T[] data, int min, int max, T target)
{
    int index = min;
    boolean found = false;

    while (!found && index <= max)
    {
        found = data[index].equals(target);
        index++;
    }

    return found;
}
```

The `while` loop steps through the elements of the array, terminating either when the element is found or when the end of the array is reached. The `boolean` variable `found` is initialized to `false` and is changed to `true` only if the target element is located.

Variations on this implementation could return the element found in the array if it is found and return a null reference if it is not found. Alternatively, an exception could be thrown if the target element is not found.

The `linearSearch` method could be incorporated into any class. Our version of this method is defined as part of a class containing methods that provide various searching capabilities.

The linear search algorithm is fairly easy to understand, although it is not particularly efficient. Note that a linear search does not require the elements in the

search pool to be in any particular order within the array. The only criterion is that we must be able to examine them one at a time in turn. The binary search algorithm, described next, improves on the efficiency of the search process, but it works only if the search pool is ordered.

Binary Search

If the group of items in the search pool is sorted, then our approach to searching can be much more efficient than that of a linear search. A *binary search* algorithm eliminates large parts of the search pool with each comparison by capitalizing on the fact that the search pool is in sorted order.

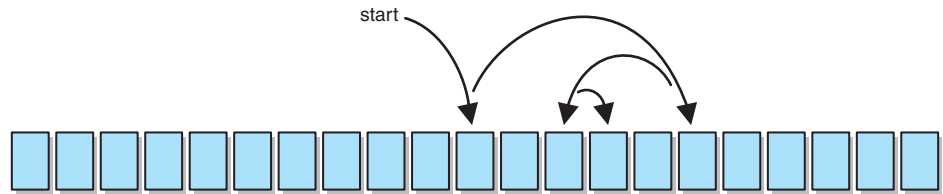


FIGURE 9.2 A binary search

KEY CONCEPT

A binary search capitalizes on the fact that the search pool is sorted.

Instead of starting the search at one end or the other, a binary search begins in the middle of the sorted list. If the target element is not found at that middle element, then the search continues. And because the list is sorted, we know that if the target is in the list, it will be on one side of the array or the other, depending on whether the target is less than or greater than the middle element. Thus, because the list is sorted, we eliminate half of the search pool with one carefully chosen comparison. The remaining half of the search pool represents the *viable candidates* in which the target element may yet be found.

The search continues in this same manner, examining the middle element of the viable candidates, eliminating half of them. Each comparison reduces the viable candidates by half until eventually the target element is found or there are no more viable candidates, which means the target element is not in the search pool. The process of a binary search is depicted in Figure 9.2.

Let's look at an example. Consider the following sorted list of integers:

10 12 18 22 31 34 40 46 59 67 69 72 80 84 98

Suppose we were trying to determine whether the number 67 is in the list. Initially, the target could be anywhere in the list (all items in the search pool are viable candidates).

The binary search approach begins by examining the middle element, in this case 46. That element is not our target, so we must continue searching. But since we know that the list is sorted, we know that if 67 is in the list, it must be in the second half of the data, because all data items to the left of the middle have values of 46 or less. This leaves the following viable candidates to search (shown in bold):

10 12 18 22 31 34 40 46 **59 67 69 72 80 84 98**

Continuing the same approach, we examine the middle value of the viable candidates (72). Again, this is not our target value, so we must continue the search. This time we can eliminate all values higher than 72, which leaves (again in bold)

10 12 18 22 31 34 40 46 **59 67 69** 72 80 84 98

Note that in only two comparisons, we have reduced the viable candidates from 15 items to 3 items. Employing the same approach again, we select the middle element, 67, and find the element we are seeking. If 67 had not been our target, we would have continued with this process until we had either found the target value or eliminated all possible data.

With each comparison, a binary search eliminates approximately half of the data remaining to be searched (it also eliminates the middle element). That is, a binary search eliminates half of the data with the first comparison, another quarter of the data with the second comparison, another eighth of the data with the third comparison, and so on.

The following method implements a binary search. Like the `linearSearch` method, it accepts an array of `Comparable` objects to be searched as well as the target value. It also takes integer values representing the minimum index and maximum index that define the portion of the array to search (the viable candidates).

KEY CONCEPT

A binary search eliminates half of the viable candidates with each comparison.



VideoNote

Demonstration of a binary search

```
/**
 * Searches the specified array of objects using a binary search
 * algorithm.
 *
 * @param data    the array to be searched
 * @param min     the integer representation of the minimum value
 * @param max     the integer representation of the maximum value
 * @param target  the element being searched for
 * @return       true if the desired element is found
 */
public static <T extends Comparable<T>>
    boolean binarySearch(T[] data, int min, int max, T target)
{
```



```

boolean found = false;
int midpoint = (min + max) / 2; // determine the midpoint

if (data[midpoint].compareTo(target) == 0)
    found = true;

else if (data[midpoint].compareTo(target) > 0)
{
    if (min <= midpoint - 1)
        found = binarySearch(data, min, midpoint - 1, target);
}

else if (midpoint + 1 <= max)
    found = binarySearch(data, midpoint + 1, max, target);

return found;
}

```

Note that the `binarySearch` method is implemented recursively. If the target element is not found, and there are more data to search, the method calls itself, passing parameters that shrink the size of viable candidates within the array. The `min` and `max` indexes are used to determine whether there are still more data to search. That is, if the reduced search area does not contain at least one element, the method does not call itself, and a value of `false` is returned.

At any point in this process, we may have an even number of values to search—and therefore two “middle” values. As far as the algorithm is concerned, the midpoint used can be either of the two middle values, as long as the same choice is made consistently. In this implementation of the binary search, the calculation that determines the midpoint index discards any fractional part and therefore picks the first of the two middle values.

Comparing Search Algorithms

For a linear search, the best case occurs when the target element happens to be the first item we examine in the group. The worst case occurs when the target is not in the group, and we have to examine every element before we determine that it isn’t present. The expected case is that we will have to search half of the list before we find the element. That is, if there are n elements in the search pool, then on average we will have to examine $n/2$ elements before finding the one for which we were searching.

Therefore, the linear search algorithm has a linear time complexity of $O(n)$. Because the elements are searched one at a time in turn, the complexity is linear—in direct proportion to the number of elements to be searched.

A binary search, on the other hand, is generally much faster. Because we can eliminate half of the remaining data with each comparison, we can find the element much more quickly. The best case is that we find the target in one comparison—that is, the target element happens to be at the midpoint of the array. The worst case occurs when the element is not present in the list, in which case we have to make approximately $\log_2 n$ comparisons before we eliminate all of the data. Thus, the expected case for finding an element that is in the search pool is approximately $(\log_2 n)/2$ comparisons.

Therefore, binary search is a *logarithmic algorithm* and has a time complexity of $O(\log_2 n)$. Compared to a linear search, a binary search is much faster for large values of n .

A question might be asked here: If a binary search is more efficient than a linear search, why would we ever use a linear search? First, a linear search is generally simpler than a binary search, and it is therefore easier to program and debug. Second, a linear search does not require the additional overhead of sorting the search list. Thus conducting a binary search involves a trade-off: Achieving maximum efficiency requires investing the effort to keep the search pool sorted.

For small problems, there is little practical difference between the two types of algorithms. However, as n gets larger, the binary search becomes increasingly attractive. Suppose a given set of data contains a million elements. In a linear search, we would have to examine each of the one million elements to determine that a particular target element is not in the group. In a binary search, we could make that conclusion in roughly 20 comparisons.

KEY CONCEPT

A binary search has logarithmic complexity, which makes it a very efficient way to examine a large search pool.

9.2 Sorting

Sorting is the process of arranging a group of items into a defined order, either ascending or descending, based on some criterion. For example, you may want to alphabetize a list of names or put a list of survey results into descending numeric order.

Many sort algorithms have been developed and critiqued over the years. In fact, sorting is considered a classic area of study in computer science. Like search algorithms, sort algorithms generally are divided into two categories based on efficiency: *Sequential sorts* typically use a pair of nested loops and require roughly n^2 comparisons to sort n elements, and *logarithmic sorts* typically require roughly $n \log_2 n$ comparisons to sort n elements. As with the search algorithms, when n is small, there is little practical difference between the two categories of algorithms.

KEY CONCEPT

Sorting is the process of arranging a list of items into a defined order based on some criterion.

In this chapter, we examine three sequential sorts—selection sort, insertion sort, and bubble sort—and two logarithmic sorts—quick sort and merge sort. We also take a look at one additional sort algorithm—radix sort—that sorts without comparing elements.

Before we dive into particular sort algorithms, let's look at a general sorting problem to solve. The `SortPhoneList` program, shown in Listing 9.1, creates an array of `Contact` objects, sorts those objects, and then prints the sorted list. In this implementation, the `Contact` objects are sorted using a call to the `selectionSort` method, which we examine later in this chapter. However, any other sorting method described in this chapter could be used to achieve the same results.

LISTING 9.1

```
/**
 * SortPhoneList driver for testing an object selection sort.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class SortPhoneList
{
    /**
     * Creates an array of Contact objects, sorts them, then prints
     * them.
     */
    public static void main(String[] args)
    {
        Contact[] friends = new Contact[7];
        friends[0] = new Contact("John", "Smith", "610-555-7384");
        friends[1] = new Contact("Sarah", "Barnes", "215-555-3827");
        friends[2] = new Contact("Mark", "Riley", "733-555-2969");
        friends[3] = new Contact("Laura", "Getz", "663-555-3984");
        friends[4] = new Contact("Larry", "Smith", "464-555-3489");
        friends[5] = new Contact("Frank", "Phelps", "322-555-2284");
        friends[6] = new Contact("Marsha", "Grant", "243-555-2837");

        Sorting.insertionSort(friends);
        for (Contact friend : friends)
            System.out.println(friend);
    }
}
```

Each `Contact` object represents a person with a last name, a first name, and a phone number. The `Contact` class is shown in Listing 9.2. The UML description of these classes is left as an exercise.

The `Contact` class implements the `Comparable` interface and therefore provides a definition of the `compareTo` method. In this case, the contacts are sorted by last name; if two contacts have the same last name, their first names are used.

LISTING 9.2

```
/**
 * Contact represents a phone contact.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class Contact implements Comparable<Contact>
{
    private String firstName, lastName, phone;

    /**
     * Sets up this contact with the specified information.
     *
     * @param first    a string representation of a first name
     * @param last     a string representation of a last name
     * @param telephone a string representation of a phone number
     */
    public Contact(String first, String last, String telephone)
    {
        firstName = first;
        lastName = last;
        phone = telephone;
    }

    /**
     * Returns a description of this contact as a string.
     *
     * @return a string representation of this contact
     */
    public String toString()
    {
        return lastName + ", " + firstName + "\t" + phone;
    }
}
```

LISTING 9.2

continued

```

/**
 * Uses both last and first names to determine lexical ordering.
 *
 * @param other the contact to be compared to this contact
 * @return      the integer result of the comparison
 */
public int compareTo(Contact other)
{
    int result;
    if (lastName.equals(other.lastName))
        result = firstName.compareTo(other.firstName);
    else
        result = lastName.compareTo(other.lastName);
    return result;
}

```

KEY CONCEPT

The selection sort algorithm sorts a list of values by repetitively putting a particular value into its final, sorted position.

Now let's examine several sort algorithms and their implementations. Any of these could be used to put the `Contact` objects into sorted order.

Selection Sort

The *selection sort* algorithm sorts a list of values by repetitively putting a particular value into its final, sorted position. In other words, for each position in the list, the algorithm selects the value that should go in that position and puts it there.

The general strategy of the selection sort algorithm is as follows: Scan the entire list to find the smallest value. Exchange that value with the value in the first position of the list. Scan the rest of the list (all but the first value) to find the smallest value, and then exchange it with the value in the second position of the list. Scan the rest of the list (all but the first two values) to find the smallest value, and then exchange it with the value in the third position of the list. Continue this process for each position in the list. When complete, the list is sorted. The selection sort process is illustrated in Figure 9.3.

The following method defines an implementation of the selection sort algorithm. It accepts an array of objects as a parameter. When it returns to the calling method, the elements within the array are sorted.

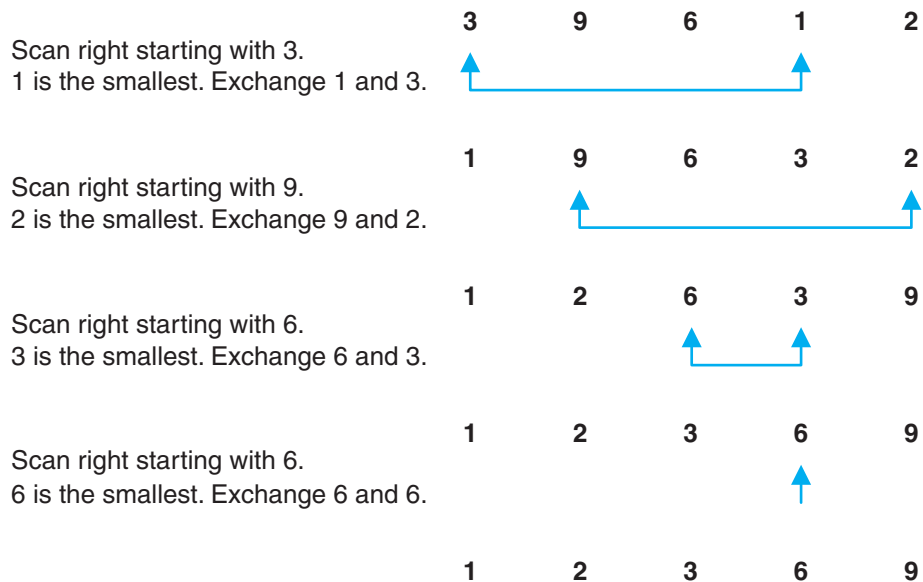


FIGURE 9.3 Example of selection sort processing

```

/**
 * Sorts the specified array of integers using the selection
 * sort algorithm.
 *
 * @param data the array to be sorted
 */
public static <T extends Comparable<T>>
    void selectionSort(T[] data)
{
    int min;
    T temp;

    for (int index = 0; index < data.length-1; index++)
    {
        min = index;
        for (int scan = index+1; scan < data.length; scan++)
            if (data[scan].compareTo(data[min])<0)
                min = scan;

        swap(data, min, index);
    }
}

```

The implementation of the `selectionSort` method uses two loops to sort an array. The outer loop controls the position in the array where the next smallest value will be stored. The inner loop finds the smallest value in the rest of the list by scanning all positions greater than or equal to the index specified by the outer loop. When the smallest value is determined, it is exchanged with the value stored at `index`. This exchange is accomplished by three assignment statements using an extra variable called `temp`. This type of exchange is called *swapping* and makes use of a private `swap` method. This method is also used by several of our other sorting algorithms.

```
/**
 * Swaps two elements in an array. Used by various sorting algorithms.
 *
 * @param data    the array in which the elements are swapped
 * @param index1  the index of the first element to be swapped
 * @param index2  the index of the second element to be swapped
 */
private static <T extends Comparable<T>>
    void swap(T[] data, int index1, int index2)
{
    T temp = data[index1];
    data[index1] = data[index2];
    data[index2] = temp;
}
```

Note that because this algorithm finds the smallest value during each iteration, the result is an array sorted in ascending order (that is, from smallest to largest). The algorithm can easily be changed to put values in descending order by finding the largest value each time.

Insertion Sort

The *insertion sort* algorithm sorts a list of values by repetitively inserting a particular value into a subset of the list that has already been sorted. One at a time, each unsorted element is inserted at the appropriate position in that sorted subset until the entire list is in order.

KEY CONCEPT

The insertion sort algorithm sorts a list of values by repetitively inserting a particular value into a subset of the list that has already been sorted.

The general strategy of the insertion sort algorithm is as follows: Sort the first two values in the list relative to each other by exchanging them if necessary. Insert the list's third value into the appropriate

position relative to the first two (sorted) values. Then insert the fourth value into its proper position relative to the first three values in the list. Each time an insertion is made, the number of values in the sorted subset increases by one. Continue this process until all values in the list are completely sorted. The insertion process requires that the other values in the array shift to make room for the inserted element. Figure 9.4 illustrates the insertion sort process.

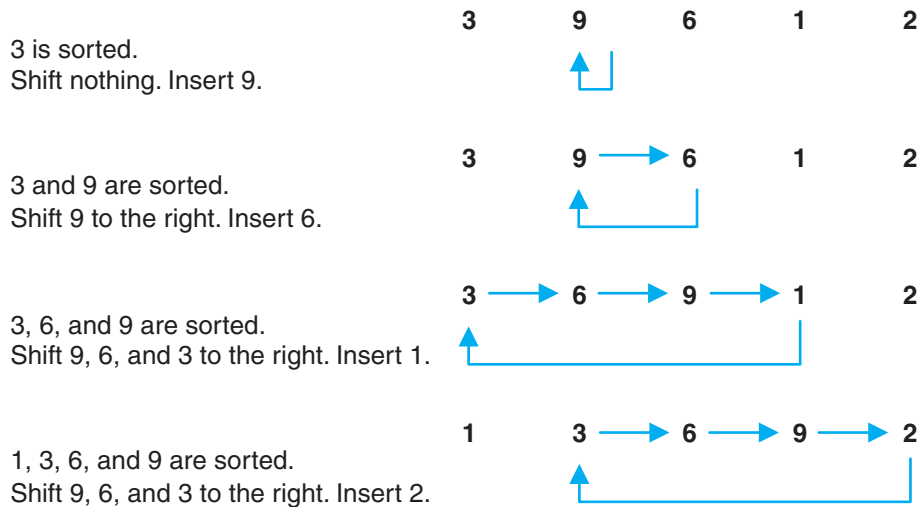


FIGURE 9.4 Example of insertion sort processing

The following method implements an insertion sort.

```
/**
 * Sorts the specified array of objects using an insertion
 * sort algorithm.
 *
 * @param data the array to be sorted
 */
public static <T extends Comparable<T>>
    void insertionSort(T[] data)
{
    for (int index = 1; index < data.length; index++)
    {
        T key = data[index];
```



```

    int position = index;

    // shift larger values to the right
    while (position > 0 && data[position-1].compareTo(key) > 0)
    {
        data[position] = data[position-1];
        position--;
    }

    data[position] = key;
}
}

```

Like the selection sort implementation, the `insertionSort` method uses two loops to sort an array of objects. In the insertion sort, however, the outer loop controls the index in the array of the next value to be inserted. The inner loop compares the current insert value with values stored at lower indexes (which make up a sorted subset of the entire list). If the current insert value is less than the value at `position`, then that value is shifted to the right. Shifting continues until the proper position is opened to accept the insert value. Each iteration of the outer loop adds one more value to the sorted subset of the list, until the entire list is sorted.

Bubble Sort

A *bubble sort* is another sequential sort algorithm that uses two nested loops. It sorts values by repeatedly comparing neighboring elements in the list and swapping their positions if they are not in order relative to each other.

KEY CONCEPT

The bubble sort algorithm sorts a list by repeatedly comparing neighboring elements and swapping them if necessary.

The general strategy of the bubble sort algorithm is as follows: Scan through the list comparing adjacent elements, and exchange them if they are not in relative order. This has the effect of “bubbling” the largest value to the last position in the list, which is its appropriate position in the final, sorted list. Then scan through the list again, bubbling up the second-to-last value. This process continues until all elements have been bubbled into their correct positions.

Each pass through the bubble sort algorithm moves the largest value to its final position. A pass may also reposition other elements as well. For example, if we started with the list

9 6 8 12 3 1 7

we would first compare 9 and 6 and, finding them not in the correct order, swap them, which yields

6 9 8 12 3 1 7

Then we would compare 9 to 8 and, again, finding them not in the correct order, swap them, which yields

6 8 9 12 3 1 7

Then we would compare 9 to 12. Since they are in the correct order, we don't swap them. Instead, we move to the next pair of values to compare. That is, we then compare 12 to 3. Because they are not in order, we swap them, which yields

6 8 9 3 12 1 7

We then compare 12 to 1 and swap them, which yields

6 8 9 3 1 12 7

We then compare 12 to 7 and swap them, which yields

6 8 9 3 1 7 12

This completes one pass through the data to be sorted. After this first pass, the largest value in the list (12) is in its correct position, but we cannot be sure about any of the other numbers. Each subsequent pass through the data guarantees that one more element is put into the correct position. Thus we make $n-1$ passes through the data, because if $n-1$ elements are in the correct, sorted positions, then the n th item must also be in the correct location.

An implementation of the bubble sort algorithm is shown in the following method:

```
/**
 * Sorts the specified array of objects using a bubble sort
 * algorithm.
 *
 * @param data the array to be sorted
 */
public static <T extends Comparable<T>>
    void bubbleSort(T[] data)
{
    int position, scan;
    T temp;
```

```

for (position = data.length - 1; position >= 0; position--)
{
    for (scan = 0; scan <= position - 1; scan++)
    {
        if (data[scan].compareTo(data[scan+1]) > 0)
            swap(data, scan, scan + 1);
    }
}
}

```

The outer `for` loop in the `bubbleSort` method represents the $n-1$ passes through the data. The inner `for` loop scans through the data, performs the pairwise comparisons of the neighboring data, and swaps them if necessary.

Note that the outer loop also has the effect of decreasing the position that represents the maximum index to examine in the inner loop. That is, after the first pass, which puts the last value in its correct position, there is no need to consider that value in future passes through the data. After the second pass, we can forget about the last two, and so on. Thus the inner loop examines one less value on each pass.

Quick Sort

The sort algorithms we have discussed thus far in this chapter (selection sort, insertion sort, and bubble sort) are relatively simple, but they are inefficient sequential sorts that use a pair of nested loops and require roughly n^2 comparisons to sort a list of n elements. Now we can turn our attention to more efficient sorts that lend themselves to a recursive implementation.

The *quick sort* algorithm sorts a list by partitioning the list using an arbitrarily chosen *partition element* and then recursively sorting the sublists on either side of the partition element. The general strategy of the quick sort algorithm is as follows: First, choose one element of the list to act as a partition element. Next, partition the list so that all elements less than the partition element are to the left of that element and all elements greater than the partition element are to the right. Finally, apply this quick sort strategy (recursively) to both partitions.

KEY CONCEPT

The quick sort algorithm sorts a list by partitioning the list and then recursively sorting the two partitions.

If the order of the data is truly random, the choice of the partition element is arbitrary. We will use the element in the middle of

the section we want to partition. For efficiency reasons, it is nice if the partition element happens to divide the list roughly in half, but the algorithm works no matter what element is chosen as the partition.

Let's look at an example of creating a partition. If we started with the list

305 65 7 90 120 110 8

we would choose 90 as our partition element. We would then rearrange the list, swapping the elements that are less than 90 to the left side and those that are greater than 90 to the right side, which would yield

8 65 7 90 120 110 305

We would then apply the quick sort algorithm separately to both partitions. This process continues until a partition contains only one element, which is inherently sorted. Thus, after the algorithm is applied recursively to either side, the entire list is sorted. Once the initial partition element is determined and placed, it is never considered or moved again.

The following method implements the quick sort algorithm. It accepts an array of objects to sort and the minimum and maximum index values used for a particular call to the method. Notice that the public method takes the array to be sorted and then calls the private method providing the array, the min, and the max.

```
/**
 * Sorts the specified array of objects using the quick sort algorithm.
 *
 * @param data the array to be sorted
 */
public static <T extends Comparable<T>>
    void quickSort(T[] data)
{
    quickSort(data, 0, data.length - 1);
}

/**
 * Recursively sorts a range of objects in the specified array using the
 * quick sort algorithm.
 *
 * @param data the array to be sorted
 * @param min the minimum index in the range to be sorted
 * @param max the maximum index in the range to be sorted
 */
```

```

private static <T extends Comparable<T>>
    void quickSort(T[] data, int min, int max)
{
    if (min < max)
    {
        // create partitions
        int indexofpartition = partition(data, min, max);
        // sort the left partition (lower values)
        quickSort(data, min, indexofpartition - 1);
        // sort the right partition (higher values)
        quickSort(data, indexofpartition + 1, max);
    }
}

```

The `quickSort` method relies heavily on the `partition` method, which it calls initially to divide the sort area into two partitions. The `partition` method returns the index of the partition value. Then the `quickSort` method is called twice (recursively) to sort the two partitions. The base case of the recursion, represented by the `if` statement in the `quickSort` method, is a list of one element or less, which is already inherently sorted. An example of the `partition` method follows.

```

/**
 * Used by the quick sort algorithm to find the partition.
 *
 * @param data the array to be sorted
 * @param min the minimum index in the range to be sorted
 * @param max the maximum index in the range to be sorted
 */
private static <T extends Comparable<T>>
    int partition(T[] data, int min, int max)
{
    T partitionelement;
    int left, right;
    int middle = (min + max)/2;

    // use the middle data value as the partition element
    partitionelement = data[middle];
    // move it out of the way for now

```

```
    swap(data, middle, min);
    left = min;
    right = max;

    while (left < right)
    {
        // search for an element that is > the partition element
        while (left < right && data[left].compareTo(partitionelement) <= 0)
            left++;

        // search for an element that is < the partition element
        while (data[right].compareTo(partitionelement) > 0)
            right--;

        // swap the elements
        if (left < right)
            swap(data, left, right);
    }

    // move the partition element into place
    swap(data, min, right);
    return right;
}
```

The two inner while loops of the `partition` method are used to find elements to swap that are in the wrong partitions. The first loop scans from left to right, looking for an element that is greater than the partition element. The second loop scans from right to left, looking for an element that is less than the partition element. When these two elements are found, they are swapped. This process continues until the right and left indexes meet in the “middle” of the list. The location where they meet also indicates where the partition element (which isn’t moved from its initial location until the end) will ultimately reside.

What happens if we get a poor partition element? If the partition element is near the smallest or the largest element in the list, then we effectively waste a pass through the data. One way to ensure a better partition element is to choose the middle of three elements. For example, the algorithm could check the first, middle, and last elements in the list and choose the middle value as

the partition element. This middle-of-three approach is left as a programming project.

Merge Sort

The *merge sort* algorithm, another recursive sort algorithm, sorts a list by recursively dividing the list in half until each sublist has one element and then recombining these sublists in order.

KEY CONCEPT

The merge sort algorithm sorts a list by recursively dividing the list in half until each sublist has one element and then merging these sublists into the sorted order.

The general strategy of the merge sort algorithm is as follows: Begin by dividing the list into two roughly equal parts and then recursively calling itself with each of those lists. Continue the recursive decomposition of the list until the base case of the recursion is reached, where the list is divided into lists of length one, which are by definition sorted. Then, as control passes back up the recursive calling structure, the algorithm merges the two sorted sublists resulting from the two recursive calls into one sorted list.

For example, if we started with the initial list from our example in the previous section, the recursive decomposition portion of the algorithm would yield the results shown in Figure 9.5.

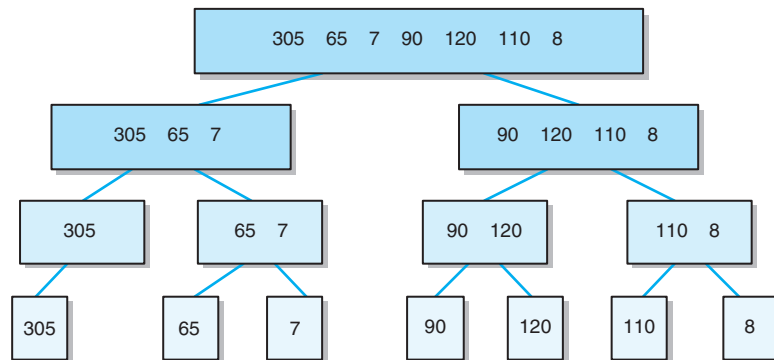


FIGURE 9.5 The decomposition of merge sort

The merge portion of the algorithm would then recombine the list as shown in Figure 9.6.

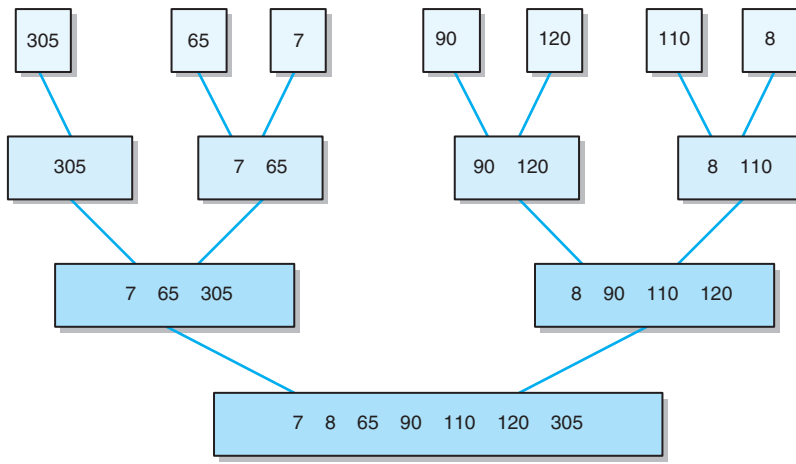


FIGURE 9.6 The merge portion of the merge sort algorithm

An implementation of the merge sort algorithm is shown below. Note that, just as for the quick sort algorithm, we make use of a public method that accepts the array to be sorted, and then a private method accepts the array as well as the min and max indexes of the section of the array to be sorted. This algorithm also make use of a private merge method to recombine the sorted sections of the array.

```
/**
 * Sorts the specified array of objects using the merge sort
 * algorithm.
 *
 * @param data the array to be sorted
 */
public static <T extends Comparable<T>>
    void mergeSort(T[] data)
{
    mergeSort(data, 0, data.length - 1);
}

/**
 * Recursively sorts a range of objects in the specified array using the
 * merge sort algorithm.
 *
 * @param data the array to be sorted
 * @param min the index of the first element
 * @param max the index of the last element
 */
```



```

private static <T extends Comparable<T>>
    void mergeSort(T[] data, int min, int max)
{
    if (min < max)
    {
        int mid = (min + max) / 2;
        mergeSort(data, min, mid);
        mergeSort(data, mid+1, max);
        merge(data, min, mid, max);
    }
}

```

```

/**
 * Merges two sorted subarrays of the specified array.
 *
 * @param data the array to be sorted
 * @param first the beginning index of the first subarray
 * @param mid the ending index of the first subarray
 * @param last the ending index of the second subarray
 */
@SuppressWarnings("unchecked")
private static <T extends Comparable<T>>
    void merge(T[] data, int first, int mid, int last)
{
    T[] temp = (T[])(new Comparable[data.length]);

    int first1 = first, last1 = mid; // endpoints of first subarray
    int first2 = mid+1, last2 = last; // endpoints of second subarray
    int index = first1; // next index open in temp array

    // Copy smaller item from each subarray into temp until one
    // of the subarrays is exhausted
    while (first1 <= last1 && first2 <= last2)
    {
        if (data[first1].compareTo(data[first2]) < 0)
        {
            temp[index] = data[first1];
            first1++;
        }
        else
        {
            temp[index] = data[first2];
            first2++;
        }
    }
}

```

```
        index++;
    }
    // Copy remaining elements from first subarray, if any
    while (first1 <= last1)
    {
        temp[index] = data[first1];
        first1++;
        index++;
    }

    // Copy remaining elements from second subarray, if any
    while (first2 <= last2)
    {
        temp[index] = data[first2];
        first2++;
        index++;
    }

    // Copy merged data into original array
    for (index = first; index <= last; index++)
        data[index] = temp[index];
}
```

9.3 Radix Sort

To this point, all of the sorting techniques we have discussed have involved comparing elements within the list to each other. As we have seen, the best of these comparison-based sorts is $O(n \log n)$. What if there were a way to sort elements without comparing them directly to each other? It might then be possible to build a more efficient sorting algorithm. We can find such a technique by revisiting our discussion of queues from Chapter 5.

A sort is based on some particular value, called the *sort key*. For example, a set of people might be sorted by their last name. A *radix sort*, rather than comparing items by sort key, is based on the structure of the sort key. Separate queues are created for each possible value of each digit or character of the sort key. The number of queues, or the number of possible values, is called the *radix*. For example, if we were sorting strings made up of lowercase alphabetic characters, the radix would be 26. We would use 26 separate queues, one for each possible character. If we were sorting decimal numbers, then the radix would be 10, one for each digit 0 through 9.

KEY CONCEPT

A radix sort is inherently based on queue processing.

Let's look at an example that uses a radix sort to put ten three-digit numbers in order. To keep things manageable, we will restrict the digits of these numbers to 0 through 5, which means we will need only six queues.

Each three-digit number to be sorted has a 1s position (right digit), a 10s position (middle digit), and a 100s position (left digit). The radix sort will make three passes through the values, one for each digit position. On the first pass, each number is put on the queue corresponding to its 1s digit. On the second pass, each number is put on the queue corresponding to its 10s digit. And finally, on the third pass, each number is put on the queue corresponding to its 100s digit.

Originally, the numbers are loaded into the queues from the original list. On the second pass, the numbers are taken from the queues in a particular order. They are retrieved from the digit 0 queue first, and then from the digit 1 queue, and so on. For each queue, the numbers are processed in the order in which they come off the queue. This processing order is crucial to the operation of a radix sort. Likewise, on the third pass, the numbers are again taken from the queues in the same way. When the numbers are pulled off the queues after the third pass, they will be completely sorted.

Figure 9.7 shows the processing of a radix sort for ten three-digit numbers. The number 442 is taken from the original list and put onto the queue corresponding to digit 2. Then 503 is put onto the queue corresponding to digit 3. Then 312 is put onto the queue corresponding to digit 2 (following 442). This continues for all values, resulting in the set of queues for the 1s position.

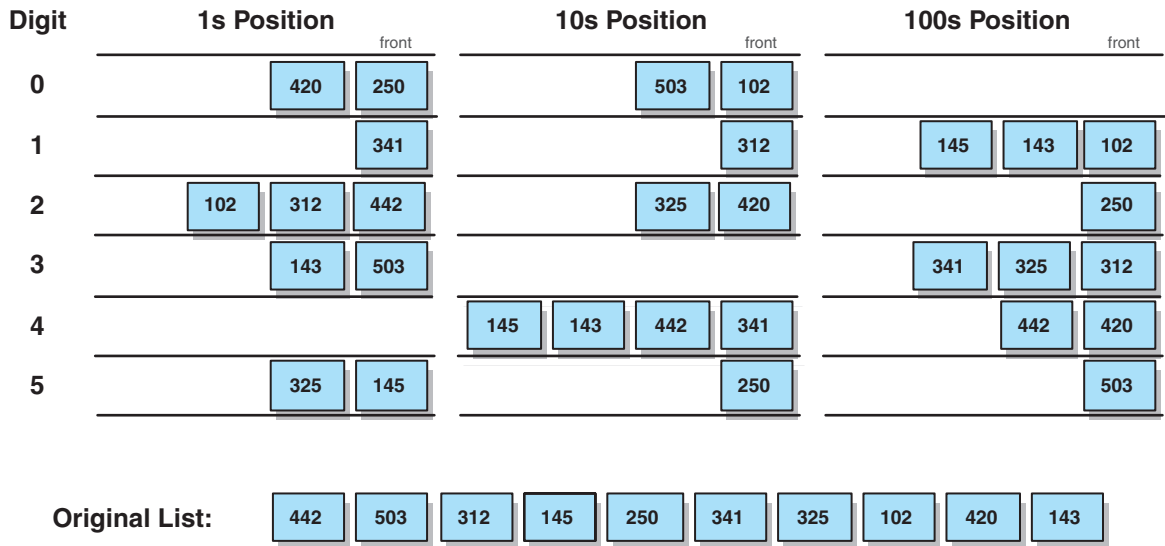


FIGURE 9.7 A radix sort of ten three-digit numbers

Assume, as we begin the second pass, that we have a fresh set of six empty digit queues. In actuality, the queues can be used over again if processed carefully. To begin the second pass, the numbers are taken from the 0 digit queue first. The number 250 is put onto the queue for digit 5, and then 420 is put onto the queue for digit 2. Then we can move to the next queue, taking 341 and putting it onto the queue for digit 4. This continues until all numbers have been taken off the 1s position queues, resulting in the set of queues for the 10s position.

For the third pass, the process is repeated. First, 102 is put onto the queue for digit 1, then 503 is put onto the queue for digit 5, and then 312 is put onto the queue for digit 3. This continues until we have the final set of digit queues for the 100s position. These numbers are now in sorted order if taken off each queue in turn.

Let's now look at a program that implements the radix sort. For this example, we will sort four-digit numbers, and we won't restrict the digits used in those numbers. Listing 9.3 shows the `RadixSort` class, which contains a single main method. Using an array of ten queue objects (one for each digit 0 through 9), this method carries out the processing steps of a radix sort. Figure 9.8 shows the UML description of the `RadixSort` class.

LISTING 9.3

```
import java.util.*;

/**
 * RadixSort driver demonstrates the use of queues in the execution of a radix sort.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class RadixSort
{
    /**
     * Performs a radix sort on a set of numeric values.
     */
    public static void main(String[] args)
    {
        int[] list = {7843, 4568, 8765, 6543, 7865, 4532, 9987, 3241,
                     6589, 6622, 1211};

        String temp;
```

LISTING 9.3 *continued*

```

Integer numObj;
int digit, num;

Queue<Integer>[] digitQueues = (LinkedList<Integer>[]) (new
LinkedList[10]);
for (int digitVal = 0; digitVal <= 9; digitVal++)
    digitQueues[digitVal] = (Queue<Integer>) (new LinkedList<Integer>());

// sort the list
for (int position=0; position <= 3; position++)
{
    for (int scan=0; scan < list.length; scan++)
    {
        temp = String.valueOf(list[scan]);
        digit = Character.digit(temp.charAt(3-position), 10);
        digitQueues[digit].add(new Integer(list[scan]));
    }

    // gather numbers back into list
    num = 0;
    for (int digitVal = 0; digitVal <= 9; digitVal++)
    {
        while (!(digitQueues[digitVal].isEmpty()))
        {
            numObj = digitQueues[digitVal].remove();
            list[num] = numObj.intValue();
            num++;
        }
    }

    // output the sorted list
    for (int scan=0; scan < list.length; scan++)
        System.out.println(list[scan]);
}
}

```

In the RadixSort program, the numbers are originally stored in an array called `list`. After each pass, the numbers are pulled off the queues and stored back into the `list` array in the proper order. This allows the program to reuse the original array of ten queues for each pass of the sort.

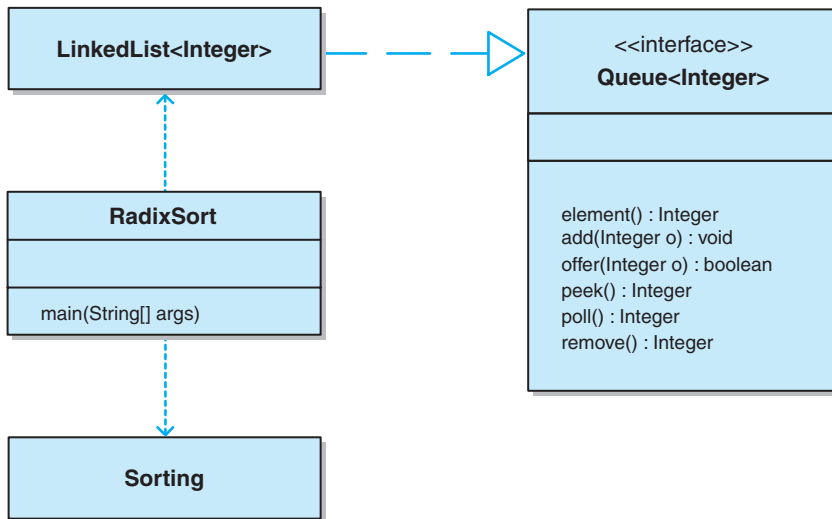


FIGURE 9.8 UML description of the RadixSort program

The concept of a radix sort can be applied to any type of data as long as the sort key can be dissected into well-defined positions. Note that, unlike the sorts we discussed earlier in this chapter, it's not reasonable to create a generic radix sort for any object, because dissecting the key values is an integral part of the process.

So what is the time complexity of a radix sort? In this case, there is not any comparison or swapping of elements. Elements are simply removed from a queue and placed in another one on each pass. For any given radix, the number of passes through the data is a constant based on the number of characters in the key; let's call it c . Then the time complexity of the algorithm is simply $c \cdot n$. Keep in mind, from our discussion in Chapter 2, that we ignore constants when computing the time complexity of an algorithm. Thus the radix sort algorithm is $O(n)$. So why not use radix sort for all of our sorting? First, each radix sort algorithm has to be designed specifically for the key of a given problem. Second, for keys where the number of digits in the key (c) and the number of elements in the list (n) are very close together, the actual time complexity of the radix sort algorithm mimics n^2 instead of n . In addition, we also need to keep in mind that there is another constant that affects space complexity; it is the radix, or the number of possible values for each position or character in the key. Imagine, for example, trying to implement a radix sort for a key that allows any character from the Unicode character set. Because this set has more than 100,000 characters, we would need that many queues!

Summary of Key Concepts

- Searching is the process of finding a designated target within a group of items or determining that the target doesn't exist.
- An efficient search minimizes the number of comparisons made.
- A method is made static by using the static modifier in the method declaration.
- A binary search capitalizes on the fact that the search pool is sorted.
- A binary search eliminates half of the viable candidates with each comparison.
- A binary search has logarithmic complexity, which makes it very efficient for a large search pool.
- Sorting is the process of arranging a list of items into a defined order based on some criterion.
- The selection sort algorithm sorts a list of values by repetitively putting a particular value into its final, sorted position.
- The insertion sort algorithm sorts a list of values by repetitively inserting a particular value into a subset of the list that has already been sorted.
- ☒ The bubble sort algorithm sorts a list by repeatedly comparing neighboring elements and swapping them if necessary.
- The quick sort algorithm sorts a list by partitioning the list and then recursively sorting the two partitions.
- The merge sort algorithm sorts a list by recursively dividing the list in half until each sublist has one element and then merging these sublists into the sorted order.
- A radix sort is inherently based on queue processing.

Summary of Terms

binary search A search that occurs on a sorted list and in which each comparison eliminates approximately half of the remaining viable candidates.

bubble sort A sorting algorithm that sorts elements by repeatedly comparing adjacent values and swapping them.

class method See *static method*.

generic method A method that includes the definition of a type parameter in the header of the method.

insertion sort A sorting algorithm that sorts elements by repetitively inserting a particular element into a previously sorted sublist.

linear search A search that begins at one end of a list of items and continues linearly until the element is found or the end of the list is reached.

logarithmic algorithm An algorithm that has a time complexity of $O(\log_2 n)$, such as a binary search.

logarithmic sort A sorting algorithm that requires approximately $n \log_2 n$ comparisons to sort n elements.

merge sort A sorting algorithm that sorts elements by recursively dividing the list in half until each sublist has one element and then merging the sublists.

partition A set of unsorted elements, used by the quick sort algorithm, that are all either less than or greater than a chosen partition element.

partition element An element used by the quick sort algorithm to separate unsorted elements into two distinct partitions.

quick sort A sorting algorithm that sorts elements by partitioning the unsorted elements into two partitions and then recursively sorting each partition.

radix sort A sorting algorithm that sorts elements using a sort key instead of directly comparing elements.

searching The process of finding a designated target element within a group of elements, or determining that the target is not in the group.

search pool A group of items to be searched.

selection sort A sorting algorithm that sorts elements by repetitively finding a particular element and putting it in its final position.

sequential sort A sorting algorithm that typically uses nested loops and requires approximately n^2 comparisons to sort n elements.

sorting The process of arranging a group of items into a particular order based on some criterion.

static method A method that is invoked through the class name and that cannot refer to instance data. Also called a class method.

target element The element that is being sought during a search operation.

viable candidates The elements in a search pool among which the target element may still be found.

Self-Review Questions

SR 9.1 When would a linear search be preferable to a logarithmic search?

SR 9.2 Which searching method requires that the list be sorted?

SR 9.3 When would a sequential sort be preferable to a recursive sort?

- SR 9.4 The insertion sort algorithm sorts using what technique?
- SR 9.5 The bubble sort algorithm sorts using what technique?
- SR 9.6 The selection sort algorithm sorts using what technique?
- SR 9.7 The quick sort algorithm sorts using what technique?
- SR 9.8 The merge sort algorithm sorts using what technique?
- SR 9.9 How many queues would it take to use a radix sort to sort names stored as all lowercase?

Exercises

- EX 9.1 Compare and contrast the `linearSearch` and `binarySearch` algorithms by searching for the numbers 45 and 54 in the list 3, 8, 12, 34, 54, 84, 91, 110.
- EX 9.2 Using the list from Exercise 9.1, construct a table showing the number of comparisons required to sort that list for each of the sort algorithms (selection sort, insertion sort, bubble sort, quick sort, and merge sort).
- EX 9.3 Consider the same list from Exercise 9.1. What happens to the number of comparisons for each of the sort algorithms if the list is already sorted?
- EX 9.4 Consider the following list:
- 90 8 7 56 123 235 9 1 653
- Show a trace of execution for:
- selection sort
 - insertion sort
 - bubble sort
 - quick sort
 - merge sort
- EX 9.5 Given the resulting sorted list from Exercise 9.4, show a trace of execution for a binary search, searching for the number 235.
- EX 9.6 Draw the UML description of the `SortPhoneList` example.
- EX 9.7 Hand trace a radix sort for the following list of five-digit student ID numbers:
- 13224
32131
54355

12123
22331
21212
33333
54312

EX 9.8 What is the time complexity of a radix sort?

Programming Projects

- PP 9.1 The bubble sort algorithm shown in this chapter is less efficient than it could be. If a pass is made through the list without exchanging any elements, this means that the list is sorted and there is no reason to continue. Modify this algorithm so that it will stop as soon as it recognizes that the list is sorted. *Do not* use a `break` statement!
- PP 9.2 There is a variation of the bubble sort algorithm called a *gap sort* that, rather than comparing neighboring elements each time through the list, compares elements that are i positions apart, where i is an integer less than n . For example, the first element would be compared to the $(i+1)$ element, the second element would be compared to the $(i+2)$ element, the n th element would be compared to the $(n-i)$ element, and so on. A single iteration is completed when all of the elements that can be compared have been compared. On the next iteration, i is reduced by some number greater than 1, and the process continues until i is less than 1. Implement a gap sort.
- PP 9.3 Modify the sorts listed in the chapter (selection sort, insertion sort, bubble sort, quick sort, and merge sort) by adding code to each to tally the total number of comparisons and total execution time of each algorithm. Execute the sort algorithms against the same list, recording information for the total number of comparisons and total execution time for each algorithm. Try several different lists, including at least one that is already in sorted order.
- PP 9.4 Modify the quick sort method to choose the partition element using the middle-of-three technique described in the chapter. Run this new version against the old version for several sets of data, and compare the total execution times.

Answers to Self-Review Questions

- SRA 9.1 A linear search would be preferable for relatively small, unsorted lists and in languages where recursion is not supported.
- SRA 9.2 Binary search.
- SRA 9.3 A sequential sort would be preferable for relatively small data sets and in languages where recursion is not supported.
- SRA 9.4 The insertion sort algorithm sorts a list of values by repetitively inserting a particular value into a subset of the list that has already been sorted.
- SRA 9.5 The bubble sort algorithm sorts a list by repeatedly comparing neighboring elements in the list and swapping their positions if they are not already in order.
- SRA 9.6 The selection sort algorithm, which is an $O(n^2)$ sort algorithm, sorts a list of values by repetitively putting a particular value into its final, sorted position.
- SRA 9.7 The quick sort algorithm sorts a list by partitioning the list using an arbitrarily chosen partition element and then recursively sorting the sublists on either side of the partition element.
- SRA 9.8 The merge sort algorithm sorts a list by recursively dividing the list in half until each sublist has one element and then recombining these sublists in order.
- SRA 9.9 It would require 27 queues, one for each of the 26 letters in the alphabet and one to store the whole list before, during, and after sorting.



Trees

10

This chapter begins our exploration of nonlinear collections and data structures. We discuss the use and implementation of trees, define the terms associated with trees, analyze possible tree implementations, and look at examples of implementing and using trees.

CHAPTER OBJECTIVES

- Define trees as data structures.
- Define the terms associated with trees.
- Discuss the possible implementations of trees.
- Analyze tree implementations of collections.
- Discuss methods for traversing trees.
- Examine a binary tree example.

10.1 Trees

KEY CONCEPT

A tree is a nonlinear structure whose elements are organized into a hierarchy.

The collections we have examined up to this point (stacks, queues, and lists) are all linear data structures, which means that their elements are arranged in order, one after another. A *tree* is a nonlinear structure in which elements are organized into a hierarchy. This section describes trees in general and establishes some important terminology.

Conceptually, a tree is composed of a set of *nodes* in which elements are stored and *edges* that connect one node to another. Each node is at a particular *level* in the tree hierarchy. The *root* of the tree is the only node at the top level of the tree. There is only one root node in a tree. Figure 10.1 shows a tree that helps to illustrate these terms.

KEY CONCEPT

Trees are described by a large set of related terms.

The nodes at lower levels of the tree are the *children* of nodes at the previous level. In Figure 10.1, the nodes labeled B, C, D, and E are the children of A. Nodes F and G are the children of B. A node can have only one parent, but a node may have multiple children. Nodes that have the same parent are called *siblings*. Thus, nodes H, I, and J are siblings because they are all children of node D.

The root node is the only node in a tree that does not have a parent. A node that does not have any children is called a *leaf*. A node that is not the root and has at least one child is called an *internal node*. Note that the tree analogy is upside-down. Our trees “grow” from the root at the top of the tree to the leaves toward the bottom of the tree.

The root is the entry point into a tree structure. We can follow a *path* through the tree from parent to child. For example, the path from node A to node N in Figure 10.1 is A, D, I, N. A node is the *ancestor* of another node if it is above it on

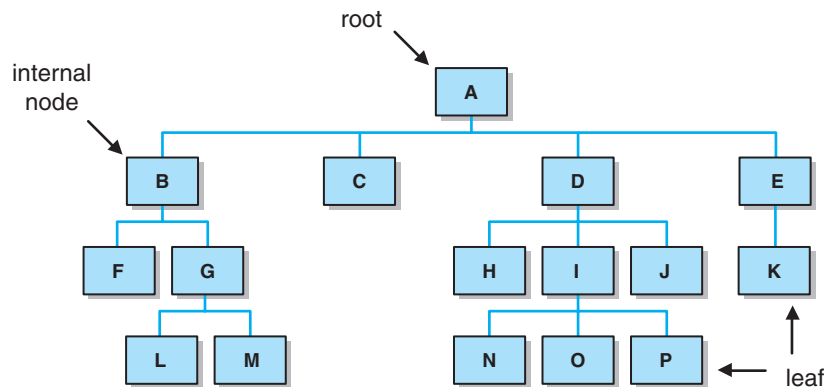


FIGURE 10.1 Tree terminology

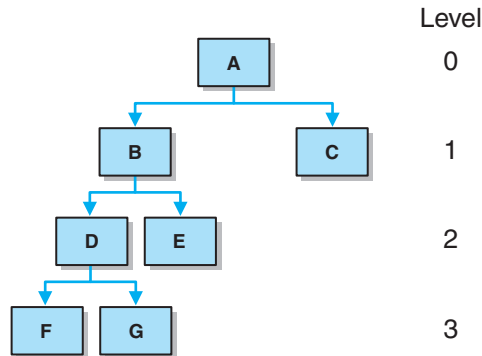


FIGURE 10.2 Path length and level

the path from the root. Thus the root is the ultimate ancestor of all nodes in a tree. Nodes that can be reached by following a path from a particular node are the *descendants* of that node.

The level of a node is also the length of the path from the root to the node. This *path length* is determined by counting the number of edges that must be followed to get from the root to the node. The root is considered to be level 0, the children of the root are at level 1, the grandchildren of the root are at level 2, and so on. Path length and level are depicted in Figure 10.2.

The *height* of a tree is the length of the longest path from the root to a leaf. Thus the height of the tree in Figure 10.2 is 3, because the path length from the root to leaves F and G is 3. The path length from the root to leaf C is 1.

Tree Classifications

Trees can be classified in many ways. The most important criterion is the maximum number of children any node in the tree may have. This value is sometimes referred to as the *order* of the tree. A tree that has no limit to the number of children a node may have is called a *general tree*. A tree that limits each node to no more than n children is referred to as an *n -ary tree*.

One n -ary tree is of particular importance. A tree in which nodes may have at most two children is called a *binary tree*. This type of tree is helpful in many situations. Much of our exploration of trees will focus on binary trees.

Another way to classify a tree is in terms of whether or not it is balanced. There are many definitions of balance, depending on the algorithms being used. We will explore some of these algorithms in the next chapter. Roughly speaking, a tree is considered *balanced* if all the leaves of the tree are on the same level or at least within one level of each other. Thus, the tree shown on the left in Figure 10.3 is balanced, and the one on the right is not. A balanced n -ary tree with m elements

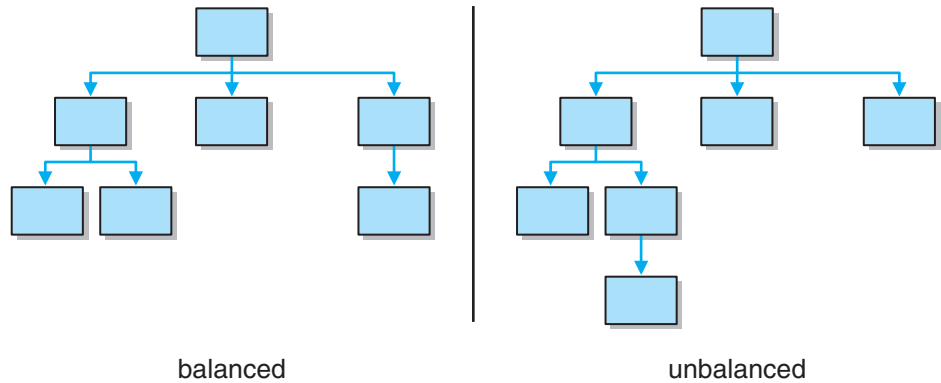


FIGURE 10.3 A balanced tree and an unbalanced tree

has a height of $\log_n m$. Thus a balanced binary tree with n nodes has a height of $\log_2 n$.

The concept of a complete tree is related to the balance of a tree. A tree is considered *complete* if it is balanced and all of the leaves at the bottom level are on the left side of the tree. Although it seems arbitrary, this definition has implications for how the tree is stored in certain implementations. Another way to express this concept is to say that a complete binary tree has 2^k nodes at every level k except the last, where the nodes must be leftmost.

A related concept is the notion of a full tree. An n -ary tree is considered *full* if all the leaves of the tree are at the same level and every node either is a leaf or has exactly n children. The balanced tree in Figure 10.3 is not considered complete. Among the 3-ary (or tertiary) trees shown in Figure 10.4, the trees in parts (a) and (c) are complete, but only the tree in part (c) is full.

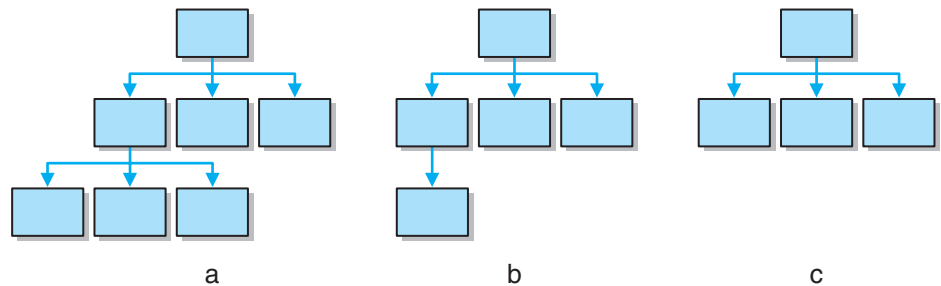


FIGURE 10.4 Some complete trees

10.2 Strategies for Implementing Trees

Let's examine some general strategies for implementing trees. The most obvious implementation of a tree is a linked structure. Each node could be defined as a `TreeNode` class, similar to what we did with the `LinearNode` class for linked lists. Each node would contain a pointer to the element to be stored in that node, as well as pointers for each of the possible children of the node. Depending on the implementation, it may also be useful for each node to store a pointer to its parent. This use of pointers is similar to the concept of a doubly linked list, where each node points not only to the next node in the list but to the previous node as well.

Another possibility would be to implement a tree recursively using links. This strategy would involve defining each node as a tree with attributes for each of its children. Thus each node, and all of its descendants, represents a tree unto itself. The implementation of this strategy is left as a programming project.

Because a tree is a nonlinear structure, it may not seem reasonable to try to implement it using an underlying linear structure such as an array. However, sometimes that approach is useful. The strategies for array implementation of a tree may be less obvious. There are two principal approaches: a computational strategy and a simulated link strategy.

Computational Strategy for Array Implementation of Trees

For certain types of trees, specifically binary trees, a computational strategy can be used for storing a tree using an array. One possible strategy is as follows: For any element stored in position n of the array, that element's left child will be stored in position $(2 * n + 1)$ and that element's right child will be stored in position $(2 * (n + 1))$. This strategy is very effective and can be managed in terms of capacity in much the same way as managing capacity for the array implementations of lists, queues, and stacks. However, despite the conceptual elegance of this solution, it is not without drawbacks. For example, if the tree that we are storing is not complete or is only relatively complete, we may be wasting large amounts of memory allocated in the array for positions of the tree that do not contain data. The computational strategy is illustrated in Figure 10.5.

KEY CONCEPT

One possible computational strategy places the left child of element n at position $(2 * n + 1)$ and the right child at position $(2 * (n + 1))$.

Simulated Link Strategy for Array Implementation of Trees

A second possible array implementation of trees is modeled after the way operating systems manage memory. Instead of assigning elements of the tree to array

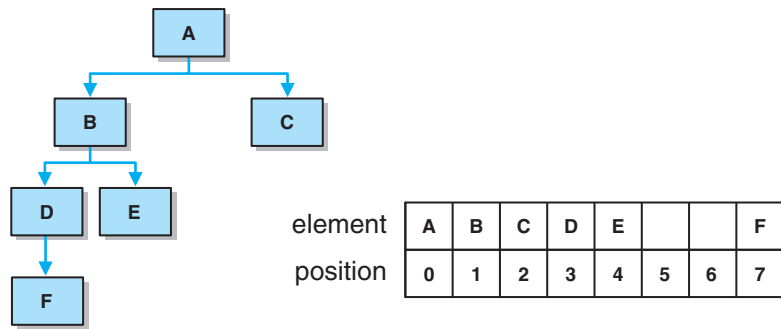


FIGURE 10.5 Computational strategy for array implementation of trees

positions by location in the tree, array positions are allocated contiguously on a first-come, first-served basis. Each element of the array will be a node class similar to the `TreeNode` class that we discussed earlier. However, instead of storing object reference variables as pointers to its children (and perhaps its parent), each node would store the array index of each child (and perhaps its parent). This approach allows elements to be stored contiguously in the array so that space is not wasted.

KEY CONCEPT

The simulated link strategy allows array positions to be allocated contiguously, regardless of the completeness of the tree.

However, this approach increases the overhead for deleting elements in the tree, because it requires either that remaining elements be shifted to maintain contiguity or that a *freelist* be maintained. This strategy is illustrated in Figure 10.6. The order of the elements in the array is determined simply by their entry order into the tree. In this case, the entry order is assumed to have been A, C, B, E, D, F.

This same strategy may also be used when tree structures need to be stored directly on disk using a direct I/O approach. In this case, rather than using an array index as a pointer, each node will store the relative position in the file of its children so that an offset can be calculated given the base address of the file.

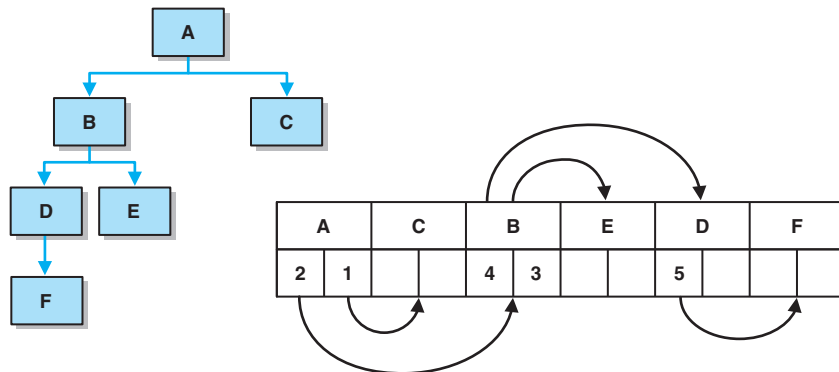


FIGURE 10.6 Simulated link strategy for array implementation of trees

**DESIGN FOCUS**

When does it begin to make sense to define an ADT for a collection? At this point, we have defined many of the terms for a tree, and we have a general understanding of how a tree might be used, but are we ready to define an ADT? Not really. Trees, in the general sense, are more of an abstract data structure than a collection, so trying to define an ADT for a general tree is not likely to be very useful. Instead, we will wait until we have specified more details about the type of the tree and its use before we attempt to define an interface.

Analysis of Trees

As we noted earlier, trees are a useful and efficient way to implement other collections. Let's consider an ordered list as an example. In our analysis of list implementations in Chapter 6, we described the `find` operation as having efficiency $n/2$ or $O(n)$. However, if we were to implement an ordered list using a balanced *binary search tree*—a binary tree with the added property that the left child is always less than the parent, which is always less than or equal to the right child—then we could improve the efficiency of the `find` operation to $O(\log n)$. We will discuss binary search trees in much greater detail in Chapter 11.

This increased efficiency is due to the fact that the height of such a tree will always be $\log_2 n$, where n is the number of elements in the tree. This is very similar to our discussion of the binary search in Chapter 9. In fact, for any balanced n -ary tree with m elements, the tree's height will be $\log_n m$. With the added ordering property of a binary search tree, you are guaranteed to search, at worst, one path from the root to a leaf, and that path can be no longer than $\log_n m$.

KEY CONCEPT

In general, a balanced n -ary tree with m elements will have height $\log_n m$.

**DESIGN FOCUS**

If trees provide more efficient implementations than linear structures, why would we ever use linear structures? There is an overhead associated with trees in terms of maintaining the structure and order of the tree that may not be present in other structures; thus there is a trade-off between this overhead and the size of the problem. With a relatively small n , the difference between the analysis of tree implementations and that of linear structures is not particularly significant relative to the overhead involved in the tree. However, as n increases, the efficiency of a tree becomes more attractive.

10.3 Tree Traversals

Because a tree is a nonlinear structure, the concept of traversing a tree is generally more interesting than the concept of traversing a linear structure. There are four basic methods for traversing a tree:

- *Preorder traversal*, which is accomplished by visiting each node, followed by its children, starting with the root
- *Inorder traversal*, which is accomplished by visiting the left child of the node, then the node, and then any remaining nodes, starting with the root
- *Postorder traversal*, which is accomplished by visiting the children and then the node, starting with the root
- *Level-order traversal*, which is accomplished by visiting all of the nodes at each level, one level at a time, starting with the root

KEY CONCEPT

There are four basic methods for traversing a tree: preorder, inorder, postorder, and level-order traversals.

Each of these definitions applies to all trees. However, as an example, let us examine how each of these definitions would apply to a binary tree (that is, a tree in which each node has at most two children).

Preorder Traversal

Given the tree shown in Figure 10.7, a preorder traversal would produce the sequence A, B, D, E, C. The definition stated previously says that preorder traversal is accomplished by visiting each node, followed by its children, starting with the root. So, starting with the root, we visit the root, giving us A. Next we traverse to the first child of the root, which is the node containing B. We then use the same algorithm by first visiting the current node, which yields B, and then visiting its children. Next we traverse to the first child of B, which is the node containing D. We then use the same algorithm again by first visiting the current node, which yields D, and then visiting its children. Only this time, there are no children. We then traverse to any other children of B. This yields E, and because E has no children, we then traverse to any other children of A. This brings us to the node containing C, where we again use the same algorithm, first visiting the node, which yields C, and then visiting any children. Because there are no children of C and no more children of A, the traversal is complete.

KEY CONCEPT

Preorder traversal means visit the node, then the left child, and then the right child.

Stated in pseudocode for a binary tree, the algorithm for a preorder traversal is

```
Visit node
Traverse(left child)
Traverse(right child)
```

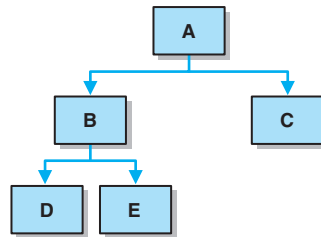


FIGURE 10.7 A complete tree

Inorder Traversal

Given the tree shown in Figure 10.7, an inorder traversal would produce the sequence D, B, E, A, C. As defined earlier, inorder traversal is accomplished by visiting the left child of the node, then the node, and then any remaining nodes, starting with the root. So, starting with the root, we traverse to the left child of the root, the node containing B. We then use the same algorithm again and traverse to the left child of B, the node containing D. Note that we have not yet visited any nodes. Using the same algorithm again, we attempt to traverse to the left child of D. Because there is no such left child, we then visit the current node, which yields D. Continuing the same algorithm, we attempt to traverse to any remaining children of D. Because there are no children, we then visit the previous node, which yields B. We then attempt to traverse to any remaining children of B. This brings us to the node containing E. Because E does not have a left child, we visit the node, which yields E. Because E has no right child, we then visit the previous node, which yields A. We then traverse to any remaining children of A, which takes us to the node containing C. Using the same algorithm, we then attempt to traverse to the left child of C. Because there is no such left child, we then visit the current node, which yields C. We then attempt to traverse to any remaining children of C. Because there are no such children, we return to the previous node, which happens to be the root. Because there are no more children of the root, the traversal is complete.

Stated in pseudocode for a binary tree, the algorithm for an inorder traversal is

```
Traverse(left child)
Visit node
Traverse(right child)
```

Postorder Traversal

Given the tree shown in Figure 10.7, a postorder traversal would produce the sequence D, E, B, C, A. As previously defined, postorder traversal is accomplished

KEY CONCEPT

Inorder traversal means visit the left child, then the node, and then the right child.

KEY CONCEPT

Postorder traversal means visit the left child, then the right child, and then the node.

by visiting the children and then the node, starting with the root. So, starting from the root, we traverse to the left child, the node containing B. Repeating that process, we traverse to the left child again, the node containing D. Because that node does not have any children, we then visit that node, which yields D. Returning to the previous node, we visit the right child, the node containing E. Because this node does not have any children, we visit the node, which yields E, and then return to the previous node and visit it, which yields B. Returning to the previous node, in this case the root, we find that it has a right child, so we traverse to the right child, the node containing C. Because this node does not have any children, we visit it, which yields C. Returning to the previous node (the root), we find that it has no remaining children, so we visit it, which yields A, and the traversal is complete.

Stated in pseudocode for a binary tree, the algorithm for a postorder traversal is

```

Traverse(left child)
Traverse(right child)
Visit node

```

Level-Order Traversal

Given the tree shown in Figure 10.7, a level-order traversal would produce the sequence A, B, C, D, E. As defined earlier, a level-order traversal is accomplished by visiting all of the nodes at each level, one level at a time, starting with the root. Using this definition, we first visit the root, which yields A. Next we visit the left child of the root, which yields B, then the right child of the root, which yields C, and then the children of B, which yields D and E.

Stated in pseudocode for a binary tree, an algorithm for a level-order traversal is

```

Create a queue called nodes
Create an unordered list called results
Enqueue the root onto the nodes queue
While the nodes queue is not empty
{
    Dequeue the first element from the queue
    If that element is not null
        Add that element to the rear of the results list
        Enqueue the children of the element on the nodes queue
    Else
        Add null on the result list
}
Return an iterator for the result list

```

This algorithm for a level-order traversal is only one of many possible solutions. However, it does have some interesting properties. First, note that we are

using collections—namely a queue and a list—to solve a problem within another collection—namely a binary tree. Second, recall that in our earlier discussions of iterators, we talked about their behavior with respect to the collection if the collection is modified while the iterator is in use. In this case, using a list to store the elements in the proper order and then returning an iterator over the list, this iterator behaves like a snapshot of the binary tree and is not affected by any concurrent modifications. This can be either a positive or a negative attribute, depending on how the iterator is used.

KEY CONCEPT

Level-order traversal means visit the nodes at each level, one level at a time, starting with the root.



VideoNote

Demonstration of the four basic tree traversals

10.4 A Binary Tree ADT

Let's take a look at a simple binary tree implementation using links. In Sections 10.5 and 10.6, we will consider examples using this implementation. As we discussed earlier in this chapter, it is difficult to abstract an interface for all trees. However, once we have narrowed our focus to binary trees, the task becomes more reasonable. One possible set of operations for a binary tree ADT is listed in Figure 10.8. Keep in mind that the definition of a collection is not universal. You will find variations in the operations defined for specific collections from one text to another. We have been very careful in this text to define the operations on each collection so that they are consistent with its purpose.

Notice that in all of the operations listed, there are no operations to add elements to the tree or remove elements from it. This is because until we specify the purpose and organization of the binary tree, there is no way to know how or—

Operation	Description
<code>getRoot</code>	Returns a reference to the root of the binary tree
<code>isEmpty</code>	Determines whether the tree is empty
<code>size</code>	Returns the number of elements in the tree
<code>contains</code>	Determines whether the specified target is in the tree
<code>find</code>	Returns a reference to the specified target element if it is found
<code>toString</code>	Returns a string representation of the tree
<code>iteratorInOrder</code>	Returns an iterator for an inorder traversal of the tree
<code>iteratorPreOrder</code>	Returns an iterator for a preorder traversal of the tree
<code>iteratorPostOrder</code>	Returns an iterator for a postorder traversal of the tree
<code>iteratorLevelOrder</code>	Returns an iterator for a level-order traversal of the tree

FIGURE 10.8 The operations on a binary tree

more specifically—where to add an element to the tree. Similarly, any operation to remove one or more elements from the tree may violate the purpose or structure of the tree as well. As with adding an element, we do not yet have enough information to know how to remove an element. When we were dealing with stacks in Chapters 3 and 4, we could think about the concept of removing an element from a stack, and it was easy to conceptualize the state of the stack after removal of the element. The same can be said of queues, because we could remove an element from only one end of the linear structures. Even with lists, where we could remove an element from the middle of the linear structure, it was easy to conceptualize the state of the resulting list.

With a tree, however, upon removing an element, we have many issues to handle that will affect the state of the tree. What happens to the children and other descendants of the element that is removed? Where does the child pointer of the element's parent now point? What if the element we are removing is the root? As we will see in our example using expression trees later in this chapter, there will be applications of trees where there is no concept of the removal of an element from the tree. Once we have specified more detail about the use of the tree, we may then decide that a `removeElement` method is appropriate. An excellent example of this is binary search trees, as we will see in Chapter 11.

Listing 10.1 shows the `BinaryTreeADT` interface. Figure 10.9 shows the UML description for the `BinaryTreeADT` interface.

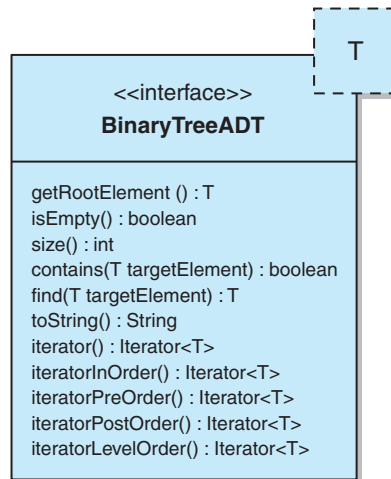


FIGURE 10.9 UML description of the `BinaryTreeADT` interface

LISTING 10.1

```
package jsjf;

import java.util.Iterator;

/**
 * BinaryTreeADT defines the interface to a binary tree data structure.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public interface BinaryTreeADT<T>
{
    /**
     * Returns a reference to the root element
     *
     * @return a reference to the root
     */
    public T getRootElement();

    /**
     * Returns true if this binary tree is empty and false otherwise.
     *
     * @return true if this binary tree is empty, false otherwise
     */
    public boolean isEmpty();

    /**
     * Returns the number of elements in this binary tree.
     *
     * @return the number of elements in the tree
     */
    public int size();

    /**
     * Returns true if the binary tree contains an element that matches
     * the specified element and false otherwise.
     *
     * @param targetElement the element being sought in the tree
     * @return true if the tree contains the target element
     */
    public boolean contains(T targetElement);

    /**
     * Returns a reference to the specified element if it is found in
```


LISTING 10.1 *continued*

```

* this binary tree. Throws an exception if the specified element
* is not found.
*
* @param targetElement the element being sought in the tree
* @return a reference to the specified element
*/
public T find(T targetElement);

/**
 * Returns the string representation of this binary tree.
 *
 * @return a string representation of the binary tree
 */
public String toString();

/**
 * Returns an iterator over the elements of this tree.
 *
 * @return an iterator over the elements of this binary tree
 */
public Iterator<T> iterator();

/**
 * Returns an iterator that represents an inorder traversal on this
 * binary tree.
 * @return an iterator over the elements of this binary tree
 */
public Iterator<T> iteratorInOrder();

/**
 * Returns an iterator that represents a preorder traversal on this
 * binary tree.
 * @return an iterator over the elements of this binary tree
 */
public Iterator<T> iteratorPreOrder();

/**
 * Returns an iterator that represents a postorder traversal on this
 * binary tree.
 * @return an iterator over the elements of this binary tree
 */

```

LISTING 10.1 *continued*

```

public Iterator<T> iteratorPostOrder();

/**
 * Returns an iterator that represents a levelorder traversal on this
 * binary tree.
 * @return an iterator over the elements of this binary tree
 */
public Iterator<T> iteratorLevelOrder();
}

```

10.5 Using Binary Trees: Expression Trees

In Chapter 3, we used a stack algorithm to evaluate postfix expressions. In this section, we modify that algorithm to construct an expression tree using an `ExpressionTree` class that extends our definition of a binary tree. Figure 10.10 illustrates the concept of an expression tree. Notice that the root and all of the internal nodes of an expression tree contain operations and that all of the leaves contain operands. An expression tree is evaluated from the bottom up. In this case, the $(5 - 3)$ term is evaluated first, which yields 2. That result is then multiplied by 4, which yields 8. Finally, the result of that term is added to 9, which yields 17.

Listing 10.2 illustrates our `ExpressionTree` class. The Java Collections API does not provide an implementation of a tree collection. Instead, the use of trees in the API is limited to their use as an implementation strategy for sets and maps.

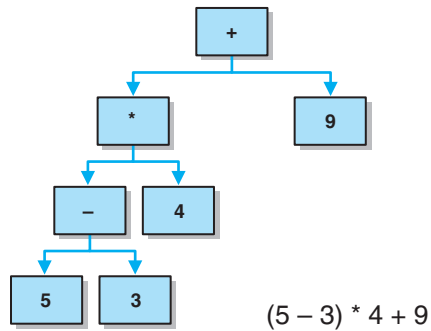


FIGURE 10.10 An example of an expression tree

Thus we will use our own implementation of a linked binary tree for this example. The `LinkedBinaryTree` class is presented in Section 10.7.

The `ExpressionTree` class extends the `LinkedBinaryTree` class, providing a new constructor that will combine expression trees to make a new tree and providing an `evaluate` method to recursively evaluate an expression tree once it has been constructed.

LISTING 10.2

```
import jsjf.*;

/**
 * ExpressionTree represents an expression tree of operators and operands.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class ExpressionTree extends LinkedBinaryTree<ExpressionTreeOp>
{
    /**
     * Creates an empty expression tree.
     */
    public ExpressionTree()
    {
        super();
    }

    /**
     * Constructs a expression tree from the two specified expression
     * trees.
     *
     * @param element    the expression tree for the center
     * @param leftSubtree the expression tree for the left subtree
     * @param rightSubtree the expression tree for the right subtree
     */
    public ExpressionTree(ExpressionTreeOp element,
        ExpressionTree leftSubtree, ExpressionTree rightSubtree)
    {
        root = new BinaryTreeNode<ExpressionTreeOp>(element, leftSubtree,
            rightSubtree);
    }

    /**
     * Evaluates the expression tree by calling the recursive
     * evaluateNode method.
     */
}
```

LISTING 10.2

continued

```
*
 * @return the integer evaluation of the tree
 */
public int evaluateTree()
{
    return evaluateNode(root);
}

/**
 * Recursively evaluates each node of the tree.
 *
 * @param root the root of the tree to be evaluated
 * @return the integer evaluation of the tree
 */
public int evaluateNode(BinaryTreeNode root)
{
    int result, operand1, operand2;
    ExpressionTreeOp temp;

    if (root==null)
        result = 0;
    else
    {
        temp = (ExpressionTreeOp)root.getElement();

        if (temp.isOperator())
        {
            operand1 = evaluateNode(root.getLeft());
            operand2 = evaluateNode(root.getRight());
            result = computeTerm(temp.getOperator(), operand1, operand2);
        }
        else
            result = temp.getValue();
    }

    return result;
}

/**
 * Evaluates a term consisting of an operator and two operands.
 *
 * @param operator the operator for the expression
 * @param operand1 the first operand for the expression
 * @param operand2 the second operand for the expression
 */
```

LISTING 10.2 *continued*

```

private int computeTerm(char operator, int operand1, int operand2)
{
    int result=0;

    if (operator == '+')
        result = operand1 + operand2;
    else if (operator == '-')
        result = operand1 - operand2;
    else if (operator == '*')
        result = operand1 * operand2;
    else
        result = operand1 / operand2;

    return result;
}

/**
 * Generates a structured string version of the tree by performing
 * a levelorder traversal.
 *
 * @return a string representation of this binary tree
 */
public String printTree()
{
    UnorderedListADT<BinaryTreeNode<ExpressionTreeOp>> nodes =
        new ArrayUnorderedList<BinaryTreeNode<ExpressionTreeOp>>();
    UnorderedListADT<Integer> levelList =
        new ArrayUnorderedList<Integer>();
    BinaryTreeNode<ExpressionTreeOp> current;
    String result = "";
    int printDepth = this.getHeight();
    int possibleNodes = (int)Math.pow(2, printDepth + 1);
    int countNodes = 0;

    nodes.addToRear(root);
    Integer currentLevel = 0;
    Integer previousLevel = -1;
    levelList.addToRear(currentLevel);

    while (countNodes < possibleNodes)
    {
        countNodes = countNodes + 1;
        current = nodes.removeFirst();
        currentLevel = levelList.removeFirst();
    }
}

```

LISTING 10.2 *continued*

```
    if (currentLevel > previousLevel)
    {
        result = result + "\n\n";
        previousLevel = currentLevel;
        for (int j = 0;
            j < ((Math.pow(2, (printDepth - currentLevel))) - 1);
            j++)
            result = result + " ";
    }
    else
    {
        for (int i = 0;
            i < ((Math.pow(2, (printDepth - currentLevel + 1)) - 1));
            i++)
        {
            result = result + " ";
        }
    }
    if (current != null)
    {
        result = result + (current.getElement()).toString();
        nodes.addToRear(current.getLeft());
        levelList.addToRear(currentLevel + 1);
        nodes.addToRear(current.getRight());
        levelList.addToRear(currentLevel + 1);
    }
    else
    {
        nodes.addToRear(null);
        levelList.addToRear(currentLevel + 1);
        nodes.addToRear(null);
        levelList.addToRear(currentLevel + 1);
        result = result + " ";
    }
}

return result;
}
```

The `evaluateTree` method calls the recursive `evaluateNode` method. The `evaluateNode` method returns the value if the node contains a number, or, if the node contains an operation, it returns the result of the operation using the value of the left and right subtrees. The `ExpressionTree` class uses the `ExpressionTreeOp` class as the element to store at each node of the tree. The `ExpressionTreeOp` class enables us to keep track of whether the element is a number or an operator and of which operator or what value is stored there. The `ExpressionTreeOp` class is illustrated in Listing 10.3.

LISTING 10.3

```
import jsjf.*;

/**
 * ExpressionTreeOp represents an element in an expression tree.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class ExpressionTreeOp
{
    private int termType;
    private char operator;
    private int value;

    /**
     * Creates a new expression tree object with the specified data.
     *
     * @param type the integer type of the expression
     * @param op the operand for the expression
     * @param val the value for the expression
     */
    public ExpressionTreeOp(int type, char op, int val)
    {
        termType = type;
        operator = op;
        value = val;
    }

    /**
     * Returns true if this object is an operator and false otherwise.
     *
     * @return true if this object is an operator, false otherwise
     */
}
```

LISTING 10.3 *continued*

```
public boolean isOperator()
{
    return (termType == 1);
}

/**
 * Returns the operator of this expression tree object.
 *
 * @return the character representation of the operator
 */
public char getOperator()
{
    return operator;
}

/**
 * Returns the value of this expression tree object.
 *
 * @return the value of this expression tree object
 */
public int getValue()
{
    return value;
}

public String toString()
{
    if (termType == 1)
        return operator + "";
    else
        return value + "";
}
}
```

The `PostfixTester` and `PostfixEvaluator` classes are a modification of our solution from Chapter 3. This solution employs the `ExpressionTree` class to build, print, and evaluate an expression tree. Figure 10.11 illustrates this process for the expression tree from Figure 10.10. Note that the top of the expression tree stack is on the right.

Input in Postfix: 5 3 - 4 * 9 +



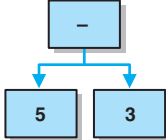
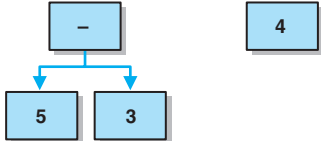
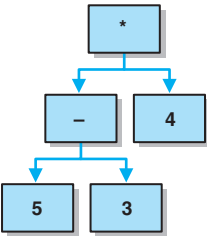
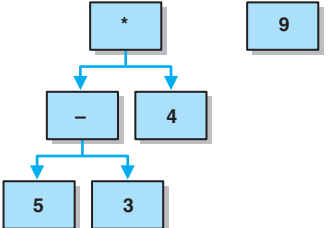
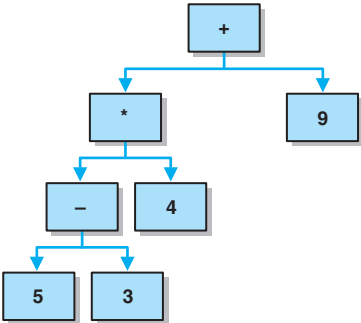
Token	Processing Steps	Expression Tree Stack (top at right)
5	push(new ExpressionTree(5, null, null))	
3	push(new ExpressionTree(3, null, null))	
-	op2 = pop op1 = pop push(new ExpressionTree(-, op1, op2))	
4	push(new ExpressionTree(4, null, null))	
*	op2 = pop op1 = pop push(new ExpressionTree(*, op1, op2))	
9	push(new ExpressionTree(9, null, null))	
+	op2 = pop op1 = pop push(new ExpressionTree(+, op1, op2))	

FIGURE 10.11 Building an expression tree from a postfix expression

LISTING 10.4

```
import java.util.Scanner;
/**
 * Demonstrates the use of a stack to evaluate postfix expressions.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class PostfixTester
{
    /**
     * Reads and evaluates multiple postfix expressions.
     */
    public static void main(String[] args)
    {
        String expression, again;
        int result;

        Scanner in = new Scanner(System.in);

        do
        {
            PostfixEvaluator evaluator = new PostfixEvaluator();
            System.out.println("Enter a valid post-fix expression one token " +
                "at a time with a space between each token (e.g. 5 4 + 3 2 1 - +
                *)");
            System.out.println("Each token must be an integer or an operator (+, -,
                *, /)");
            expression = in.nextLine();

            result = evaluator.evaluate(expression);
            System.out.println();
            System.out.println("That expression equals " + result);

            System.out.println("The Expression Tree for that expression is: ");
            System.out.println(evaluator.getTree());

            System.out.print("Evaluate another expression [Y/N]? ");
            again = in.nextLine();
            System.out.println();
        }
        while (again.equalsIgnoreCase("y"));
    }
}
```

The `PostfixTester` class is shown in Listing 10.4, and the `PostfixEvaluator` class is shown in Listing 10.5. The UML description of the `Postfix` class is shown in Figure 10.12.

LISTING 10.5

```
import jsjf.*;
import jsjf.exceptions.*;
import java.util.*;
import java.io.*;

/**
 * PostfixEvaluator this modification of our stack example uses a
 * stack to create an expression tree from a VALID integer postfix expression
 * and then uses a recursive method from the ExpressionTree class to
 * evaluate the tree.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class PostfixEvaluator
{
    private String expression;
    private Stack<ExpressionTree> treeStack;

    /**
     * Sets up this evaluator by creating a new stack.
     */
    public PostfixEvaluator()
    {
        treeStack = new Stack<ExpressionTree>();
    }

    /**
     * Retrieves and returns the next operand off of this tree stack.
     *
     * @param treeStack the tree stack from which the operand will be returned
     * @return the next operand off of this tree stack
     */
    private ExpressionTree getOperand(Stack<ExpressionTree> treeStack)
    {
        ExpressionTree temp;
        temp = treeStack.pop();

        return temp;
    }

    /**
     * Evaluates the specified postfix expression by building and evaluating

```

LISTING 10.5

continued

```

    * an expression tree.
    *
    * @param expression string representation of a postfix expression
    * @return value of the given expression
    */
public int evaluate(String expression)
{
    ExpressionTree operand1, operand2;
    char operator;
    String tempToken;

    Scanner parser = new Scanner(expression);

    while (parser.hasNext())
    {
        tempToken = parser.next();
        operator = tempToken.charAt(0);

        if ((operator == '+') || (operator == '-') || (operator == '*') ||
            (operator == '/'))
        {
            operand1 = getOperand(treeStack);
            operand2 = getOperand(treeStack);
            treeStack.push(new ExpressionTree
                (new ExpressionTreeOp(1,operator,0), operand2, operand1));
        }
        else
        {
            treeStack.push(new ExpressionTree(new ExpressionTreeOp
                (2,' ',Integer.parseInt(tempToken)), null, null));
        }
    }
    return (treeStack.peek()).evaluateTree();
}

/**
 * Returns the expression tree associated with this postfix evaluator.
 *
 * @return string representing the expression tree
 */
public String getTree()
{
    return (treeStack.peek()).printTree();
}
}

```

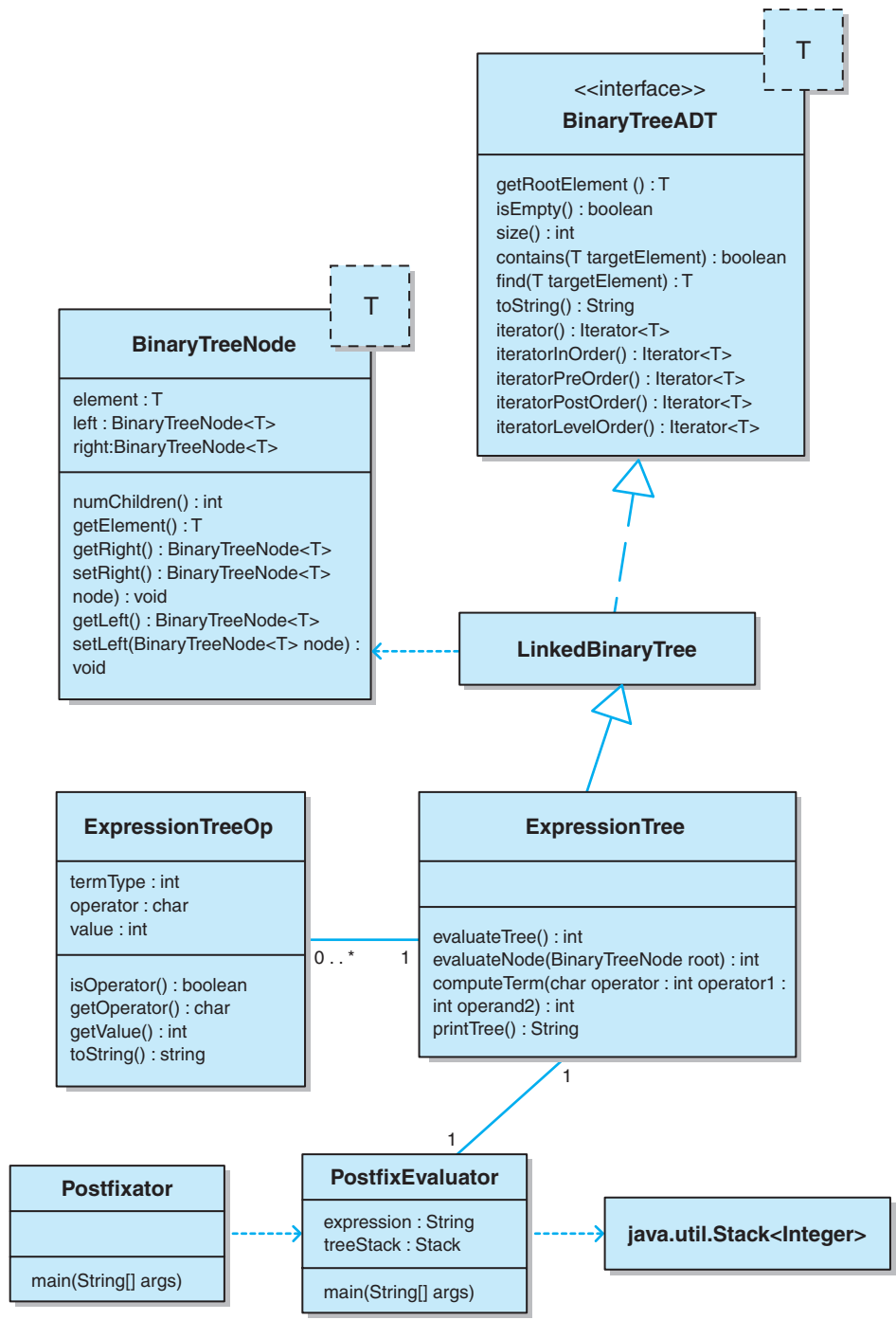


FIGURE 10.12 UML description of the Postfix example

10.6 A Back Pain Analyzer

Notice that the `ExpressionTree` class extended the `LinkedBinaryTree` class. Keep in mind that when one class is derived from another, it creates an is-a relationship. This extension to create the `ExpressionTree` class is natural, given that an expression tree is a binary tree.

Let's look at another example where our solution uses the `LinkedBinaryTree` class but does not extend it. A *decision tree* is a tree whose nodes represent decision points and whose children represent the options available at that point. The leaves of a decision tree represent the possible conclusions that might be drawn based on the answers.

A simple decision tree, with yes/no questions, can be modeled by a binary tree. Figure 10.13 shows a decision tree that helps to diagnose the cause of back pain. For each question, the left child represents the answer No, and the right child represents the answer Yes. To perform a diagnosis, begin with the question at the root and follow the appropriate path, based on the answers, until a leaf is reached.

Decision trees are sometimes used as a basis for an *expert system*, which is software that attempts to represent the knowledge of an expert in a particular field. For instance, a particular expert system might be used to model the expertise of a doctor, a car mechanic, or an accountant. Obviously, the greatly simplified decision tree in Figure 10.13 is not fleshed out enough to do a good job diagnosing the real cause of back pain, but it should give you a feel for how such systems might work.

KEY CONCEPT

A decision tree can be used as the basis for an expert system.

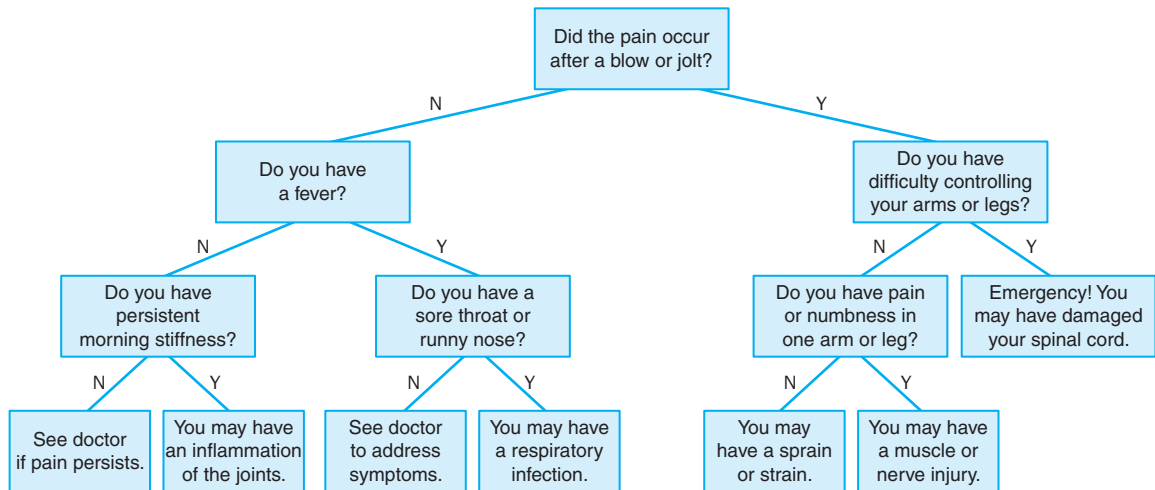


FIGURE 10.13 A decision tree for diagnosing back pain

Let's look at an example that uses the `LinkedBinaryTree` implementation discussed in the previous section to represent a decision tree. The program in Listing 10.6 uses the tree pictured in Figure 10.13 to hold a dialog with the user and draw a conclusion. The UML description of our solution to the back pain analyzer problem is presented in Figure 10.14.

LISTING 10.6

```
import java.io.*;

/**
 * BackPainAnalyzer demonstrates the use of a binary decision tree to
 * diagnose back pain.
 */
public class BackPainAnalyzer
{
    /**
     * Asks questions of the user to diagnose a medical problem.
     */
    public static void main (String[] args) throws FileNotFoundException
    {
        System.out.println ("So, you're having back pain.");

        DecisionTree expert = new DecisionTree("input.txt");
        expert.evaluate();
    }
}
```

OUTPUT

```
So, you're having back pain.
Did the pain occur after a blow or jolt?
Y
Do you have difficulty controlling your arms or legs?
N
Do you have pain or numbness in one arm or leg?
Y
You may have a muscle or nerve injury.
```

The tree is constructed and used in the `DecisionTree` class, shown in Listing 10.7. The only instance data is the variable `tree` that represents the entire decision tree, which is defined to store `String` objects as elements. Note that this

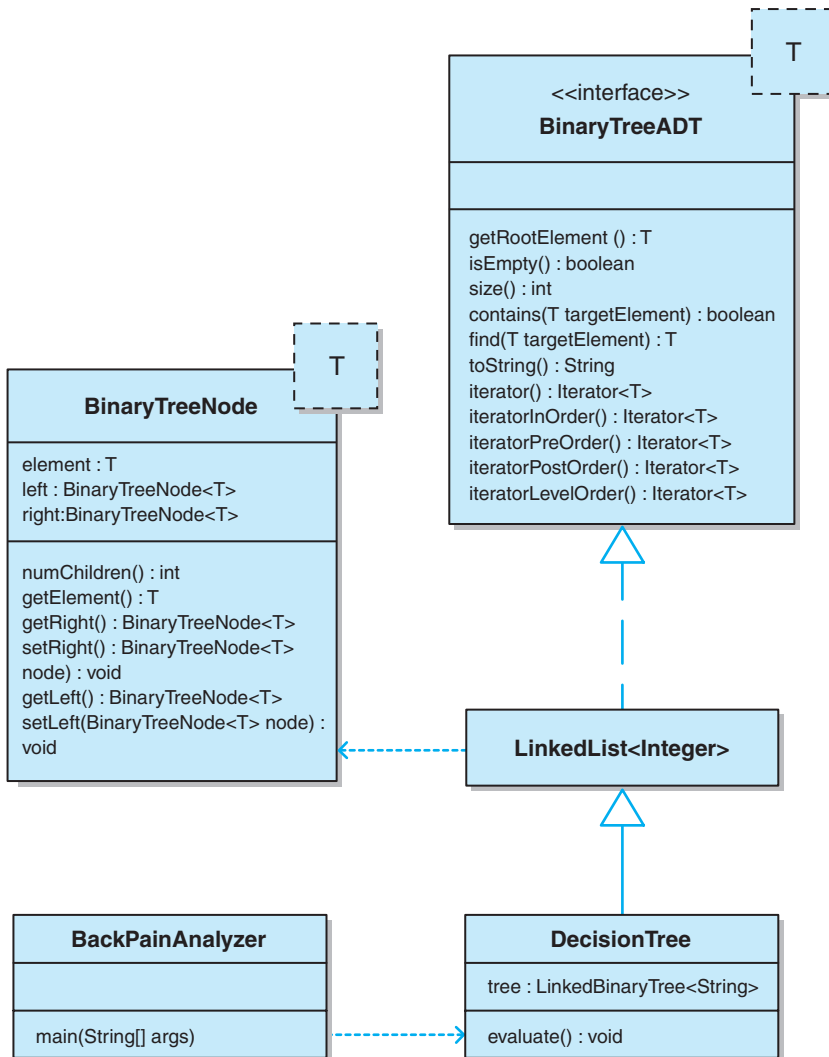


FIGURE 10.14 UML description of the back pain analyzer

version of the `DecisionTree` class is not specific to the back pain analyzer. It could be used for any binary decision tree.

The constructor of `DecisionTree` reads the various string elements to be stored in the tree nodes from the given file. Then the nodes themselves are created, with no children for the leaves and with previously defined nodes (or subtrees) as children for internal nodes. The tree is basically created from the bottom up.

The `evaluate` method uses the variable `current` to indicate the current node in the tree being processed, beginning at the root. The `while` loop continues until a leaf is found. The current question is printed, and the answer is read from the user. If the answer is `No`, `current` is updated to point to the left child. Otherwise, it is updated to point to the right child. After falling out of the loop, the element stored in the leaf (the conclusion) is printed.

LISTING 10.7

```
import jsjf.*;
import java.util.*;
import java.io.*;

/**
 * The DecisionTree class uses the LinkedBinaryTree class to implement
 * a binary decision tree. Tree elements are read from a given file and
 * then the decision tree can be evaluated based on user input using the
 * evaluate method.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class DecisionTree
{
    private LinkedBinaryTree<String> tree;

    /**
     * Builds the decision tree based on the contents of the given file
     *
     * @param filename the name of the input file
     * @throws FileNotFoundException if the input file is not found
     */
    public DecisionTree(String filename) throws FileNotFoundException
    {
        File inputFile = new File(filename);
        Scanner scan = new Scanner(inputFile);
        int numberNodes = scan.nextInt();
        scan.nextLine();
        int root = 0, left, right;

        List<LinkedBinaryTree<String>> nodes =
            new java.util.ArrayList<LinkedBinaryTree<String>>();
        for (int i = 0; i < numberNodes; i++)
            nodes.add(i, new LinkedBinaryTree<String>(scan.nextLine()));

        while (scan.hasNext())
        {
```

LISTING 10.7 *continued*

```
    root = scan.nextInt();
    left = scan.nextInt();
    right = scan.nextInt();
    scan.nextLine();

    nodes.set(root,
        new LinkedBinaryTree<String>((nodes.get(root)).getRootElement(),
            nodes.get(left), nodes.get(right)));
    }
    tree = nodes.get(root);
}

/**
 * Follows the decision tree based on user responses.
 */
public void evaluate()
{
    LinkedBinaryTree<String> current = tree;
    Scanner scan = new Scanner(System.in);

    while (current.size() > 1)
    {
        System.out.println (current.getRootElement());
        if (scan.nextLine().equalsIgnoreCase("N"))
            current = current.getLeft();
        else
            current = current.getRight();
    }

    System.out.println (current.getRootElement());
}
}
```

10.7 Implementing Binary Trees with Links

We will examine how some of these methods might be implemented using a linked implementation; others will be left as exercises. The `LinkedBinaryTree` class implementing the `BinaryTreeADT` interface will need to keep track of the node that is at the root of the tree and the number of elements on the tree. The `LinkedBinaryTree` header and instance data could be declared as

```

package jsjf;

import java.util.*;
import jsjf.exceptions.*;

/**
 * LinkedBinaryTree implements the BinaryTreeADT interface
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class LinkedBinaryTree<T> implements BinaryTreeADT<T>, Iterable<T>
{
    protected BinaryTreeNode<T> root;
    protected int modCount;

```

The constructors for the `LinkedBinaryTree` class should handle three cases: We want to create a binary tree with nothing in it, we want to create a binary tree with a single element but no children, and we want to create a binary tree with a particular element at the root and two given trees as children. With these goals in mind, the `LinkedBinaryTree` class might have the following constructors. Note that each of the constructors must account for both the root and count attributes.

```

/**
 * Creates an empty binary tree.
 */
public LinkedBinaryTree()
{
    root = null;
}

/**
 * Creates a binary tree with the specified element as its root.
 *
 * @param element the element that will become the root of the binary tree
 */
public LinkedBinaryTree(T element)
{
    root = new BinaryTreeNode<T>(element);
}

/**
 * Creates a binary tree with the specified element as its root and the

```

```

    * given trees as its left child and right child
    *
    * @param element the element that will become the root of the binary tree
    * @param left the left subtree of this tree
    * @param right the right subtree of this tree
    */
    public LinkedBinaryTree(T element, LinkedBinaryTree<T> left,
        LinkedBinaryTree<T> right)
    {
        root = new BinaryTreeNode<T>(element);
        root.setLeft(left.root);
        root.setRight(right.root);
    }

```

Note that both the instance data and the constructors use an additional class called `BinaryTreeNode`. As discussed earlier, this class keeps track of the element stored at each location, as well as pointers to the left and right subtree or children of each node. In this particular implementation, we chose not to include a pointer back to the parent of each node. Listing 10.8 shows the `BinaryTreeNode` class. The `BinaryTreeNode` class also includes a recursive method to return the number of children of the given node.

LISTING 10.8

```

package jsjf;

/**
 * BinaryTreeNode represents a node in a binary tree with a left and
 * right child.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class BinaryTreeNode<T>
{
    protected T element;
    protected BinaryTreeNode<T> left, right;

    /**
     * Creates a new tree node with the specified data.
     *
     * @param obj the element that will become a part of the new tree node
     */

```

LISTING 10.8 *continued*

```

public BinaryTreeNode(T obj)
{
    element = obj;
    left = null;
    right = null;
}

/**
 * Creates a new tree node with the specified data.
 *
 * @param obj the element that will become a part of the new tree node
 * @param left the tree that will be the left subtree of this node
 * @param right the tree that will be the right subtree of this node
 */
public BinaryTreeNode(T obj, LinkedBinaryTree<T> left,
    LinkedBinaryTree<T> right)
{
    element = obj;
    if (left == null)
        this.left = null;
    else
        this.left = left.getRootNode();

    if (right == null)
        this.right = null;
    else
        this.right = right.getRootNode();
}

/**
 * Returns the number of non-null children of this node.
 *
 * @return the integer number of non-null children of this node
 */
public int numChildren()
{
    int children = 0;

    if (left != null)
        children = 1 + left.numChildren();

    if (right != null)
        children = children + 1 + right.numChildren();
}

```

LISTING 10.8*continued*

```
        return children;
    }

    /**
     * Return the element at this node.
     *
     * @return the element stored at this node
     */
    public T getElement()
    {
        return element;
    }

    /**
     * Return the right child of this node.
     *
     * @return the right child of this node
     */
    public BinaryTreeNode<T> getRight()
    {
        return right;
    }

    /**
     * Sets the right child of this node.
     *
     * @param node the right child of this node
     */
    public void setRight(BinaryTreeNode<T> node)
    {
        right = node;
    }

    /**
     * Return the left child of this node.
     *
     * @return the left child of the node
     */
    public BinaryTreeNode<T> getLeft()
    {
        return left;
    }

    /**
     * Sets the left child of this node.
```

LISTING 10.8

continued

```

*
* @param node the left child of this node
*/
public void setLeft(BinaryTreeNode<T> node)
{
    left = node;
}
}

```

There are a variety of other possibilities for implementation of a tree node or binary tree node class. For example, methods could be included to test whether the node is a leaf (does not have any children), to test whether the node is an internal node (has at least one child), to test the depth of the node from the root, or to calculate the height of the left and right subtrees.

Another alternative would be to use polymorphism such that, rather than testing a node to see whether it has data or has children, we would create various implementations, such as an `emptyTreeNode`, an `innerTreeNode`, and a `leafTreeNode`, that would distinguish the various possibilities.

The `find` Method

As with our earlier collections, our `find` method traverses the tree using the `equals` method of the class stored in the tree to determine equality. This puts the definition of equality under the control of the class being stored in the tree. The `find` method throws an exception if the target element is not found.

Many methods associated with trees may be written either recursively or iteratively. Often, when written recursively, these methods require the use of a private support method, because the signature and/or the behavior of the first call and each successive call may not be the same. The `find` method in our simple implementation is an excellent example of this strategy.

We have chosen to use a recursive `findAgain` method. We know that the first call to `find` will start at the root of the tree, and if that instance of the `find` method completes without finding the target, we need to throw an exception. The private `findAgain` method enables us to distinguish between this first instance of the `find` method and each successive call.

```
/**
 * Returns a reference to the specified target element if it is
 * found in this binary tree. Throws a ElementNotFoundException if
 * the specified target element is not found in the binary tree.
 *
 * @param targetElement the element being sought in this tree
 * @return a reference to the specified target
 * @throws ElementNotFoundException if the element is not in the tree
 */
public T find(T targetElement) throws ElementNotFoundException
{
    BinaryTreeNode<T> current = findNode(targetElement, root);

    if (current == null)
        throw new ElementNotFoundException("LinkedBinaryTree");

    return (current.getElement());
}

/**
 * Returns a reference to the specified target element if it is
 * found in this binary tree.
 *
 * @param targetElement the element being sought in this tree
 * @param next the element to begin searching from
 */
private BinaryTreeNode<T> findNode(T targetElement,
                                   BinaryTreeNode<T> next)
{
    if (next == null)
        return null;

    if (next.getElement().equals(targetElement))
        return next;

    BinaryTreeNode<T> temp = findNode(targetElement, next.getLeft());

    if (temp == null)
        temp = findNode(targetElement, next.getRight());

    return temp;
}
```


As seen in earlier examples, the `contains` method can make use of the `find` method. Our implementation of this is left as a programming project.

The `iteratorInOrder` Method

Another interesting operation is the `iteratorInOrder` method. The task is to create an `Iterator` object that will allow a user class to step through the elements of the tree in an inorder traversal. The solution to this problem provides another example of using one collection to build another. We simply traverse the tree using a definition of “visit” from earlier pseudocode that adds the contents of the node onto an unordered list. We then use the list iterator to create a new `TreeIterator`. This approach is possible because of the linear nature of an unordered list and the way that we implemented the iterator method for a list. The iterator method for a list returns an `Iterator` that starts with the element at the front of the list and steps through the list in a linear fashion. It is important to understand that this behavior is not a requirement for an iterator associated with a list. It is simply an artifact of the way we chose to implement the `iterator` method for a list. What would happen if we simply returned the `Iterator` for our list without creating a `TreeIterator`? The problem with that solution would be that our `Iterator` would no longer be fail-fast (that is, it would no longer throw a concurrent modification exception if the underlying tree were modified while the iterator was in use).

As in the `find` operation, we use a private helper method in our recursion.

```
/**
 * Performs an inorder traversal on this binary tree by calling an
 * overloaded, recursive inorder method that starts with
 * the root.
 *
 * @return an in order iterator over this binary tree
 */
public Iterator<T> iteratorInOrder()
{
    ArrayUnorderedList<T> tempList = new ArrayUnorderedList<T>();
    inorder(root, tempList);

    return new TreeIterator(tempList.iterator());
}

/**
 * Performs a recursive inorder traversal.
 */
```

```
    * @param node the node to be used as the root for this traversal
    * @param tempList the temporary list for use in this traversal
    */
    protected void inOrder(BinaryTreeNode<T> node,
        ArrayUnorderedList<T> tempList)
    {
        if (node != null)
        {
            inOrder(node.getLeft(), tempList);
            tempList.addToRear(node.getElement());
            inOrder(node.getRight(), tempList);
        }
    }
}
```

The other iterator operations are similar and are left as exercises. Likewise, the array implementation of a binary tree is left as an exercise and will be revisited in Chapter 12.

Summary of Key Concepts

- A tree is a nonlinear structure whose elements are organized into a hierarchy.
- Trees are described by a large set of related terms.
- The simulated link strategy allows array positions to be allocated contiguously, regardless of the completeness of the tree.
- In general, a balanced n -ary tree with m elements will have height $\log_n m$.
- There are four basic methods for traversing a tree: preorder, inorder, postorder, and level-order traversals.
- *Preorder traversal* means visit the node, then the left child, then the right child.
- *Inorder traversal* means visit the left child, then the node, then the right child.
- *Postorder traversal* means visit the left child, then the right child, then the node.
- *Level-order traversal* means visit the nodes at each level, one level at a time, starting with the root.
- A decision tree can be used as the basis for an expert system.

Summary of Terms

tree A tree is a nonlinear structure whose elements are organized into a hierarchy.

node A location within a tree.

edge A connection between two nodes of a tree.

root The node at the top level of a tree and the one node in the tree that does not have a parent.

level The position of a node relative to the root of the tree.

child A node that is below the current node in tree and directly connected to it by an edge.

siblings Nodes that are children of the same node.

leaf A node in a tree that does not have any children.

internal node A node in a tree that is not the root and has at least one child.

path The collection of edges that directly connects a node to another node of the tree.

ancestor A node that is above the current node on the path from the root.

descendant A node that is below the current node in the tree and is on a path from the current node to a leaf (including the leaf).

path length The number of edges that must be followed to connect one node to another.

tree height The length of the longest path from the root to a leaf.

tree order The maximum number of children that any node in the tree may have.

general tree A tree that has no limit on the number of children a node may have.

n-ary tree A tree that limits each node to no more than n children.

binary tree A tree in which nodes may have at most two children.

balanced Roughly speaking, a tree is considered balanced if all the leaves of the tree are on the same level or at least within one level of each other.

complete A tree is considered complete if it is balanced and all of the leaves at the bottom level are on the left side of the tree.

full An n -ary tree is considered full if all the leaves of the tree are at the same level and every node either is a leaf or has exactly n children.

freelist A list of available positions in an array implementation of a tree.

binary search tree A binary tree with the added property that the left child is always less than the parent, which is always less than or equal to the right child.

preorder traversal A tree traversal accomplished by visiting each node, followed by its children, starting with the root.

inorder traversal A tree traversal accomplished by visiting the left child of the node, then the node, and then any remaining nodes, starting with the root.

postorder traversal A tree traversal accomplished by visiting the children and then the node, starting with the root.

level-order traversal A tree traversal accomplished by visiting all of the nodes at each level, one level at a time, starting with the root.

Self-Review Questions

SR 10.1 What is a tree?

SR 10.2 What is a node?

SR 10.3 What is the root of a tree?

SR 10.4 What is a leaf?

SR 10.5 What is an internal node?

- SR 10.6 Define the height of a tree.
- SR 10.7 Define the level of a node.
- SR 10.8 What are the advantages and disadvantages of the computational strategy?
- SR 10.9 What are the advantages and disadvantages of the simulated link strategy?
- SR 10.10 What attributes should be stored in the `TreeNode` class?
- SR 10.11 Which method of traversing a tree would result in a sorted list for a binary search tree?
- SR 10.12 We used a list to implement the iterator methods for a binary tree. What must be true for this strategy to be successful?

Exercises

- EX 10.1 Develop a pseudocode algorithm for a level-order traversal of a binary tree.
- EX 10.2 Draw either a matrilineage (following your mother's lineage) or a patrilineage (following your father's lineage) diagram for a couple of generations. Develop a pseudocode algorithm for inserting a person into the proper place in the tree.
- EX 10.3 Develop a pseudocode algorithm to build an expression tree from a prefix expression.
- EX 10.4 Develop a pseudocode algorithm to build an expression tree from an infix expression.
- EX 10.5 Calculate the time complexity of the `find` method.
- EX 10.6 Calculate the time complexity of the `iteratorInOrder` method.
- EX 10.7 Develop a pseudocode algorithm for the `size` method, assuming that there is not a `count` variable.
- EX 10.8 Develop a pseudocode algorithm for the `isEmpty` operation, assuming that there is not a `count` variable.
- EX 10.9 Draw an expression tree for the expression $(9 + 4) * 5 + (4 - (6 - 3))$.

Programming Projects

- PP 10.1 Complete the implementation of the `getRoot` and `toString` operations of a binary tree.

- PP 10.2 Complete the implementation of the `size` and `isEmpty` operations of a binary tree, assuming that there is not a `count` variable.
- PP 10.3 Create boolean methods for our `BinaryTreeNode` class to determine whether the node is a leaf or an internal node.
- PP 10.4 Create a method called `depth` that will return an `int` representing the level or depth of the given node from the root.
- PP 10.5 Complete the implementation of the `contains` method for a binary tree.
- PP 10.6 Implement the `contains` method for a binary tree without using the `find` operation.
- PP 10.7 Complete the implementation of the iterator methods for a binary tree.
- PP 10.8 Implement the iterator methods for a binary tree without using a list.
- PP 10.9 Modify the `ExpressionTree` class to create a method called `draw` that will graphically depict the expression tree.
- PP 10.10 We use postfix notation in the example in this chapter because it eliminates the need to parse an infix expression by precedence rules and parentheses. Some infix expressions do not need parentheses to modify precedence. Implement a method for the `ExpressionTree` class that will determine whether an integer expression would require parentheses if it were written in infix notation.
- PP 10.11 Create an array-based implementation of a binary tree using the computational strategy.
- PP 10.12 Create an array-based implementation of a binary tree using the simulated link strategy.
- PP 10.13 Create an implementation of a binary tree using the recursive approach introduced in the chapter. In this approach, each node is a binary tree. Thus a binary tree contains a reference to the element stored at its root, as well as references to its left and right subtrees. You may also want to include a reference to its parent.

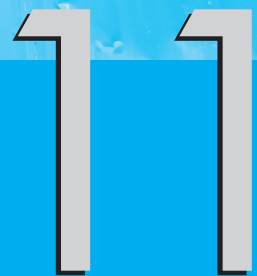
Answers to Self-Review Questions

- SRA 10.1 A tree is a nonlinear structure defined by the concept that each node in the tree, other than the first node or root node, has exactly one parent.

- SRA 10.2 A node is a location in a tree where an element is stored.
- SRA 10.3 The root of a tree is the node at the base of the tree, or the one node in the tree that does not have a parent.
- SRA 10.4 A leaf is a node that does not have any children.
- SRA 10.5 An internal node is any non-root node that has at least one child.
- SRA 10.6 The height of the tree is the length of the longest path from the root to a leaf.
- SRA 10.7 The level of a node is measured by the number of links that must be followed to reach that node from the root.
- SRA 10.8 The computational strategy does not have to store links from parent to child because that relationship is fixed by position. However, this strategy may lead to substantial wasted space for trees that are not balanced and/or not complete.
- SRA 10.9 The simulated link strategy stores array index values as pointers between parent and child and allows the data to be stored contiguously no matter how balanced and/or complete the tree. However, this strategy increases the overhead in terms of maintaining a freelist or shifting elements in the array.
- SRA 10.10 The `TreeNode` class must store a pointer to the element stored in that position, as well as pointers to each of the children of that node. The class may also contain a pointer to the parent of the node.
- SRA 10.11 Inorder traversal of a binary search tree would result in a sorted list in ascending order.
- SRA 10.12 For this strategy to be successful, the iterator for a list must return the elements in the order in which they were added. For this particular implementation of a list, we know this is indeed the case.



Binary Search Trees



In this chapter, we will explore the concept of binary search trees and options for their implementation. We will examine algorithms for adding and removing elements from binary search trees and for maintaining balanced binary search trees. We will discuss the analysis of these implementations and also explore various uses of binary search trees.

CHAPTER OBJECTIVES

- Define a binary search tree abstract data structure.
- Demonstrate how a binary search tree can be used to solve problems.
- Examine a binary search tree implementation.
- Discuss strategies for balancing a binary search tree.

11.1 A Binary Search Tree

A *binary search tree* is a binary tree with the added property that, for each node, the left child is less than the parent, which is less than or equal to the right child. As we discussed in Chapter 10, it is very difficult to abstract a set of operations for a tree without knowing what type of tree it is and its intended purpose. With the added ordering property that must be maintained, we can now extend our definition to include the operations on a binary search tree that are listed in Figure 11.1.

Operation	Description
<code>addElement</code>	Adds an element to the tree.
<code>removeElement</code>	Removes an element from the tree.
<code>removeAllOccurrences</code>	Removes all occurrences of element from the tree.
<code>removeMin</code>	Removes the minimum element in the tree.
<code>removeMax</code>	Removes the maximum element in the tree.
<code>findMin</code>	Returns a reference to the minimum element in the tree.
<code>findMax</code>	Returns a reference to the maximum element in the tree.

FIGURE 11.1 The operations on a binary search tree

KEY CONCEPT

A binary search tree is a binary tree with the added property that the left child is less than the parent, which is less than or equal to the right child.

KEY CONCEPT

The definition of a binary search tree is an extension of the definition of a binary tree.

As we discussed in Chapter 10, the Java Collections API does not provide an implementation of a general tree. Instead, trees are used as an implementation strategy for Sets and Maps. We will discuss the API treatment of trees in Chapter 13. In the meantime, we will build upon our own linked implementation of trees from Chapter 10.

We must keep in mind that the definition of a binary search tree is an extension of the definition of a binary tree discussed in the last chapter. Thus, these operations are in addition to the ones defined for a binary tree. At this point we are simply discussing binary search trees, but as we will see shortly, the interface for a balanced binary search tree will be the same. Listing 11.1 and Figure 11.2 describe a `BinarySearchTreeADT`.

LISTING 11.1

```
package jsjf;

/**
 * BinarySearchTreeADT defines the interface to a binary search tree.
 *
 * @author Lewis and Chase
```

LISTING 11.1*continued*

```
* @version 4.0
*/
public interface BinarySearchTreeADT<T> extends BinaryTreeADT<T>
{
    /**
     * Adds the specified element to the proper location in this tree.
     *
     * @param element the element to be added to this tree
     */
    public void addElement(T element);

    /**
     * Removes and returns the specified element from this tree.
     *
     * @param targetElement the element to be removed from the tree
     * @return the element to be removed from the tree
     */
    public T removeElement(T targetElement);

    /**
     * Removes all occurrences of the specified element from this tree.
     *
     * @param targetElement the element to be removed from the tree
     */
    public void removeAllOccurrences(T targetElement);

    /**
     * Removes and returns the smallest element from this tree.
     *
     * @return the smallest element from the tree.
     */
    public T removeMin();

    /**
     * Removes and returns the largest element from this tree.
     *
     * @return the largest element from the tree
     */
    public T removeMax();

    /**
     * Returns the smallest element in this tree without removing it.
     *
     * @return the smallest element in the tree
     */
}
```

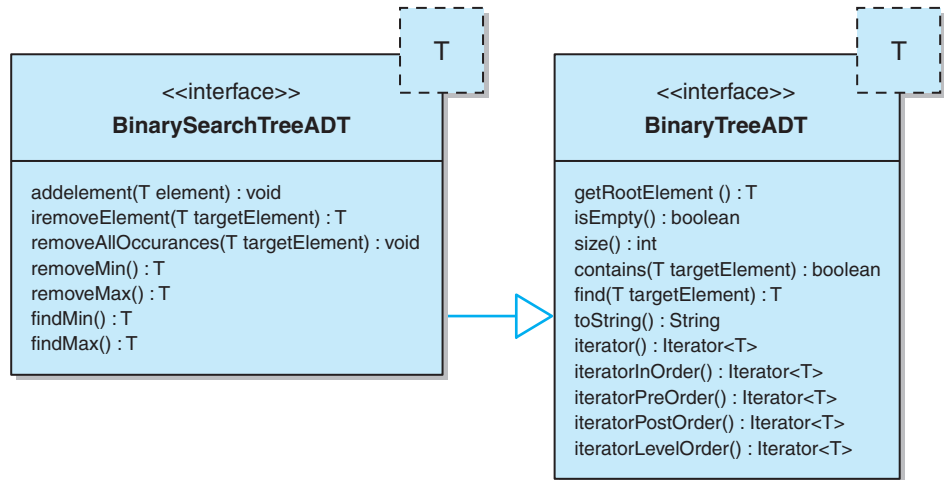
LISTING 11.1 *continued*

```

public T findMin();

/**
 * Returns the largest element in this tree without removing it.
 *
 * @return the largest element in the tree
 */
public T findMax();
}

```

**FIGURE 11.2** UML description of the `BinarySearchTreeADT`

11.2 Implementing Binary Search Trees: With Links

In Chapter 10, we introduced a simple implementation of a `LinkedBinaryTree` class using a `BinaryTreeNode` class to represent each node of the tree. Each `BinaryTreeNode` object maintains a reference to the element stored at that node, as well as references to each of the node's children. We can simply extend that definition with a `LinkedBinarySearchTree` class that implements the `BinarySearchTreeADT` interface. Because we are extending the `LinkedBinaryTree` class from Chapter 10, all of the methods we discussed are still supported, including the various traversals.

KEY CONCEPT

Each `BinaryTreeNode` object maintains a reference to the element stored at that node, as well as references to each of the node's children.

Our `LinkedBinarySearchTree` class offers two constructors: one to create an empty `LinkedBinarySearchTree` and the other to create a `LinkedBinarySearchTree` with a particular element at the root. Both of these constructors simply refer to the equivalent constructors of the super class (that is, the `LinkedBinaryTree` class).

```
/**
 * Creates an empty binary search tree.
 */
public LinkedBinarySearchTree()
{
    super();
}

/**
 * Creates a binary search with the specified element as its root.
 *
 * @param element the element that will be the root of the new binary
 *     search tree
 */
public LinkedBinarySearchTree(T element)
{
    super(element);

    if (!(element instanceof Comparable))
        throw new NonComparableElementException("LinkedBinarySearchTree");
}
```

The addElement Operation

The `addElement(element)` method adds a given element to the appropriate location in the tree using the private, recursive `addElement(element, tree)` method. If the element is not `Comparable`, the method throws a `NonComparableElementException`. If the tree is empty, the new element becomes the root. If the tree is not empty, the new element is compared to the element at the root. If it is less than the element stored at the root and the left child of the root is null, then the new element becomes the left child of the root. If the new element is less than the element stored at the root and the left child of the root is not null, then we recursively add the element to the left subtree of the root. If the new element is greater than or equal to the element stored at the root and the right child of the root is null, then the new element becomes the right child of the root. If the new element is greater than or equal to the element stored at the root and the right child of the root is not null, then we recursively add the element

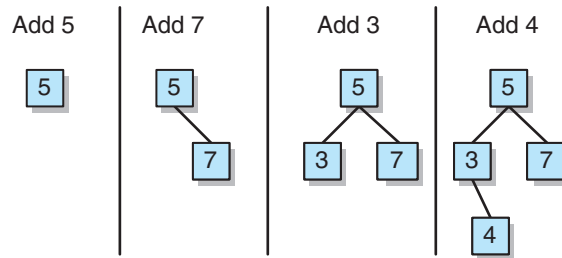


FIGURE 11.3 Adding elements to a binary search tree

to the right subtree of the root. Figure 11.3 illustrates this process of adding elements to a binary search tree. As in any recursive algorithm, we could have chosen to implement the add operation iteratively. The iterative version of the add operation is left as a programming project.

DESIGN FOCUS

Once we have a definition of the type of tree that we wish to construct and how it is to be used, we have the ability to define an interface and implementations. In Chapter 10, we defined a binary tree that enabled us to define a very basic set of operations. Now that we have limited our scope to a binary search tree, we can fill in more details of the interface and the implementation. Determining the level at which to build interface descriptions and determining the boundaries between parent and child classes are design choices . . . and they are not always easy design choices.

```
/**
 * Adds the specified object to the binary search tree in the
 * appropriate position according to its natural order. Note that
 * equal elements are added to the right.
 *
 * @param element the element to be added to the binary search tree
 */
public void addElement(T element)
{
    if (!(element instanceof Comparable))
        throw new NonComparableElementException("LinkedBinarySearchTree");

    Comparable<T> comparableElement = (Comparable<T>)element;
```

```
    if (isEmpty())
        root = new BinaryTreeNode<T>(element);
    else
    {
        if (comparableElement.compareTo(root.getElement()) < 0)
        {
            if (root.getLeft() == null)
                this.getRootNode().setLeft(new BinaryTreeNode<T>(element));
            else
                addElement(element, root.getLeft());
        }
        else
        {
            if (root.getRight() == null)
                this.getRootNode().setRight(new BinaryTreeNode<T>(element));
            else
                addElement(element, root.getRight());
        }
    }
    modCount++;
}

/**
 * Adds the specified object to the binary search tree in the
 * appropriate position according to its natural order. Note that
 * equal elements are added to the right.
 *
 * @param element the element to be added to the binary search tree
 */
private void addElement(T element, BinaryTreeNode<T> node)
{
    Comparable<T> comparableElement = (Comparable<T>)element;

    if (comparableElement.compareTo(node.getElement()) < 0)
    {
        if (node.getLeft() == null)
            node.setLeft(new BinaryTreeNode<T>(element));
        else
            addElement(element, node.getLeft());
    }
    else
    {
        if (node.getRight() == null)
            node.setRight(new BinaryTreeNode<T>(element));
        else
            addElement(element, node.getRight());
    }
}
```

The removeElement Operation

The `removeElement` method removes a given `Comparable` element from a binary search tree or throws an `ElementNotFoundException` if the given target is not found in the tree. Unlike our earlier study of linear structures, we cannot simply remove the node by making the reference point around the node to be removed. Instead, another node will have to be *promoted* to replace the one being removed. The protected method `replacement` returns a reference to a node that will replace the one specified for removal. There are three cases for selecting the replacement node:

- If the node has no children, `replacement` returns `null`.
- If the node has only one child, `replacement` returns that child.
- If the node to be removed has two children, `replacement` returns the inorder successor of the node to be removed (because equal elements are placed to the right).

KEY CONCEPT

In removing an element from a binary search tree, another node must be promoted to replace the node being removed.

Like our recursive `addElement` method, the `removeElement` (`targetElement`) method is recursive and makes use of the private `removeElement(targetElement, node, parent)` method. In this way, the special case of removing the root element can be handled separately.

```
/**
 * Removes the first element that matches the specified target
 * element from the binary search tree and returns a reference to
 * it. Throws a ElementNotFoundException if the specified target
 * element is not found in the binary search tree.
 *
 * @param targetElement the element being sought in the binary search tree
 * @throws ElementNotFoundException if the target element is not found
 */
public T removeElement(T targetElement)
    throws ElementNotFoundException
{
    T result = null;

    if (isEmpty())
        throw new ElementNotFoundException("LinkedBinarySearchTree");
    else
    {
        BinaryTreeNode<T> parent = null;
        if (((Comparable<T>)targetElement).equals(root.element))
        {
```

```

        result = root.element;
        BinaryTreeNode<T> temp = replacement(root);
        if (temp == null)
            root = null;
        else
        {
            root.element = temp.element;
            root.setRight(temp.right);
            root.setLeft(temp.left);
        }

        modCount--;
    }
    else
    {
        parent = root;
        if (((Comparable)targetElement).compareTo(root.element) < 0)
            result = removeElement(targetElement, root.getLeft(), parent);
        else
            result = removeElement(targetElement, root.getRight(), parent);
    }
}

return result;
}

/**
 * Removes the first element that matches the specified target
 * element from the binary search tree and returns a reference to
 * it. Throws a ElementNotFoundException if the specified target
 * element is not found in the binary search tree.
 *
 * @param targetElement the element being sought in the binary search tree
 * @param node the node from which to search
 * @param parent the parent of the node from which to search
 * @throws ElementNotFoundException if the target element is not found
 */
private T removeElement(T targetElement, BinaryTreeNode<T> node,
    BinaryTreeNode<T> parent) throws ElementNotFoundException
{
    T result = null;

    if (node == null)
        throw new ElementNotFoundException("LinkedBinarySearchTree");
    else
    {
        if (((Comparable<T>)targetElement).equals(node.element))

```



```

    {
        result = node.element;
        BinaryTreeNode<T> temp = replacement(node);
        if (parent.right == node)
            parent.right = temp;
        else
            parent.left = temp;

        modCount--;
    }
    else
    {
        parent = node;
        if (((Comparable)targetElement).compareTo(node.element) < 0)
            result = removeElement(targetElement, node.getLeft(), parent);
        else
            result = removeElement(targetElement, node.getRight(), parent);
    }
}

return result;
}

```

The following code illustrates the replacement method. Figure 11.4 further illustrates the process of removing elements from a binary search tree.

```

/**
 * Returns a reference to a node that will replace the one
 * specified for removal. In the case where the removed node has
 * two children, the inorder successor is used as its replacement.
 *
 * @param node the node to be removed
 * @return a reference to the replacing node
 */
private BinaryTreeNode<T> replacement(BinaryTreeNode<T> node)
{
    BinaryTreeNode<T> result = null;

    if ((node.left == null) && (node.right == null))
        result = null;

    else if ((node.left != null) && (node.right == null))
        result = node.left;

    else if ((node.left == null) && (node.right != null))
        result = node.right;
}

```

```

else
{
    BinaryTreeNode<T> current = node.right;
    BinaryTreeNode<T> parent = node;

    while (current.left != null)
    {
        parent = current;
        current = current.left;
    }

    current.left = node.left;
    if (node.right != current)
    {
        parent.left = current.right;
        current.right = node.right;
    }

    result = current;
}
return result;
}

```

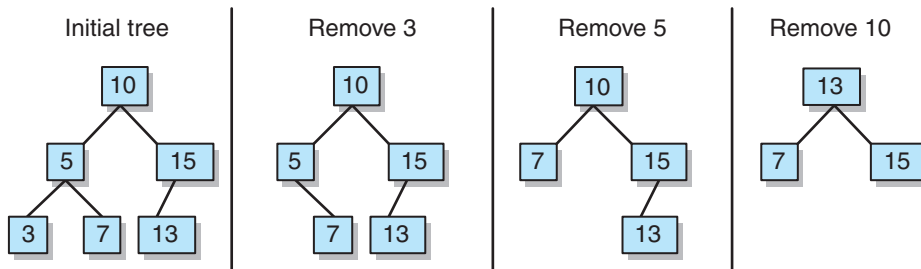


FIGURE 11.4 Removing elements from a binary tree

The removeAllOccurrences Operation

The `removeAllOccurrences` method removes all occurrences of a given element from a binary search tree and throws an `ElementNotFoundException` if the given element is not found in the tree. This method also throws a `ClassCastException` if the element given is not `Comparable`. This method makes use of the `removeElement` method by calling it once, which guarantees that the exception will be thrown if there is not at least one occurrence of the

element in the tree. The `removeElement` method is then called again as long as the tree contains the target element. Note that the `removeAllOccurrences` method makes use of the `contains` method of the `LinkedBinaryTree` class. Note that the `find` method has been overridden in the `LinkedBinarySearchTree` class to take advantage of the ordering property of a binary search tree.

```
/**
 * Removes elements that match the specified target element from
 * the binary search tree. Throws a ElementNotFoundException if
 * the specified target element is not found in this tree.
 *
 * @param targetElement the element being sought in the binary search tree
 * @throws ElementNotFoundException if the target element is not found
 */
public void removeAllOccurrences(T targetElement)
    throws ElementNotFoundException
{
    removeElement(targetElement);

    try
    {
        while (contains((T)targetElement))
            removeElement(targetElement);
    }
    catch (Exception ElementNotFoundException)
    {
    }
}
```

The `removeMin` Operation

There are three possible cases for the location of the minimum element in a binary search tree:

- If the root has no left child, then the root is the minimum element, and the right child of the root becomes the new root.
- If the leftmost node of the tree is a leaf, then it is the minimum element, and we simply set its parent's left child reference to null.
- If the leftmost node of the tree is an internal node, then we set its parent's left child reference to point to the right child of the node to be removed.

KEY CONCEPT

The leftmost node in a binary search tree will contain the minimum element, whereas the rightmost node will contain the maximum element.

Figure 11.5 illustrates these possibilities. Given these possibilities, the code for the `removeMin` operation is relatively straightforward.

```
/**
 * Removes the node with the least value from the binary search
 * tree and returns a reference to its element. Throws an
 * EmptyCollectionException if this tree is empty.
 *
 * @return a reference to the node with the least value
 * @throws EmptyCollectionException if the tree is empty
 */
public T removeMin() throws EmptyCollectionException
{
    T result = null;

    if (isEmpty())
        throw new EmptyCollectionException("LinkedBinarySearchTree");
    else
    {
        if (root.left == null)
        {
            result = root.element;
            root = root.right;
        }
        else
        {
            BinaryTreeNode<T> parent = root;
            BinaryTreeNode<T> current = root.left;
            while (current.left != null)
            {
                parent = current;
                current = current.left;
            }
            result = current.element;
            parent.left = current.right;
        }

        modCount--;
    }

    return result;
}
```

The `removeMax`, `findMin`, and `findMax` operations are left as exercises.

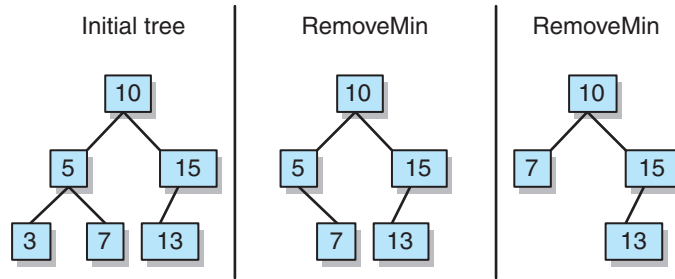


FIGURE 11.5 Removing the minimum element from a binary search tree

Implementing Binary Search Trees: With Arrays

In Chapter 10, we discussed two array implementation strategies for trees: the computational strategy and the simulated link strategy. For now, both implementations are left as programming projects. We will revisit these array-based tree implementations in Chapter 12.

11.3 Using Binary Search Trees: Implementing Ordered Lists

As we discussed in Chapter 10, one of the uses of trees is to provide efficient implementations of other collections. The `OrderedList` collection from Chapter 6 provides an excellent example. Figure 11.6 reminds us of the common operations for lists, and Figure 11.7 reminds us of the operation particular to an ordered list.

Operation	Description
<code>removeFirst</code>	Removes the first element from the list.
<code>removeLast</code>	Removes the last element from the list.
<code>remove</code>	Removes a particular element from the list.
<code>first</code>	Examines the element at the front of the list.
<code>last</code>	Examines the element at the rear of the list.
<code>contains</code>	Determines if the list contains a particular element.
<code>isEmpty</code>	Determines if the list is empty.
<code>size</code>	Determines the number of elements on the list.

FIGURE 11.6 The common operations on a list

Operation	Description
add	Adds an element to the list.

FIGURE 11.7 The operation particular to an ordered list

Using a binary search tree, we can create an implementation called `BinarySearchTreeList` that is a more efficient implementation than those we discussed in Chapter 6.

For simplicity, we have implemented both the `ListADT` and the `OrderedListADT` interfaces with the `BinarySearchTreeList` class, as shown in Listing 11.2. For some of the methods, the same method from either the `LinkedBinaryTree` or the `LinkedBinarySearchTree` class will suffice. This is the case for the `contains`, `isEmpty`, and `size` operations. For the rest of the operations, there is a one-to-one correspondence between methods of the `LinkedBinaryTree` or `LinkedBinarySearchTree` classes and the required methods for an ordered list. Thus, each of these methods is implemented by simply calling the associated method for a `LinkedBinarySearchTree`. This is the case for the `add`, `removeFirst`, `removeLast`, `remove`, `first`, `last`, and iterator methods.

KEY CONCEPT

One of the uses of trees is to provide efficient implementations of other collections.

LISTING 11.2

```
package jsjf;

import jsjf.exceptions.*;
import java.util.Iterator;

/**
 * BinarySearchTreeList represents an ordered list implemented using a binary
 * search tree.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class BinarySearchTreeList<T> extends LinkedBinarySearchTree<T>
    implements ListADT<T>, OrderedListADT<T>, Iterable<T>
{
    /**
     * Creates an empty BinarySearchTreeList.
     */
    public BinarySearchTreeList()
```

LISTING 11.2

continued

```
{
    super();
}

/**
 * Adds the given element to this list.
 *
 * @param element the element to be added to the list
 */
public void add(T element)
{
    addElement(element);
}

/**
 * Removes and returns the first element from this list.
 *
 * @return the first element in the list
 */
public T removeFirst()
{
    return removeMin();
}

/**
 * Removes and returns the last element from this list.
 *
 * @return the last element from the list
 */
public T removeLast()
{
    return removeMax();
}

/**
 * Removes and returns the specified element from this list.
 *
 * @param element the element being sought in the list
 * @return the element from the list that matches the target
 */
public T remove(T element)
{
    return removeElement(element);
}
```

LISTING 11.2*continued*

```
/**
 * Returns a reference to the first element on this list.
 *
 * @return a reference to the first element in the list
 */
public T first()
{
    return findMin();
}

/**
 * Returns a reference to the last element on this list.
 *
 * @return a reference to the last element in the list
 */
public T last()
{
    return findMax();
}

/**
 * Returns an iterator for the list.
 *
 * @return an iterator over the elements in the list
 */
public Iterator<T> iterator()
{
    return iteratorInOrder();
}
}
```

Analysis of the BinarySearchTreeList Implementation

For the sake of our analysis, we will assume that the `LinkedBinarySearchTree` implementation used in the `BinarySearchTreeList` implementation is a balanced binary search tree with the added property that the maximum depth of any node is $\log_2(n)$, where n is the number of elements stored in the tree. This is a tremendously important assumption, as we will see over the next several sections. With that assumption, Figure 11.8 shows a comparison of the order of each operation for a singly linked implementation of an ordered list and our `BinarySearchTreeList` implementation.

Operation	LinkedList	BinarySearchTreeList
removeFirst	O(1)	O(log n)
removeLast	O(n)	O(log n)
remove	O(n)	O(log n)*
first	O(1)	O(log n)
last	O(n)	O(log n)
contains	O(n)	O(log n)
isEmpty	O(1)	O(1)
size	O(1)	O(1)
add	O(n)	O(log n)*

*Both the `add` and `remove` operations may cause the tree to become unbalanced.

FIGURE 11.8 Analysis of linked list and binary search tree implementations of an ordered list

Note that given our assumption of a balanced binary search tree, both the `add` operation and the `remove` operation could cause the tree to need to be rebalanced, which, depending on the algorithm used, could affect the analysis. It is also important to note that although some operations, such as `removeLast`, `last`, and `contains`, are more efficient in the tree implementation, others, such as `removeFirst` and `first`, are less efficient when implemented using a tree.

11.4 Balanced Binary Search Trees

Why is our balance assumption important? What would happen to our analysis if the tree were not balanced? As an example, let's assume that we have read the following list of integers from a file and added them to a binary search tree:

3 5 9 12 18 20

Figure 11.9 shows the resulting binary search tree. This resulting binary tree, which is referred to as a *degenerate tree*, looks more like a linked list, and in fact it is less efficient than a linked list because of the additional overhead associated with each node.

KEY CONCEPT

If a binary search tree is not balanced, it may be less efficient than a linear structure.

If this is the tree we are manipulating, then our analysis from the previous section will look far worse. For example, without our balance assumption, the `addElement` operation would have worst-case time complexity of $O(n)$ instead of $O(\log n)$ because of the possibility that the root is the smallest element in the tree and the element we are inserting might be the largest element.

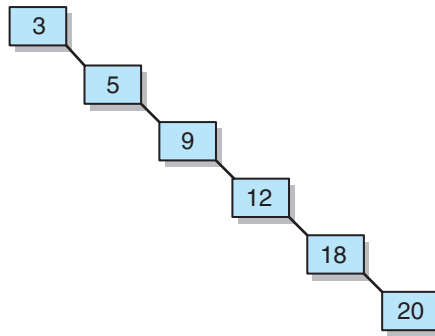


FIGURE 11.9 A degenerate binary tree

Our goal instead is to keep the maximum path length in the tree at or near $\log_2 n$. There are a variety of algorithms available for balancing or maintaining balance in a tree. There are brute force methods, which are not elegant or efficient, but get the job done. For example, we could write an inorder traversal of the tree to an array and then use a recursive method (much like binary search) to insert the middle element of the array as the root, and then build balanced left and right subtrees. Although such an approach would work, there are more elegant solutions, such as AVL trees and red/black trees, which we examine later in this chapter.

However, before we move on to these techniques, we need to understand some additional terminology that is common to many balancing techniques. The methods described here will work for any subtree of a binary search tree as well. For those subtrees, we simply replace the reference to root with the reference to the root of the subtree.

Right Rotation

Figure 11.10 shows a binary search tree that is not balanced and the processing steps necessary to rebalance it. The maximum path length in this tree is 3, and the minimum path length is 1. With only 6 elements in the tree, the maximum path length should be $\log_2 6$, or 2. To get this tree into balance, we need to

- Make the left child element of the root the new root element.
- Make the former root element the right child element of the new root.
- Make the right child of what was the left child of the former root the new left child of the former root.

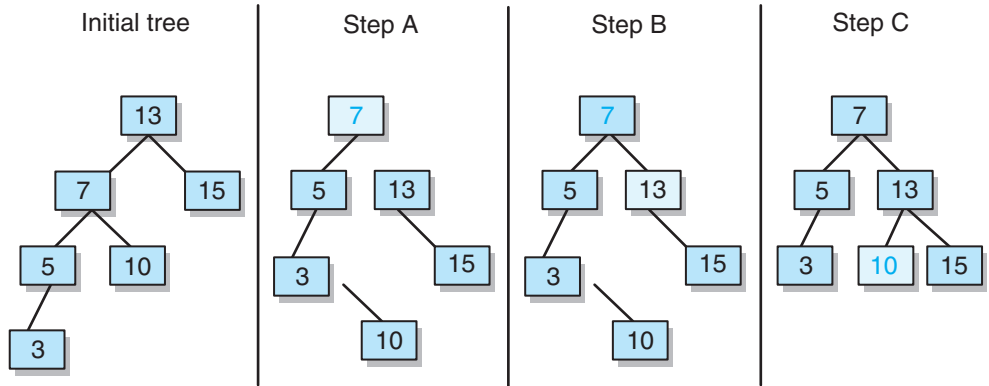


FIGURE 11.10 An unbalanced tree and the balanced tree that results from a right rotation

This *right rotation* is often referred to as a right rotation of the left child around the parent. The last image in Figure 11.10 shows the same tree after a right rotation. The same kind of rotation can be done at any level of the tree. This single rotation to the right will solve the imbalance if the imbalance is caused by a long path length in the left subtree of the left child of the root.

Left Rotation

Figure 11.11 shows another binary search tree that is not balanced. Again, the maximum path length in this tree is 3 and the minimum path length is 1. However,

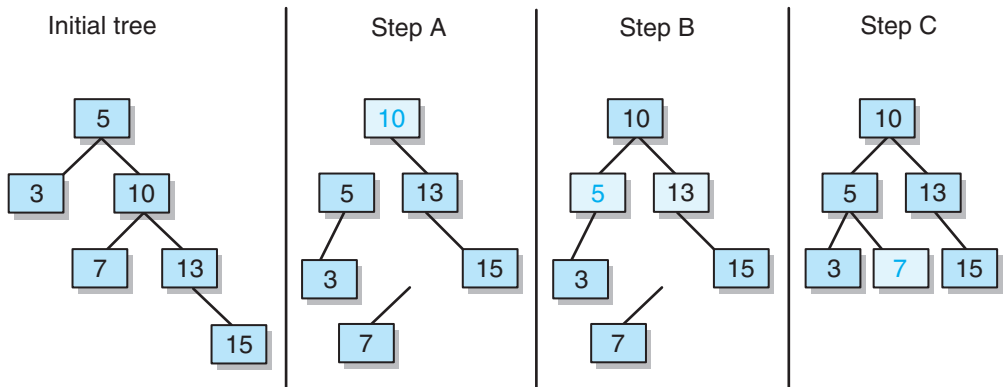


FIGURE 11.11 An unbalanced tree and the balanced tree that results from a left rotation

this time the larger path length is in the right subtree of the right child of the root. To get this tree into balance, we need to

- Make the right child element of the root the new root element.
- Make the former root element the left child element of the new root.
- Make the left child of what was the right child of the former root the new right child of the former root.

This *left rotation* is often referred to as a left rotation of the right child around the parent. Figure 11.11 follows the same tree through the processing steps of a left rotation. The same kind of rotation can be done at any level of the tree. This single rotation to the left will solve the imbalance if the imbalance is caused by a long path length in the right subtree of the right child of the root.

Rightleft Rotation

Unfortunately, not all imbalances can be solved by single rotations. If the imbalance is caused by a long path length in the left subtree of the right child of the root, we must first perform a right rotation of the left child of the right child of the root around the right child of the root, and then perform a left rotation of the resulting right child of the root around the root. Figure 11.12 illustrates this process.

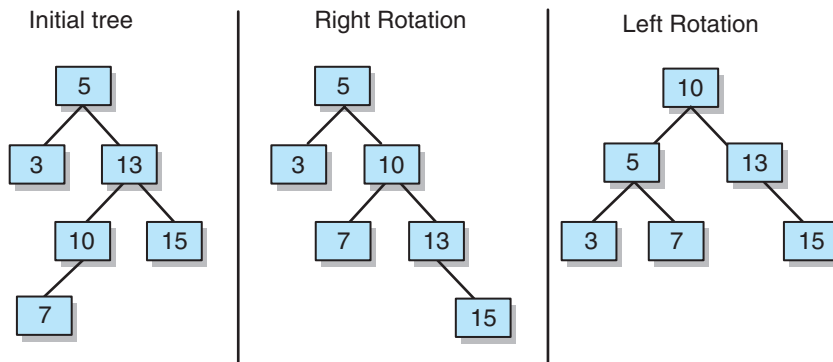


FIGURE 11.12 A rightleft rotation

Leftright Rotation

Similarly, if the imbalance is caused by a long path length in the right subtree of the left child of the root, we must first perform a left rotation of the right child of the left child of the root around the left child of the root, and then perform a right rotation of the resulting left child of the root around the root. Figure 11.13 illustrates this process.

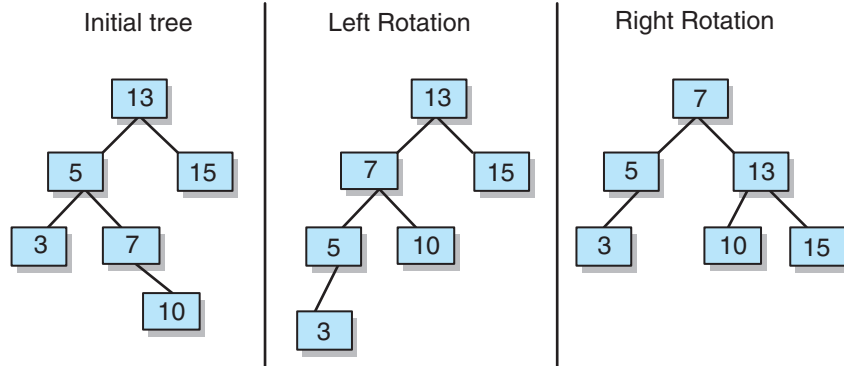


FIGURE 11.13 A left-right rotation



VideoNote

Demonstration of the four basic tree rotations

11.5 Implementing BSTs: AVL Trees

We have been discussing a generic method for balancing a tree, where the maximum path length from the root must be no more than $\log_2 n$ and the minimum path length from the root must be no less than $\log_2 n - 1$. Adel'son-Vel'skii and Landis developed a method called *AVL trees* that is a variation on this theme. For each node in the tree, we will keep track of the height of the left and right subtrees. For any node in the tree, if the *balance factor*, or the difference in the heights of its subtrees (height of the right subtree minus height of the left subtree), is greater than 1 or less than -1 , then the subtree with that node as the root needs to be rebalanced.

KEY CONCEPT

The height of the right subtree minus the height of the left subtree is called the balance factor of a node.

KEY CONCEPT

There are only two ways in which a tree, or any subtree of a tree, can become unbalanced: through the insertion of a node or through the deletion of a node.

There are only two ways in which a tree, or any subtree of a tree, can become unbalanced: through the insertion of a node or through the deletion of a node. Thus, each time one of these operations is performed, the balance factors must be updated, and the balance of the tree must be checked starting at the point of insertion or removal of a node and working up toward the root of the tree. Because of this need to work back up the tree, AVL trees are often best implemented by including a parent reference in each node. In the diagrams that follow, all edges are represented as a single bidirectional line.

The cases for rotation that we discussed in the last section apply here as well, and by using this method, we can easily identify when to use each.

Right Rotation in an AVL Tree

If the balance factor of a node is -2 , this means that the node's left subtree has a path that is too long. We then check the balance factor of the left child of the original node. If the balance factor of the left child is -1 , this means that the long path is in the left subtree of the left child, and therefore a simple right rotation of the left child around the original node will rebalance the tree. Figure 11.14 shows how an insertion of a node could cause an imbalance and how a right rotation would resolve it. Note that we are representing both the values stored at each node and the balance factors, with the balance factors shown in parentheses.

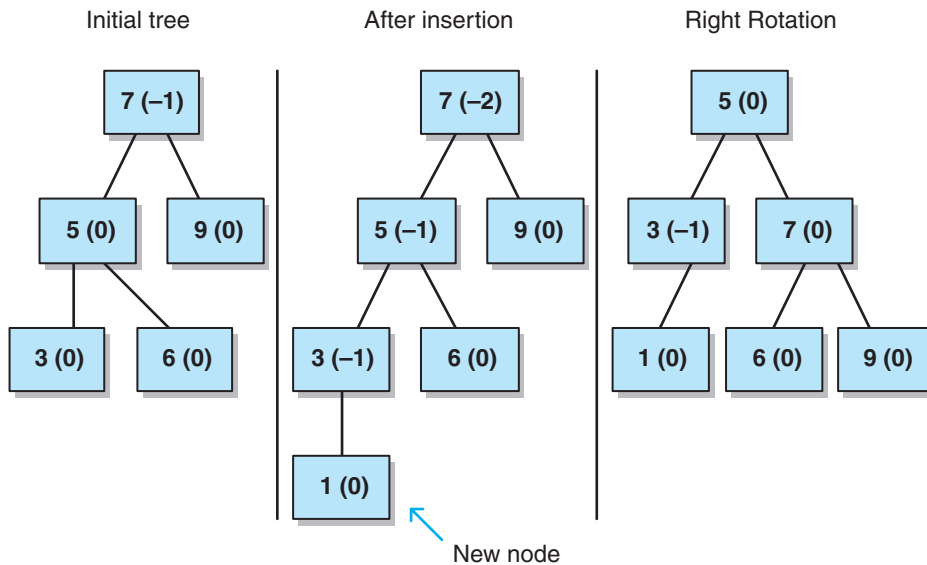


FIGURE 11.14 A right rotation in an AVL tree

Left Rotation in an AVL Tree

If the balance factor of a node is $+2$, this means that the node's right subtree has a path that is too long. We then check the balance factor of the right child of the original node. If the balance factor of the right child is $+1$, this means that the long path is in the right subtree of the right child, and therefore a simple left rotation of the right child around the original node will rebalance the tree.

Rightleft Rotation in an AVL Tree

If the balance factor of a node is $+2$, this means that the node's right subtree has a path that is too long. We then check the balance factor of the right child of the original node. If the balance factor of the right child is -1 , this means that the long path is in the left subtree of the right child, and therefore a rightleft double

rotation will rebalance the tree. This is accomplished by first performing a right rotation of the left child of the right child of the original node around the right child of the original node, and then performing a left rotation of the right child of the original node around the original node. Figure 11.15 shows how the removal

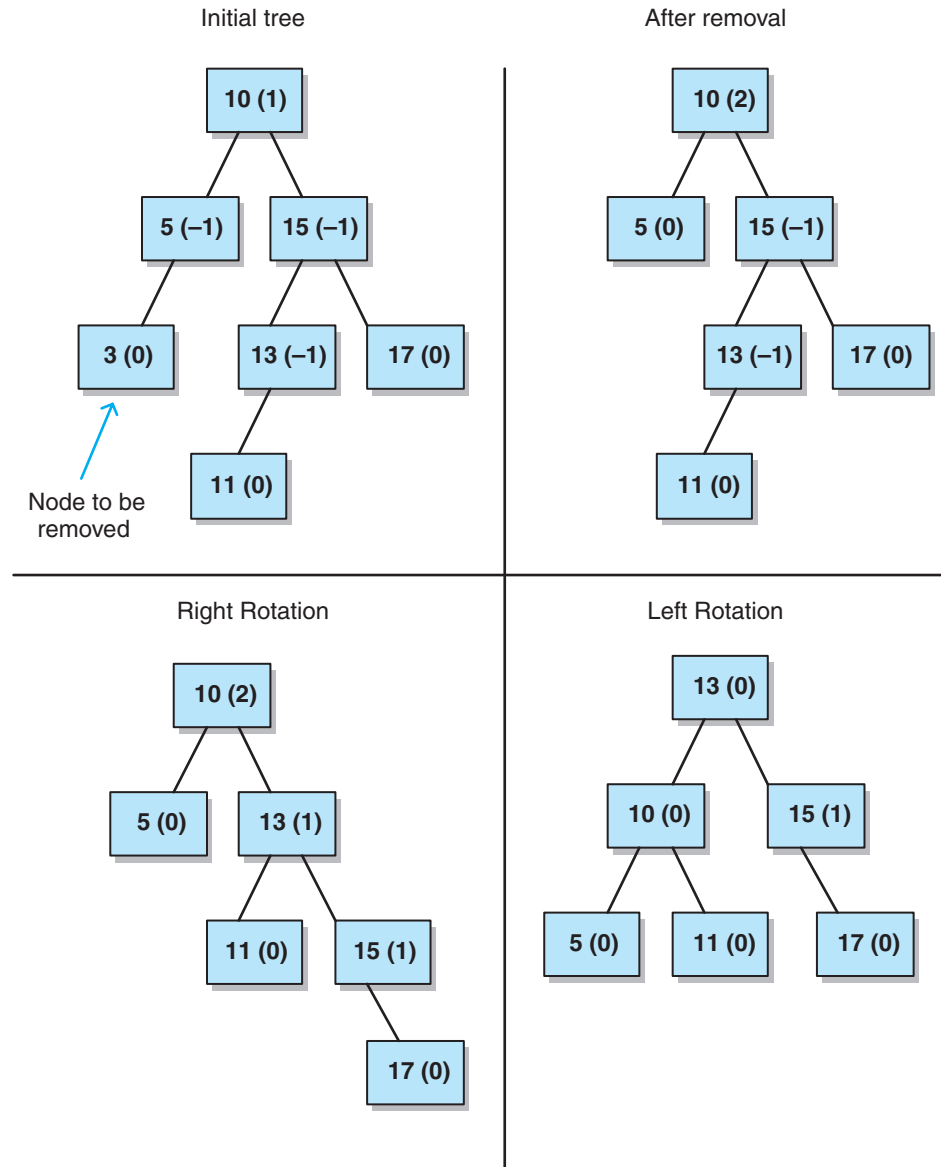


FIGURE 11.15 A rightleft rotation in an AVL tree

of an element from the tree could cause an imbalance and how a rightleft rotation would resolve it. Again, note that we are representing both the values stored at each node and the balance factors, with the balance factors shown in parentheses.

Leftright Rotation in an AVL Tree

If the balance factor of a node is -2 , this means that the node's left subtree has a path that is too long. We then check the balance factor of the left child of the original node. If the balance factor of the left child is $+1$, this means that the long path is in the right subtree of the left child, and therefore a leftright double rotation will rebalance the tree. This is accomplished by first performing a left rotation of the right child of the left child of the original node around the left child of the original node, and then performing a right rotation of the left child of the original node around the original node.

11.6 Implementing BSTs: Red/Black Trees

Another alternative to the implementation of binary search trees is the concept of a *red/black tree*, which was developed by Bayer and extended by Guibas and Sedgwick. A red/black tree is a balanced binary search tree in which we will store a color with each node (either red or black, usually implemented as a `boolean` value with `false` being equivalent to red). The following rules govern the color of a node:

- The root is black.
- All children of a red node are black.
- Every path from the root to a leaf contains the same number of black nodes.

Figure 11.16 shows three valid red/black trees (the lighter-shade nodes are “red”). Notice that the balance restriction on a red/black tree is somewhat less strict than that for AVL trees or for our earlier theoretical discussion. However, finding an element in both implementations is still an $O(\log n)$ operation. Because no red node can have a red child, at most half of the nodes in a path could be red nodes and at least half of the nodes in a path are black. From this we can argue that the maximum height of a red/black tree is roughly $2 \cdot \log n$, and thus the traversal of the longest path is still order $\log n$.

KEY CONCEPT

The balance restriction on a red/black tree is somewhat less strict than that for AVL trees.

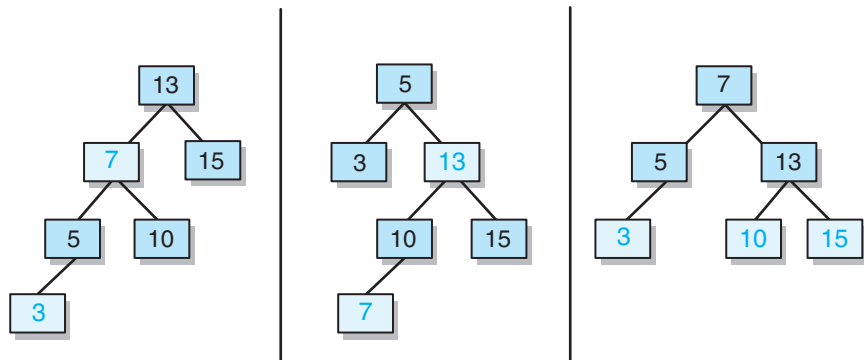


FIGURE 11.16 Valid red/black trees

As with AVL trees, the only time we need to be concerned about balance is after an insertion or removal of an element in the tree. But unlike the case with AVL trees, insertion and removal are handled quite separately.

Insertion into a Red/Black Tree

Insertion into a red/black tree will progress much as it did in our earlier `addElement` method. However, we will always begin by setting the color of the new element to red. Once the new element has been inserted, we then rebalance the tree as needed and change the color of elements as needed to maintain the properties of a red/black tree. As a last step, we always set the color of the root of the tree to black. For purposes of our discussion, we will simply refer to the color of a node as `node.color`. However, it may be more elegant in an actual implementation to create a method to return the color of a node.

The rebalancing (and recoloring) process after insertion is an iterative (or recursive) one starting at the point of insertion and working up the tree toward the root. Therefore, like AVL trees, red/black trees are best implemented by including a parent reference in each node. The termination conditions for this process are (`current == root`), where `current` is the node we are currently processing, or (`current.parent.color == black`) (that is, the color of the parent of the current node is black). The first condition terminates the process because we will always set the root color to black, and the root is included in all paths and therefore cannot violate the rule that each path have the same number of black elements. The second condition terminates the process, because the node pointed to by `current` will always be a red node. This means that if the parent of the current node is black, then all of the rules are met as well since a red node does not affect

the number of black nodes in a path, and because we are working from the point of insertion up, we will already have balanced the subtree under the current node.

In each iteration of the rebalancing process, we will focus on the color of the sibling of the parent of the current node. Keep in mind that there are two possibilities for the parent of the current node: `current.parent` could be a left child or a right child. Assuming that the parent of `current` is a right child, we can get the color information by using `current.parent.parent.left.color`, but for purposes of our discussion, we will use the terms `parentsleftsibling.color` and `parentsrightsibling.color`. It is also important to keep in mind that the color of a null element is considered to be black.

In the case where the parent of `current` is a right child, there are two cases: (`parentsleftsibling.color == red`) or (`parentsleftsibling.color == black`). Keep in mind that in either case, we are describing processing steps that are occurring inside of a loop with the termination conditions described earlier. Figure 11.17 shows a red/black tree after insertion with this first case (`parentsleftsibling.color == red`). The processing steps in this case are

- Set the color of `current`'s parent to black.
- Set the color of `parentsleftsibling` to black.
- Set the color of `current`'s grandparent to red.
- Set `current` to point to the grandparent of `current`.

In Figure 11.17, we inserted 8 into our tree. Keep in mind that `current` points to our new node and that `current.color` is set to red. Following the processing steps, we set the parent of `current` to black, we set the left sibling of the parent of `current` to black, and we set the grandparent of `current` to red. We then set `current` to point to the grandparent. Because the grandparent is the root, the loop terminates. Finally, we set the root of the tree to black.

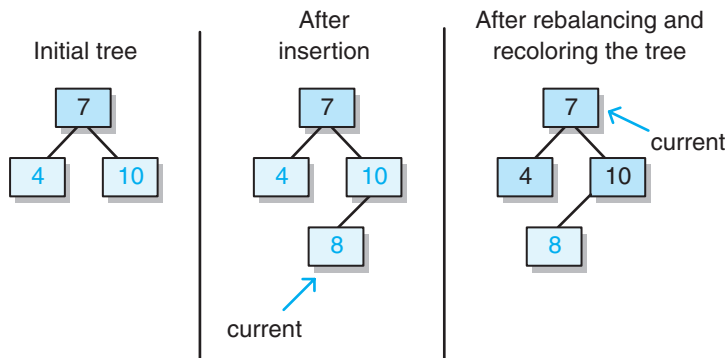


FIGURE 11.17 Red/black tree after insertion

However, if (`parentsleftsibling.color == black`), then we first need to check to see whether `current` is a left or a right child. If `current` is a left child, then we must set `current` equal to its parent and then rotate `current.left` to the right (around `current`) before continuing. Once this is accomplished, the processing steps are the same as they would be if `current` had been a right child to begin with:

- Set the color of `current`'s parent to black.
- Set the color of `current`'s grandparent to red.
- If `current`'s grandparent does not equal null, then rotate `current`'s parent to the left around `current`'s grandparent.

In the case where the parent of `current` is a left child, there are two cases: (`parentsrightsibling.color == red`) or (`parentsrightsibling.color == black`). Keep in mind that in either case, we are describing processing steps that are occurring inside of a loop with the termination conditions described earlier. Figure 11.18 shows a red/black tree after insertion in this case (`parentsrightsibling.color == red`). The processing steps in this case are

- Set the color of `current`'s parent to black.
- Set the color of `parentsrightsibling` to black.
- Set the color of `current`'s grandparent to red.
- Set `current` to point to the grandparent of `current`.

In Figure 11.18 we inserted 5 into our tree, setting `current` to point to the new node and setting `current.color` to red. Again, following our processing steps,

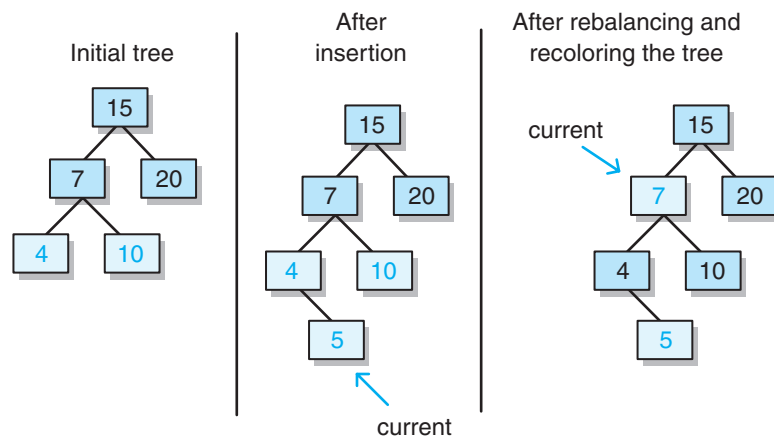


FIGURE 11.18 Red/black tree after insertion

we set the parent of `current` to black, we set the right sibling of the parent of `current` to black, and we set the grandparent of `current` to red. We then set `current` to point to its grandparent. Because the parent of the new `current` is black, our loop terminates. Last, we set the color of the root to black.

If (`parentsrightsibling.color == black`), then we first need to check to see whether `current` is a left or a right child. If `current` is a right child, then we must set `current` equal to `current.parent` and then rotate `current.right` to the left (around `current`) before continuing. Once this is accomplished, the processing steps are the same as they would be if `current` had been a left child to begin with:

- Set the color of `current`'s parent to black.
- Set the color of `current`'s grandparent to red.
- If `current`'s grandparent does not equal null, then rotate `current`'s parent to the right around `current`'s grandparent.

As you can see, the cases, depending on whether `current`'s parent is a left or a right child, are symmetrical.

Element Removal from a Red/Black Tree

As with insertion, the `removeElement` operation behaves much as it did before, only with the additional step of rebalancing (and recoloring) the tree. This rebalancing (and recoloring) process after removal of an element is an iterative one starting at the point of removal and working up the tree toward the root. Therefore, as stated earlier, red/black trees are often best implemented by including a parent reference in each node. The termination conditions for this process are (`current == root`), where `current` is the node we are currently processing, or (`current.color == red`).

As with the cases for insertion, the cases for removal are symmetrical depending on whether `current` is a left or a right child. We will examine only the case where `current` is a right child. The other cases are easily derived by simply substituting left for right and right for left in the following cases.

In insertion, we were most concerned with the color of the sibling of the parent of the current node. For removal, we will focus on the color of the sibling of `current`. We could reference this color using `current.parent.left.color`, but we will simply refer to it as `sibling.color`. We will also look at the color of the children of the sibling. It is important to note that the default for color is black. Therefore, if at any time we are attempting to get the color of a null object, the result will be black. Figure 11.19 shows a red/black tree after the removal of an element.

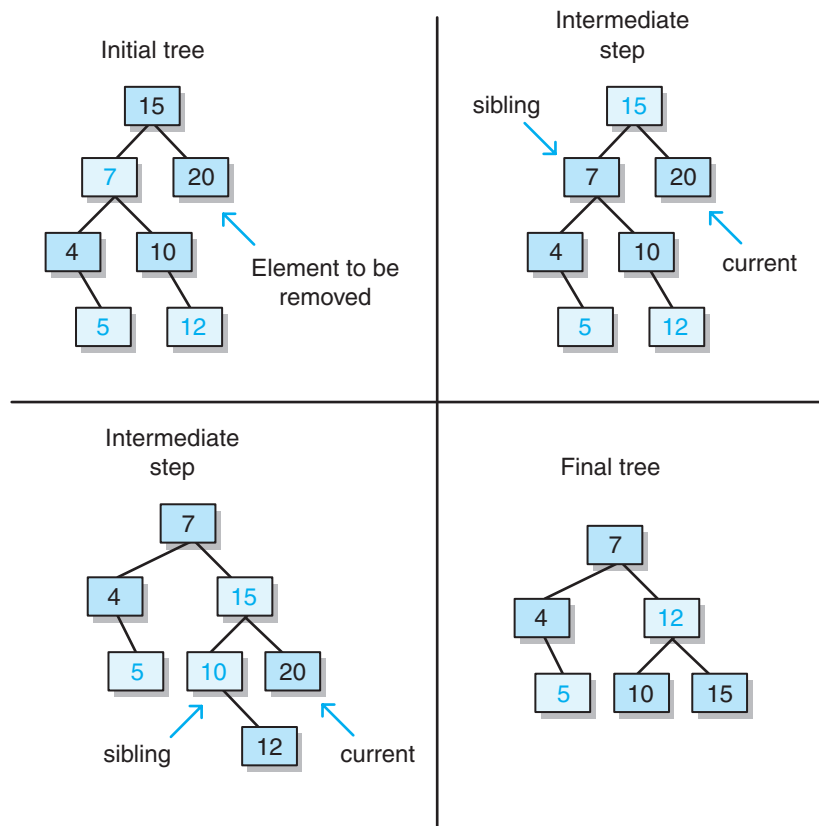


FIGURE 11.19 Red/black tree after removal

If the sibling’s color is red, then before we do anything else, we must complete the following processing steps:

- Set the color of the sibling to black.
- Set the color of *current*’s parent to red.
- Rotate the sibling right around *current*’s parent.
- Set the sibling equal to the left child of *current*’s parent.

Next, our processing continues regardless of whether the original sibling was red or black. Now our processing is divided into one of two cases based on the color of the children of the sibling. If both children of the sibling are black (or null), then we do the following:

- Set the color of the sibling to red.
- Set *current* equal to *current*’s parent.

If the children of the sibling are not both black, then we check to see whether the left child of the sibling is black. If it is, we must complete the following steps before continuing:

- Set the color of the sibling's right child to black.
- Set the color of the sibling to red.
- Rotate the sibling's right child left around the sibling.
- Set the sibling equal to the left child of `current`'s parent.

Then to complete the process when both of the sibling's children are not black, we must

- Set the color of the sibling to the color of `current`'s parent.
- Set the color of `current`'s parent to black.
- Set the color of the sibling's left child to black.
- Rotate the sibling right around `current`'s parent.
- Set `current` equal to the root.

Once the loop terminates, we must always then remove the node and set its parent's child reference to null.

Summary of Key Concepts

- A binary search tree is a binary tree with the added property that the left child is less than the parent, which is less than or equal to the right child.
- The definition of a binary search tree is an extension of the definition of a binary tree.
- Each `BinaryTreeNode` object maintains a reference to the element stored at that node, as well as references to each of the node's children.
- In removing an element from a binary search tree, another node must be promoted to replace the node being removed.
- The leftmost node in a binary search tree will contain the minimum element, whereas the rightmost node will contain the maximum element.
- One of the uses of trees is to provide efficient implementations of other collections.
- If a binary search tree is not balanced, it may be less efficient than a linear structure.
- The height of the right subtree minus the height of the left subtree is called the balance factor of a node.
- There are only two ways in which a tree, or any subtree of a tree, can become unbalanced: through the insertion of a node and through the deletion of a node.
- The balance restriction on a red/black tree is somewhat less strict than that for AVL trees. However, in both cases, the `find` operation is order $\log n$.

Summary of Terms

binary search tree A binary tree with the added property that, for each node, the left child is less than the parent, which is less than or equal to the right child.

promoted A term used to describe the concept of a node in a tree being moved up to replace a parent node or other ancestor node that is being removed from the tree.

degenerate tree A tree that does not branch.

right rotation A single rotation strategy for rebalancing a tree when the long path is in the left subtree of the left child of the root.

left rotation A single rotation strategy for rebalancing a tree when the long path is in the right subtree of the right child of the root.

rightleft rotation A double rotation strategy for rebalancing a tree when the long path is in the left subtree of the right child of the root.

leftright rotation A double rotation strategy for rebalancing a tree when the long path is in the right subtree of the left child of the root.

AVL trees A strategy for keeping a binary search tree balanced that makes use of the balance factor of each node.

balance factor A property of a node that is computed by subtracting the height of the left subtree from the height of the right subtree. If the result is either greater than 1 or less than -1 , then the tree is unbalanced.

red/black trees A strategy for keeping a binary search tree balanced that makes use of a color (either red or black) associated with each node.

Self-Review Questions

- SR 11.1 What is the difference between a binary tree and a binary search tree?
- SR 11.2 Why are we able to specify `addElement` and `removeElement` operations for a binary search tree but unable to do so for a binary tree?
- SR 11.3 Assuming that the tree is balanced, what is the time complexity (order) of the `addElement` operation?
- SR 11.4 Without the balance assumption, what is the time complexity (order) of the `addElement` operation?
- SR 11.5 As stated in this chapter, a degenerate tree might actually be less efficient than a linked list. Why?
- SR 11.6 Our `removeElement` operation uses the inorder successor as the replacement for a node with two children. What would be another reasonable choice for the replacement?
- SR 11.7 The `removeAllOccurrences` operation uses both the `contains` operation and the `removeElement` operation. What is the resulting time complexity (order) for this operation?
- SR 11.8 `RemoveFirst` and `first` were $O(1)$ operations for our earlier implementation of an ordered list. Why are they less efficient for our `BinarySearchTreeOrderedList`?
- SR 11.9 Why does the `BinarySearchTreeOrderedList` class have to define the `iterator` method? Why can't it just rely on the `iterator` method of its parent class, as it does for `size` and `isEmpty`?

- SR 11.10 What is the time complexity of the `addElement` operation after modifying to implement an AVL tree?
- SR 11.11 What imbalance is fixed by a single right rotation?
- SR 11.12 What imbalance is fixed by a leftright rotation?
- SR 11.13 What is the balance factor of an AVL tree node?
- SR 11.14 In our discussion of the process for rebalancing an AVL tree, we never discussed the possibility of the balance factor of a node being either +2 or -2 and the balance factor of one of its children being either +2 or -2. Why not?
- SR 11.15 We noted that the balance restriction for a red/black tree is less strict than that of an AVL tree, and yet we still claim that traversing the longest path in a red/black tree is still $O(\log n)$. Why?

Exercises

- EX 11.1 Draw the binary search tree that results from adding the integers (34 45 3 87 65 32 1 12 17). Assume our simple implementation with no balancing mechanism.
- EX 11.2 Starting with the tree resulting from Exercise 11.1, draw the tree that results from removing (45 12 1), again using our simple implementation with no balancing mechanism.
- EX 11.3 Repeat Exercise 11.1, this time assuming an AVL tree. Include the balance factors in your drawing.
- EX 11.4 Repeat Exercise 11.2, this time assuming an AVL tree and using the result of Exercise 11.3 as a starting point. Include the balance factors in your drawing.
- EX 11.5 Repeat Exercise 11.1, this time assuming a red/black tree. Label each node with its color.
- EX 11.6 Repeat Exercise 11.2, this time assuming a red/black tree and using the result of Exercise 11.5 as a starting point. Label each node with its color.
- EX 11.7 Starting with an empty red/black tree, draw the tree after insertion and before rebalancing, and after rebalancing (if necessary) for the following series of inserts and removals:

```
addElement (40) ;
addElement (25) ;
addElement (10) ;
addElement (5) ;
```

```
addElement (1) ;  
addElement (45) ;  
addElement (50) ;  
removeElement (40) ;  
removeElement (25) ;
```

EX 11.8 Repeat Exercise 11.7, this time with an AVL tree.

Programming Projects

- PP 11.1 Develop an array implementation of a binary search tree using the computational strategy described in Chapter 10.
- PP 11.2 The `LinkedBinarySearchTree` class is currently using the `find` and contains methods of the `LinkedBinaryTree` class. Implement these methods for the `LinkedBinarySearchTree` class so that they will be more efficient by making use of the ordering property of a binary search tree.
- PP 11.3 Implement the `removeMax`, `findMin`, and `findMax` operations for our linked binary search tree implementation.
- PP 11.4 Modify the linked implementation of a binary tree so that it will no longer allow duplicates.
- PP 11.5 Implement a balance tree method for the linked implementation using the brute force method described in Section 11.4.
- PP 11.6 Implement a balance tree method for the array implementation from Project 11.1 using the brute force method described in Section 11.4.
- PP 11.7 Develop an array implementation of a binary search tree built upon an array implementation of a binary tree by using the simulated link strategy. Each element of the array will need to maintain both a reference to the data element stored there and the array positions of the left child and the right child. You also need to maintain a list of available array positions where elements have been removed, in order to reuse those positions.
- PP 11.8 Modify the linked binary search tree implementation to make it an AVL tree.
- PP 11.9 Modify the linked binary search tree implementation to make it a red/black tree.
- PP 11.10 Modify the add operation for the linked implementation of a binary search tree to use an iterative algorithm.

Answers to Self-Review Questions

- SRA 11.1 A binary search tree has the added ordering property that the left child of any node is less than the node, and the node is less than or equal to its right child.
- SRA 11.2 With the added ordering property of a binary search tree, we are now able to define what the state of the tree should be after an `add` or `remove`. We were unable to define that state for a binary tree.
- SRA 11.3 If the tree is balanced, finding the insertion point for the new element will take at worst $\log n$ steps, and since inserting the element is simply a matter of setting the value of one reference, the operation is $O(\log n)$.
- SRA 11.4 Without the balance assumption, the worst case would be a degenerate tree, which is effectively a linked list. Therefore, the `addElement` operation would be $O(n)$.
- SRA 11.5 A degenerate tree will waste space with unused references, and many of the algorithms will check for null references before following the degenerate path, thus adding steps that the linked list implementation does not have.
- SRA 11.6 The best choice is the inorder successor because we are placing equal values to the right.
- SRA 11.7 With our balance assumption, the `contains` operation uses the `find` operation, which will be rewritten in the `BinarySearchTree` class to take advantage of the ordering property and will be $O(\log n)$. The `removeElement` operation is $O(\log n)$. The `while` loop will iterate some constant (k) number of times, depending on how many times the given element occurs within the tree. The worst case would be that all n elements of the tree were the element to be removed, which would make the tree degenerate, and in which case the complexity would be $n * 2 * n$ or $O(n^2)$. However, the expected case would be some small constant ($0 \leq k < n$) occurrences of the element in a balanced tree, which would result in a complexity of $k * 2 * \log n$ or $O(\log n)$.
- SRA 11.8 In our earlier linked implementation of an ordered list, we had a reference that kept track of the first element in the list, which made it quite simple to remove it or return it. With a binary search tree, we have to traverse to get to the leftmost element before knowing that we have the first element in the ordered list.

- SRA 11.9 Remember that the iterators for a binary tree are all followed by which traversal order to use. That is why the `iterator` method for the `BinarySearchTreeOrderedList` class calls the `iteratorInOrder` method of the `BinaryTree` class.
- SRA 11.10 Keep in mind that an `addElement` method affects only one path of the tree, which in a balanced AVL tree has a maximum length of $\log n$. As we have discussed previously, finding the position to insert and setting the reference is $O(\log n)$. We then have to progress back up the same path, updating the balance factors of each node (if necessary) and rotating if necessary. Updating the balance factors is an $O(1)$ step, and rotation is also an $O(1)$ step. Each of these will have to be done at most $\log n$ times. Therefore, `addElement` has time complexity $2 \cdot \log n$ or $O(\log n)$.
- SRA 11.11 A single right rotation will fix the imbalance if the long path is in the left subtree of the left child of the root.
- SRA 11.12 A leftright rotation will fix the imbalance if the long path is in the right subtree of the left child of the root.
- SRA 11.13 The balance factor of an AVL tree node is the height of the right subtree minus the height of the left subtree.
- SRA 11.14 Rebalancing an AVL tree is done after either an insertion or a deletion, and it is done starting at the affected node and working up along a single path to the root. As we progress upward, we update the balance factors and rotate if necessary. We will never encounter a situation where both a child and a parent have balance factors of ± 2 because we would have already fixed the child before we ever reached the parent.
- SRA 11.15 Because no red node can have a red child, at most half of the nodes in a path could be red nodes, and at least half of the nodes in a path are black. From this we can argue that the maximum height of a red/black tree is roughly $2 \cdot \log n$, and thus the traversal of the longest path is $O(\log n)$.

References

- Adel'son-Vel'skii, G. M., and E. M. Landis. "An Algorithm for the Organization of Information." *Soviet Mathematics* 3 (1962): 1259–1263.
- Bayer, R. "Symmetric Binary B-trees: Data Structure and Maintenance Algorithms." *Acta Informatica* (1972): 290–306.

- Collins, W. J. *Data Structures and the Java Collections Framework*. New York: McGraw-Hill, 2002.
- Cormen, T., C. Leieron, and R. Rivest. *Introduction to Algorithms*. New York: McGraw-Hill, 1992.
- Guibas, L., and R. Sedgewick. "A Diochromatic Framework for Balanced Trees." *Proceedings of the 19th Annual IEEE Symposium on Foundations of Computer Science* (1978): 8–21.



Heaps and Priority Queues

12

In this chapter, we will look at another ordered extension of binary trees. We will examine heaps, including both linked and array implementations, and the algorithms for adding and removing elements from a heap. We will also examine some uses for heaps, including the implementation of priority queues.

CHAPTER OBJECTIVES

- Define a heap abstract data structure.
- Demonstrate how a heap can be used to solve problems.
- Examine various heap implementations.
- Compare heap implementations.

12.1 A Heap

A *heap* is a binary tree with two added properties:

- It is a complete tree, as described in Chapter 10.
- For each node, the node is less than or equal to both the left child and the right child.

KEY CONCEPT

A minheap is a complete binary tree in which each node is less than or equal to both of its children.

KEY CONCEPT

A minheap stores its smallest element at the root of the binary tree, and both children of the root of a minheap are also minheaps.

This definition describes a *minheap*. A heap can also be a *max-heap*, in which the node is greater than or equal to its children. We will focus our discussion in this chapter on minheaps. All of the same processes work for maxheaps by reversing the comparisons.

Figure 12.1 describes the operations on a heap. The heap is defined as an extension of a binary tree and thus inherits all of those operations as well. Note that because the implementation of a binary tree does not have any operations to add or remove elements from the tree, there are not any operations that would violate the properties of a heap. Listing 12.1 shows the interface definition for a heap. Figure 12.2 shows the UML description of the `HeapADT`.

Operation	Description
<code>addElement</code>	Adds the given element to the heap.
<code>removeMin</code>	Removes the minimum element in the heap.
<code>findMin</code>	Returns a reference to the minimum element in the heap.

FIGURE 12.1 The operations on a heap

LISTING 12.1

```
package jsjf;

/**
 * HeapADT defines the interface to a Heap.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public interface HeapADT<T> extends BinaryTreeADT<T>
{
    /**
     * Adds the specified object to this heap.
     *
     */
}
```

LISTING 12.1

continued

```

    * @param obj the element to be added to the heap
    */
    public void addElement(T obj);

    /**
     * Removes element with the lowest value from this heap.
     *
     * @return the element with the lowest value from the heap
     */
    public T removeMin();

    /**
     * Returns a reference to the element with the lowest value in
     * this heap.
     *
     * @return a reference to the element with the lowest value in the heap
     */
    public T findMin();
}

```

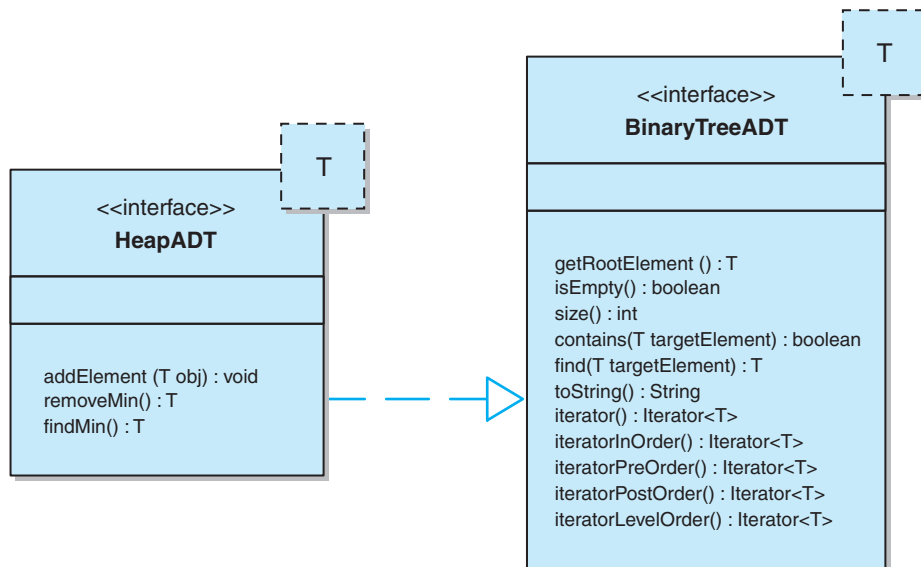


FIGURE 12.2 UML description of the HeapADT

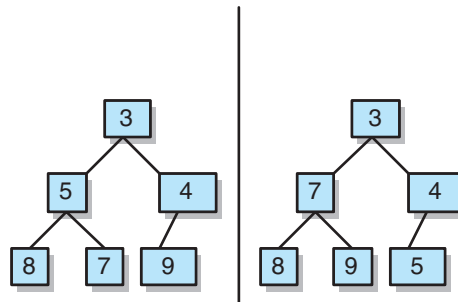


FIGURE 12.3 Two minheaps containing the same data

Simply put, a minheap will always store its smallest element at the root of the binary tree, and both children of the root of a minheap are also minheaps. Figure 12.3 illustrates two valid minheaps with the same data. Let's look at the basic operations on a heap and examine generic algorithms for each.

The addElement Operation

The `addElement` method adds a given element to the appropriate location in the heap, maintaining both the completeness property and the ordering property of the heap. This method throws a `ClassCastException` if the given element is not `Comparable`.

KEY CONCEPT

The `addElement` method adds a given `Comparable` element to the appropriate location in the heap, maintaining both the completeness property and the ordering property of the heap.

A binary tree is considered *complete* if it is balanced, meaning that all of the leaves are at level h or $h - 1$, where h is $\log_2 n$ and n is the number of elements in the tree, and all of the leaves at level h are on the left side of the tree. Because a heap is a complete tree, there is only one correct location for the insertion of a new node, and that is either the next open position from the left at level h or, if level h is full, the first position on the left at level $h + 1$. Figure 12.4 illustrates these two possibilities.

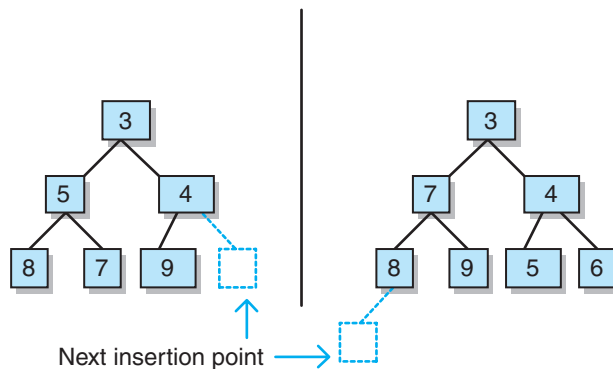


FIGURE 12.4 Insertion points for a heap

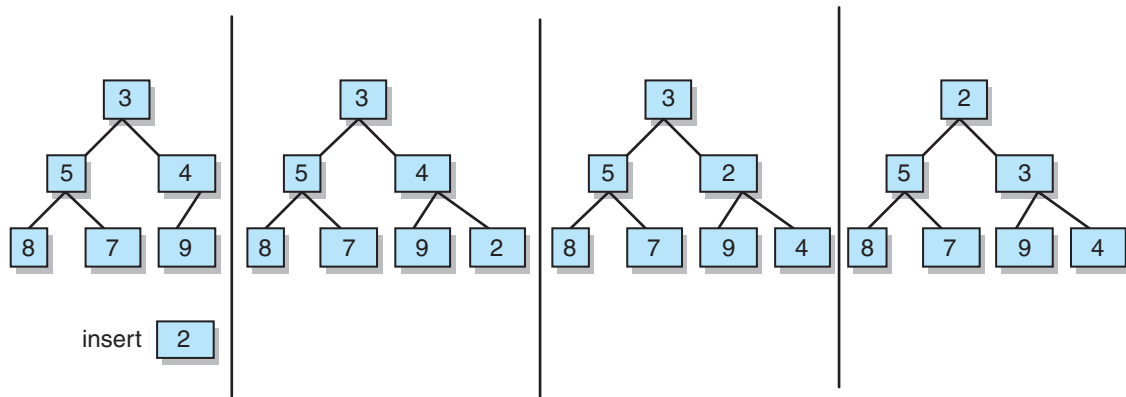


FIGURE 12.5 Insertion and reordering in a heap

Once we have located the new node in the proper position, we must account for the ordering property. To do this, we simply compare the new value to its parent value and swap the values if the new node is less than its parent. We continue this process up the tree until the new value either is greater than its parent or is in the root of the heap. Figure 12.5 illustrates this process for inserting a new element into a heap. Typically, in heap implementations, we keep track of the position of the last node or, more precisely, the last leaf in the tree. After an `addElement` operation, the last node is set to the node that was inserted.

The `removeMin` Operation

The `removeMin` method removes the minimum element from the minheap and returns it. Because the minimum element is stored in the root of a minheap, we need to return the element stored at the root and replace it with another element in the heap. As with the `addElement` operation, to maintain the completeness of the tree, there is only one valid element to replace the root, and that is the element stored in the last leaf in the tree. This last leaf will be the rightmost leaf at level h of the tree. Figure 12.6 illustrates this concept of the last leaf under a variety of circumstances.

Once the element stored in the last leaf has been moved to the root, the heap will then have to be reordered to maintain the heap's ordering property. This is accomplished by comparing the new root

KEY CONCEPT

Because a heap is a complete tree, there is only one correct location for the insertion of a new node, and that is either the next open position from the left at level h or, if level h is full, the first position on the left at level $h + 1$.

KEY CONCEPT

Typically, in heap implementations, we keep track of the position of the last node or, more precisely, the last leaf in the tree.

KEY CONCEPT

To maintain the completeness of the tree, there is only one valid element to replace the root, and that is the element stored in the last leaf in the tree.

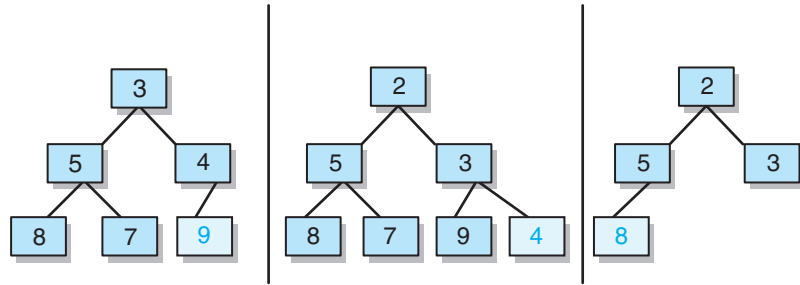


FIGURE 12.6 Examples of the last leaf in a heap

element to the smaller of its children and then swapping them if the child is smaller. This process is repeated on down the tree until the element either is in a leaf or is less than both of its children. Figure 12.7 illustrates the process of removing the minimum element and then reordering the tree.

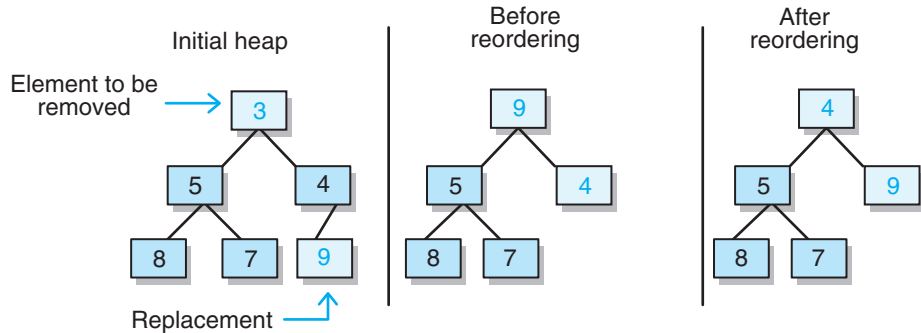


FIGURE 12.7 Removal and reordering in a heap

The findMin Operation

The `findMin` method returns a reference to the smallest element in the minheap. Because that element is always stored in the root of the tree, this method is simply implemented by returning the element stored in the root.

12.2 Using Heaps: Priority Queues

A *priority queue* is a collection that follows two ordering rules. First, items with higher priority go first. Second, items with the same priority are ordered in accordance with the first in, first out principle. Priority queues have a variety of

applications (such as task scheduling in an operating system, traffic scheduling on a network, and even job scheduling at your local auto mechanic).

A priority queue could be implemented using a list of queues where each queue represents items of a given priority. Another solution to this problem is to use a minheap. Sorting the heap by priority accomplishes the first ordering (higher-priority items go first). However, the first in, first out ordering of items with the same priority is something we will have to manipulate. The solution is to create a `PrioritizedObject` object that stores the element to be placed on the queue, the priority of the element, and the order in which elements are placed on the queue. Then, we simply define the `compareTo` method for the `PrioritizedObject` class to compare priorities first and then compare order if there is a tie. Listing 12.2 shows the `PrioritizedObject` class, and Listing 12.3 shows the `PriorityQueue` class. The UML description of the `PriorityQueue` class is left as an exercise.

KEY CONCEPT

Even though it is not a queue at all, a minheap provides an efficient implementation of a priority queue.

LISTING 12.2

```
/**
 * PrioritizedObject represents a node in a priority queue containing a
 * comparable object, arrival order, and a priority value.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class PrioritizedObject<T> implements Comparable<PrioritizedObject>
{
    private static int nextOrder = 0;
    private int priority;
    private int arrivalOrder;
    private T element;

    /**
     * Creates a new PrioritizedObject with the specified data.
     *
     * @param element the element of the new priority queue node
     * @param priority the priority of the new queue node
     */
    public PrioritizedObject(T element, int priority)
    {
        this.element = element;
        this.priority = priority;
        arrivalOrder = nextOrder;
        nextOrder++;
    }
}
```

LISTING 12.2

continued

```
/**
 * Returns the element in this node.
 *
 * @return the element contained within the node
 */
public T getElement()
{
    return element;
}

/**
 * Returns the priority value for this node.
 *
 * @return the integer priority for this node
 */
public int getPriority()
{
    return priority;
}

/**
 * Returns the arrival order for this node.
 *
 * @return the integer arrival order for this node
 */
public int getArrivalOrder()
{
    return arrivalOrder;
}

/**
 * Returns a string representation for this node.
 *
 */
public String toString()
{
    return (element + " " + priority + " " + arrivalOrder);
}

/**
 * Returns 1 if the this object has higher priority than
 * the given object and -1 otherwise.
 *
 * @param obj the object to compare to this node
 * @return the result of the comparison of the given object and
 *         this one
 */
```

LISTING 12.2 *continued*

```
public int compareTo(PrioritizedObject obj)
{
    int result;

    if (priority > obj.getPriority())
        result = 1;
    else if (priority < obj.getPriority())
        result = -1;
    else if (arrivalOrder > obj.getArrivalOrder())
        result = 1;
    else
        result = -1;
    return result;
}
}
```

LISTING 12.3

```
import jsjf.*;

/**
 * PriorityQueue implements a priority queue using a heap.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class PriorityQueue<T> extends ArrayHeap<PrioritizedObject<T>>
{
    /**
     * Creates an empty priority queue.
     */
    public PriorityQueue()
    {
        super();
    }

    /**
     * Adds the given element to this PriorityQueue.
     *
     * @param object the element to be added to the priority queue
     * @param priority the integer priority of the element to be added
     */
}
```

LISTING 12.3 *continued*

```

public void addElement(T object, int priority)
{
    PrioritizedObject<T> obj = new PrioritizedObject<T>(object, priority);
    super.addElement(obj);
}

/**
 * Removes the next highest priority element from this priority
 * queue and returns a reference to it.
 *
 * @return a reference to the next highest priority element in this queue
 */
public T removeNext()
{
    PrioritizedObject<T> obj = (PrioritizedObject<T>)super.removeMin();
    return obj.getElement();
}
}

```

12.3 Implementing Heaps: With Links

All of our implementations of trees thus far have been illustrated using links. Thus it is natural to extend that discussion to a linked implementation of a heap.

Because of the requirement that we be able to traverse up the tree after an insertion, it is necessary for the nodes in a heap to store a pointer to their parent. Because our `BinaryTreeNode` class did not have a parent pointer, we start our linked implementation by creating a `HeapNode` class that extends our `BinaryTreeNode` class and adds a parent pointer. Listing 12.4 shows the `HeapNode` class.

KEY CONCEPT

Because of the requirement that we be able to traverse up the tree after an insertion, it is necessary for the nodes in a heap to store a pointer to their parent.

LISTING 12.4

```

package jsjf;

/**
 * HeapNode represents a binary tree node with a parent pointer for use
 * in heaps.
 *
 * @author Lewis and Chase

```

LISTING 12.4*continued*

```
* @version 4.0
*/
public class HeapNode<T> extends BinaryTreeNode<T>
{
    protected HeapNode<T> parent;

    /**
     * Creates a new heap node with the specified data.
     *
     * @param obj the data to be contained within the new heap node
     */
    public HeapNode(T obj)
    {
        super(obj);
        parent = null;
    }

    /**
     * Return the parent of this node.
     *
     * @return the parent of the node
     */
    public HeapNode<T> getParent()
    {
        return parent;
    }

    /**
     * Sets the element stored at this node.
     *
     * @param the element to be stored
     */
    public void setElement(T obj)
    {
        element = obj;
    }

    /**
     * Sets the parent of this node.
     *
     * @param node the parent of the node
     */
    public void setParent(HeapNode<T> node)
    {
        parent = node;
    }
}
```



```

/**
 * Adds the specified element to this heap in the appropriate
 * position according to its key value.
 *
 * @param obj the element to be added to the heap
 */
public void addElement(T obj)
{
    HeapNode<T> node = new HeapNode<T>(obj);
    if (root == null)
        root=node;
    else
    {
        HeapNode<T> nextParent = getNextParentAdd();
        if (nextParent.getLeft() == null)
            nextParent.setLeft(node);
        else
            nextParent.setRight(node);

        node.setParent(nextParent);
    }
    lastNode = node;
    modCount++;
    if (size() > 1)
        heapifyAdd();
}

```

The additional instance data for a linked implementation will consist of a single reference to a `HeapNode` called `lastNode` so that we can keep track of the last leaf in the heap.

The addElement Operation

The `addElement` method must accomplish three tasks: add the new node at the appropriate location, reorder the heap to maintain the ordering property, and then reset the `lastNode` pointer to point to the new last node.

This method also uses two private methods: `getNextParentAdd`, which returns a reference to the node that will be the parent of the node to be inserted, and `heapifyAdd`, which accomplishes any necessary reordering of the heap, starting with the new leaf and working up toward the root.

```

/**
 * Returns the node that will be the parent of the new node
 *
 * @return the node that will be the parent of the new node
 */
private HeapNode<T> getNextParentAdd()
{
    HeapNode<T> result = lastNode;

    while ((result != root) && (result.getParent().getLeft() != result))
        result = result.getParent();

    if (result != root)
        if (result.getParent().getRight() == null)
            result = result.getParent();
        else
        {
            result = (HeapNode<T>)result.getParent().getRight();
            while (result.getLeft() != null)
                result = (HeapNode<T>)result.getLeft();
        }
    else
        while (result.getLeft() != null)
            result = (HeapNode<T>)result.getLeft();

    return result;
}

```

```

/**
 * Reorders this heap after adding a node.
 */
private void heapifyAdd()
{
    T temp;
    HeapNode<T> next = lastNode;

    temp = next.getElement();

    while ((next != root) &&
        (((Comparable)temp).compareTo(next.getParent().getElement()) < 0))
    {
        next.setElement(next.getParent().getElement());
        next = next.parent;
    }
    next.setElement(temp);
}

```

In this linked implementation, the first step in the process of adding an element is to determine the parent of the node to be inserted. Because, in the worst case, this involves traversing from the bottom-right node of the heap up to the root and then down to the bottom-left node of the heap, this step has time complexity $2 \times \log n$. The next step is to insert the new node. Because it involves only simple assignment statements, this step has constant time complexity ($O(1)$). The last step is to reorder the path from the inserted leaf to the root, if necessary. This process involves at most $\log n$ comparisons, because that is the length of the path. Thus the `addElement` operation for the linked implementation has time complexity $2 \times \log n + 1 + \log n$ or $O(\log n)$.

Note that the `heapifyAdd` method does not perform a full swap of parent and child as it moves up the heap. Instead, it simply shifts parent elements down until a proper insertion point is found and then assigns the new value into that location. This does not actually improve the $O()$ of the algorithm, because it would be $O(\log n)$ even if we were performing full swaps. However, it does improve the efficiency, because it reduces the number of assignments performed at each level of the heap.

The `removeMin` Operation

The `removeMin` method must accomplish three tasks: replace the element stored in the root with the element stored in the last node, reorder the heap if necessary, and return the original root element. Like the `addElement` method, the `removeMin` method uses two additional methods: `getNewLastNode`, which returns a reference to the node that will be the new last node, and `heapifyRemove`, which accomplishes any necessary reordering of the tree starting from the root down. All three of these methods are shown below.

```
/**
 * Remove the element with the lowest value in this heap and
 * returns a reference to it. Throws an EmptyCollectionException
 * if the heap is empty.
 *
 * @return the element with the lowest value in this heap
 * @throws EmptyCollectionException if the heap is empty
 */
public T removeMin() throws EmptyCollectionException
{
    if (isEmpty())
        throw new EmptyCollectionException("LinkedHeap");

    T minElement = root.getElement();
```

```

if (size() == 1)
{
    root = null;
    lastNode = null;
}
else
{
    HeapNode<T> nextLast = getNewLastNode();
    if (lastNode.getParent().getLeft() == lastNode)
        lastNode.getParent().setLeft(null);
    else
        lastNode.getParent().setRight(null);

    ((HeapNode<T>)root).setElement(lastNode.getElement());
    lastNode = nextLast;
    heapifyRemove();
}

modCount++;

return minElement;
}

```

```

/**
 * Returns the node that will be the new last node after a remove.
 *
 * @return the node that will be the new last node after a remove
 */
private HeapNode<T> getNewLastNode()
{
    HeapNode<T> result = lastNode;

    while ((result != root) && (result.getParent().getLeft() == result))
        result = result.getParent();

    if (result != root)
        result = (HeapNode<T>)result.getParent().getLeft();

    while (result.getRight() != null)
        result = (HeapNode<T>)result.getRight();

    return result;
}

```

```

/**
 * Reorders this heap after removing the root element.
 */
private void heapifyRemove()
{
    T temp;
    HeapNode<T> node = (HeapNode<T>)root;
    HeapNode<T> left = (HeapNode<T>)node.getLeft();
    HeapNode<T> right = (HeapNode<T>)node.getRight();
    HeapNode<T> next;

    if ((left == null) && (right == null))
        next = null;
    else if (right == null)
        next = left;
    else if (((Comparable)left.getElement()).compareTo(right.
getElement()) < 0)
        next = left;
    else
        next = right;

    temp = node.getElement();
    while ((next != null) &&
        (((Comparable)next.getElement()).compareTo(temp) < 0))
    {
        node.setElement(next.getElement());
        node = next;
        left = (HeapNode<T>)node.getLeft();
        right = (HeapNode<T>)node.getRight();

        if ((left == null) && (right == null))
            next = null;
        else if (right == null)
            next = left;
        else if (((Comparable)left.getElement()).compareTo(right.
getElement()) < 0)
            next = left;
        else
            next = right;
    }
    node.setElement(temp);
}

```

The `removeMin` method for the linked implementation must remove the root element and replace it with the element from the last node. Because this process is accomplished with simple assignment statements, this step has time complexity 1. Next, this method must reorder the heap, if necessary, from the root down to a leaf. Because the maximum path length from the root to a leaf is $\log n$, this step has time complexity $\log n$. Finally, we must determine the new last node. Like the process for determining the next parent node for the `addElement` method, the worst case is that we must traverse from a leaf through the root and down to another leaf. Thus the time complexity of this step is $2 \cdot \log n$. The resulting time complexity of the `removeMin` operation is $2 \cdot \log n + \log n + 1$ or $O(\log n)$.

The `findMin` Operation

The `findMin` method simply returns a reference to the element stored at the root of the heap and therefore is $O(1)$.

12.4 Implementing Heaps: With Arrays

To this point, we have focused our discussion of the implementation of trees around linked structures. If you recall, however, in Chapter 10 we discussed a couple of different array implementation strategies for trees: the computational strategy and the simulated link strategy. An array implementation of a heap may provide a simpler alternative than our linked implementation. Many of the intricacies of the linked implementation are related to the need to traverse up and down the tree to determine the last leaf of the tree or to determine the parent of the next node to insert. Many of those difficulties do not exist in the array implementation, because we are able to determine the last node in the tree by looking at the last element stored in the array.

As we discussed in Chapter 10, a simple array implementation of a binary tree can be created using the notion that the root of the tree is in position 0, and that for each node n , n 's left child will be in position $2n + 1$ of the array and n 's right child will be in position $2(n + 1)$ of the array. Of course, the inverse is also true. For any node n other than the root, n 's parent is in position $(n - 1)/2$. Because of our ability to calculate the location of both parent and child, the array implementation (unlike the linked implementation) does not require the creation of a `HeapNode` class. The UML description of the array implementation of a heap is left as an exercise.

Just as the `LinkedHeap` class extends the `LinkedBinaryTree` class, the `ArrayHeap` class extends the `ArrayBinaryTree` class. The class header, attributes, and constructors for both classes are provided for context.

KEY CONCEPT

In an array implementation of a binary tree, the root of the tree is in position 0, and for each node n , n 's left child is in position $2n + 1$, and n 's right child is in position $2(n + 1)$.

```

package jsjf;

import java.util.*;
import jsjf.exceptions.*;

/**
 * ArrayBinaryTree implements the BinaryTreeADT interface using an array
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class ArrayBinaryTree<T> implements BinaryTreeADT<T>, Iterable<T>
{
    private static final int DEFAULT_CAPACITY = 50;

    protected int count;
    protected T[] tree;
    protected int modCount;

    /**
     * Creates an empty binary tree.
     */
    public ArrayBinaryTree()
    {
        count = 0;
        tree = (T[]) new Object[DEFAULT_CAPACITY];
    }

    /**
     * Creates a binary tree with the specified element as its root.
     *
     * @param element the element which will become the root of the new tree
     */
    public ArrayBinaryTree(T element)
    {
        count = 1;
        tree = (T[]) new Object[DEFAULT_CAPACITY];
        tree[0] = element;
    }
}

```

```

package jsjf;

import jsjf.exceptions.*;

/**
 * ArrayHeap provides an array implementation of a minheap.

```

```

*
* @author Lewis and Chase
* @version 4.0
*/
public class ArrayHeap<T> extends ArrayBinaryTree<T> implements HeapADT<T>
{
    /**
     * Creates an empty heap.
     */
    public ArrayHeap()
    {
        super();
    }
}

```

The addElement Operation

The `addElement` method for the array implementation must accomplish three tasks: add the new node at the appropriate location, reorder the heap to maintain the ordering property, and increment the count by one. Of course, as with all of our array implementations, the method must first check for available space and expand the capacity of the array if necessary. Like the linked implementation, the `addElement` operation of the array implementation uses a private method called `heapifyAdd` to reorder the heap if necessary.

```

/**
 * Adds the specified element to this heap in the appropriate
 * position according to its key value.
 *
 * @param obj the element to be added to the heap
 */
public void addElement(T obj)
{
    if (count == tree.length)
        expandCapacity();

    tree[count] = obj;
    count++;
    modCount++;

    if (count > 1)
        heapifyAdd();
}

```



```

/**
 * Reorders this heap to maintain the ordering property after
 * adding a node.
 * /
private void heapifyAdd()
{
    T temp;
    int next = count - 1;

    temp = tree[next];

    while ((next != 0) &&
        (((Comparable)temp).compareTo(tree[(next-1)/2]) < 0))
    {
        tree[next] = tree[(next-1)/2];
        next = (next-1)/2;
    }

    tree[next] = temp;
}

```

Unlike the linked implementation, the array implementation does not require the first step of determining the parent of the new node. However, both of the other steps are the same as those for the linked implementation. Thus the time complexity for the `addElement` operation for the array implementation is $1 + \log n$ or $O(\log n)$. Granted, the two implementations have the same $\text{Order}()$, but the array implementation is more efficient and more elegant.

KEY CONCEPT

The `addElement` operation for both the linked implementation and the array implementation is $O(\log n)$.

The `removeMin` Operation

The `removeMin` method must accomplish three tasks: replace the element stored in the root with the element stored in the last element, reorder the heap if necessary, and return the original root element. In the case of the array implementation, we know the last element of the heap is stored in position `count1` of the array. We then use a private method `heapifyRemove` to reorder the heap as necessary.

```
/**
 * Remove the element with the lowest value in this heap and
 * returns a reference to it. Throws an EmptyCollectionException if
 * the heap is empty.
 *
 * @return a reference to the element with the lowest value in this heap
 * @throws EmptyCollectionException if the heap is empty
 */
public T removeMin() throws EmptyCollectionException
{
    if (isEmpty())
        throw new EmptyCollectionException("ArrayHeap");

    T minElement = tree[0];
    tree[0] = tree[count-1];
    heapifyRemove();
    count--;
    modCount--;

    return minElement;
}
```

```
/**
 * Reorders this heap to maintain the ordering property
 * after the minimum element has been removed.
 */
private void heapifyRemove()
{
    T temp;
    int node = 0;
    int left = 1;
    int right = 2;
    int next;

    if ((tree[left] == null) && (tree[right] == null))
        next = count;
    else if (tree[right] == null)
        next = left;
    else if (((Comparable)tree[left]).compareTo(tree[right]) < 0)
        next = left;
    else
        next = right;
    temp = tree[node];
```

```

while ((next < count) &&
      (((Comparable)tree[next]).compareTo(temp) < 0))
{
    tree[node] = tree[next];
    node = next;
    left = 2 * node + 1;
    right = 2 * (node + 1);
    if ((tree[left] == null) && (tree[right] == null))
        next = count;
    else if (tree[right] == null)
        next = left;
    else if (((Comparable)tree[left]).compareTo(tree[right]) < 0)
        next = left;
    else
        next = right;
}
tree[node] = temp;
}

```

KEY CONCEPT

The `removeMin` operation for both the linked implementation and the array implementation is $O(\log n)$.

Like the `addElement` method, the array implementation of the `removeMin` operation looks just like the linked implementation, except that it does not have to determine the new last node. Thus the resulting time complexity is $n + 1$ or $O(\log n)$.

The `findMin` Operation

Like the linked implementation, the `findMin` method simply returns a reference to the element stored at the root of the heap or position 0 of the array and therefore is $O(1)$.

12.5 Using Heaps: Heap Sort

Now that we have examined an array implementation of a heap, let's consider another way we might use it. In Chapter 9, we introduced a variety of sorting techniques, some of which were sequential sorts (bubble sort, selection sort, and insertion sort) and some of which were logarithmic sorts (merge sort and quick sort). In that chapter, we also introduced a queue-based sort called a radix sort. Given the ordering property of a heap, it is natural to think of using a heap to sort a list of numbers. A brute force approach to a heap sort would be to add each of

the elements of the list to a heap and then remove them one at a time from the root. In the case of a minheap, the result will be the list in ascending order. In the case of a maxheap, the result will be the list in descending order. Because both the add operation and the remove operation are $O(\log n)$, it might be tempting to conclude that a heap sort is also $O(\log n)$. However, keep in mind that those operations are $O(\log n)$ to add or remove a single element in a list of n elements. Insertion into a heap is $O(\log n)$ for any given node and thus would be $O(n \log n)$ for n nodes. Removal is also $O(\log n)$ for a single node and thus $O(n \log n)$ for n nodes. With the heap sort algorithm, we are performing both operations, `addElement` and `removeMin`, n times, once for each of the elements in the list. Therefore, the resulting time complexity is $2 \times n \log n$ or $O(n \log n)$.

It is also possible to “build” a heap in place using the array to be sorted. Because we know the relative position of each parent and child in the heap, we can simply start with the first non-leaf node in the array, compare it to its children, and swap if necessary. We then work backward in the array until we reach the root. Because, at most, this will require us to make two comparisons for each non-leaf node, this approach is $O(n)$ to build the heap. However, with this approach, removing each element from the heap and maintaining the properties of the heap would still be $O(n \log n)$. Thus, even though this approach is slightly more efficient, roughly $2 \times n + n \log n$, it is still $O(n \log n)$. The implementation of this approach is left as an exercise. The `heapSort` method could be added to our class of sort methods described in Chapter 9. Listing 12.5 illustrates how it might be created as a standalone class.

KEY CONCEPT

The `heapSort` method consists of adding each of the elements of the list to a heap and then removing them one at a time.

KEY CONCEPT

Heap sort is $O(n \log n)$.



VideoNote

Demonstration of a heap sort on an array

LISTING 12.5

```
package jsjf;

/**
 * HeapSort sorts a given array of Comparable objects using a heap.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class HeapSort<T>
{
    /**
     * Sorts the specified array using a Heap
     *
     */
}
```

LISTING 12.5 *continued*

```
* @param data the data to be added to the heapsort
*/
public void HeapSort(T[] data)
{
    ArrayHeap<T> temp = new ArrayHeap<T>();

    // copy the array into a heap
    for (int i = 0; i < data.length; i++)
        temp.addElement(data[i]);

    // place the sorted elements back into the array
    int count = 0;
    while (!(temp.isEmpty()))
    {
        data[count] = temp.removeMin();
        count++;
    }
}
```

Summary of Key Concepts

- A minheap is a complete binary tree in which each node is less than or equal to both the left child and the right child.
- A minheap stores its smallest element at the root of the binary tree, and both children of the root of a minheap are also minheaps.
- The `addElement` method adds a given `Comparable` element to the appropriate location in the heap, maintaining both the completeness property and the ordering property of the heap.
- Because a heap is a complete tree, there is only one correct location for the insertion of a new node, and that is either the next open position from the left at level h or, if level h is full, the first position on the left at level $h + 1$.
- Typically, in heap implementations, we keep track of the position of the last node or, more precisely, the last leaf, in the tree.
- To maintain the completeness of the tree, there is only one valid element to replace the root, and that is the element stored in the last leaf in the tree.
- Even though it is not a queue at all, a minheap provides an efficient implementation of a priority queue.
- Because of the requirement that we be able to traverse up the tree after an insertion, it is necessary for the nodes in a heap to store a pointer to their parent.
- In an array implementation of a binary tree, the root of the tree is in position 0, and for each node n , n 's left child is in position $2n + 1$, and n 's right child is in position $2(n + 1)$.
- The `addElement` operation for both the linked implementation and the array implementation is $O(\log n)$.
- The `removeMin` operation for both the linked implementation and the array implementation is $O(\log n)$.
- The `heapSort` method consists of adding each of the elements of the list to a heap and then removing them one at a time.
- Heap sort is $O(n \log n)$.

Summary of Terms

heap A binary tree that is complete and is either a minheap or a maxheap.

minheap A binary tree with two added properties: It is a complete tree, and for each node, the node is less than or equal to both the left child and the right child.

maxheap A binary tree with two added properties: It is a complete tree, and for each node, the node is greater than or equal to both the left child and the right child.

complete binary tree A balanced binary tree in which all of the leaves at level h (the lowest level of the tree) are on the left side of the tree.

priority queue A collection that follows two ordering rules: Items with higher priority go first, and items with the same priority are ordered in accordance with the first in, first out principle.

Self-Review Questions

- SR 12.1 What is the difference between a heap (a minheap) and a binary search tree?
- SR 12.2 What is the difference between a minheap and a maxheap?
- SR 12.3 What does it mean for a binary tree to be complete?
- SR 12.4 Does a heap ever have to be rebalanced?
- SR 12.5 The `addElement` operation for the linked implementation must determine the parent of the next node to be inserted. Why?
- SR 12.6 Why does the `addElement` operation for the array implementation not have to determine the parent of the next node to be inserted?
- SR 12.7 The `removeMin` operation for both implementations replaces the element at the root with the element in the last leaf of the heap. Why is this the proper replacement?
- SR 12.8 What is the time complexity of the `addElement` operation?
- SR 12.9 What is the time complexity of the `removeMin` operation?
- SR 12.10 What is the time complexity of heap sort?

Exercises

- EX 12.1 Draw the heap that results from adding the following integers.

34 45 3 87 65 32 1 12 17

- EX 12.2 Starting with the tree resulting from Exercise 12.1, draw the heap that results from performing a `removeMin` operation.
- EX 12.3 Starting with an empty minheap, draw the heap after each of the following operations.

```
addElement(40);
addElement(25);
```

```
removeMin();
addElement(10);
removeMin();
addElement(5);
addElement(1);
removeMin();
addElement(45);
addElement(50);
```

- EX 12.4 Repeat Exercise 12.3, this time with a maxheap.
- EX 12.5 Draw the UML description for the `PriorityQueue` class described in this chapter.
- EX 12.6 Draw the UML description for the array implementation of heap described in this chapter.

Programming Projects

- PP 12.1 Implement a queue using a heap. Keep in mind that a queue is a first in, first out structure. Thus the comparison in the heap will have to be according to order entry into the queue.
- PP 12.2 Implement a stack using a heap. Keep in mind that a stack is a last in, first out structure. Thus the comparison in the heap will have to be according to order entry into the queue.
- PP 12.3 Implement a maxheap using an array implementation.
- PP 12.4 Implement a maxheap using a linked implementation.
- PP 12.5 As described in Section 12.5, it is possible to make the heap sort algorithm more efficient by writing a method that will build a heap in place, using the array to be sorted. Implement such a method, and rewrite the heap sort algorithm to make use of it.
- PP 12.6 Use a heap to implement a simulator for a process scheduling system. In this system, jobs will be read from a file consisting of the job id (a six-character string), the length of the job (an `int` representing seconds), and the priority of the job (an `int` where the higher the number, the higher the priority). Each job will also be assigned an arrival number (an `int` representing the order of its arrival). The simulation should output the job id, the priority, the length of the job, and the completion time (relative to a simulation start time of 0).
- PP 12.7 Create a birthday reminder system using a minheap such that the ordering on the heap is done each day according to days remaining until the individual's birthday. Keep in mind that when a birthday passes, the heap must be reordered.

- PP 12.8 Complete the implementation of an `ArrayHeap` including the `ArrayBinaryTree` class that the `ArrayHeap` extends.
- PP 12.9 Complete the implementation of the `LinkedHeap` class.

Answers to Self-Review Questions

- SRA 12.1 A binary search tree has the ordering property that the left child of any node is less than the node, and the node is less than or equal to its right child. A minheap is complete and has the ordering property that the node is less than both of its children.
- SRA 12.2 A minheap has the ordering property that the node is less than both of its children. A maxheap has the ordering property that the node is greater than both of its children.
- SRA 12.3 A binary tree is considered complete if it is balanced, which means that all of the leaves are at level h or $h - 1$, where h is $\log_2 n$ and n is the number of elements in the tree, and all of the leaves at level h are on the left side of the tree.
- SRA 12.4 No. By definition, a complete heap is balanced and the algorithms for `add` and `remove` maintain that balance.
- SRA 12.5 The `addElement` operation must determine the parent of the node to be inserted so that a child pointer of that node can be set to the new node.
- SRA 12.6 The `addElement` operation for the array implementation does not have to determine the parent of the new node, because the new element is inserted in position `count` of the array, and its parent is determined by position in the array.
- SRA 12.7 To maintain the completeness of the tree, the only valid replacement for the element at the root is the element at the last leaf. Then the heap must be reordered as necessary to maintain the ordering property.
- SRA 12.8 For both implementations, the `addElement` operation is $O(\log n)$. However, despite having the same order, the array implementation is somewhat more efficient because it does not have to determine the parent of the node to be inserted.
- SRA 12.9 For both implementations, the `removeMin` operation is $O(\log n)$. However, despite having the same order, the array implementation is somewhat more efficient because it does not have to determine the new last leaf.
- SRA 12.10 The heap sort algorithm is $O(n \log n)$.



Sets and Maps

13

This chapter introduces the Java concepts of sets and maps. We will explore these collections and compare and contrast them with our previous implementations. We will also introduce the concept of hashing.

CHAPTER OBJECTIVES

- Introduce the Java set and map collections.
- Explore the use of sets and maps to solve problems.
- Introduce the concept of hashing.
- Discuss how the Java API implements sets and maps.

13.1 Set and Map Collections

A *set* can be defined as a collection of elements with no duplicates. You should not assume that there is any particular positional relationship among the elements of a set.

For the most part, set collections in Java can be thought of in the mathematical sense of a set. They represent a collection of unique elements that can be used to determine the relationship of an element to the set. That is, the primary purpose of a set is to determine whether a particular element is a member of the set.

KEY CONCEPT

A set is a unique collection of objects generally used to determine whether a particular element is a member of the set.

Of course, other collections (such as a list) have the ability to test for containment. However, if such tests are an important part of a program, you should consider using sets. The implementation of a set is explicitly designed to be efficient when searching for an element.

A *map* is a collection that establishes a relationship between keys and values, providing an efficient way to retrieve a value given its key. The keys of a map must be unique, and each key can map to only one value. For example, you could use a unique membership id (a `String`) to retrieve the information about that member of a club (a `Member` object).

It doesn't have to be a one-to-one mapping, however. Multiple keys could map to the same object. For example, in a situation where information about a topic is being looked up, multiple keywords can map to the same topic entry. The key "gardening" and the key "mulch beds" and the key "flowers" could all map to the `Topic` object describing gardening, for instance.

KEY CONCEPT

A map is a collection of objects that can be retrieved using a unique key.

The keys of a map don't have to be character strings, although they often are. Both the keys and the values of a map can be any type of object.

Like that of a set, a map's implementation is specifically designed to provide efficient lookup. In fact, as we'll see in more detail later in this chapter, the set and map classes defined in the Java API are implemented using similar underlying techniques.



VideoNote

A comparison of sets and maps

13.2 Sets and Maps in the Java API

The Java API defines interfaces called `Set` and `Map` to define the public interaction available for these types of collections. In the remainder of this chapter, we'll explore the interfaces for these classes, use them to solve some problems, and then discuss the underlying implementation strategies.

The operations of the `Set` interface are listed in Figure 13.1. Like other collections, a set has operations that allow the user to add elements, remove elements, and check whether a particular element is in the collection. Some operations, such as `isEmpty` and `size`, are common to nearly all collections as well. The `contains` and `containsAll` methods perform the key operations of determining whether the set contains particular elements.

Method Summary	
boolean	<code>add(E e)</code> Adds the specified element to this set if it is not already present (optional operation).
boolean	<code>addAll(Collection<? extends > c)</code> Adds all of the elements in the specified collection to this set if they are not already present (optional operation).
void	<code>clear()</code> Removes all of the elements from this set (optional operation).
boolean	<code>contains(Object o)</code> Returns true if this set contains the specified element.
boolean	<code>containsAll(Collection<?> c)</code> Returns true if this set contains all of the elements of the specified collection.
boolean	<code>equals(Object o)</code> Compares the specified object with this set for equality.
int	<code>hashCode()</code> Returns the hash code value for this set.
boolean	<code>isEmpty()</code> Returns true if this set contains no elements.
Iterator	<code>iterator()</code> Returns an iterator over the elements in this set.
boolean	<code>remove(Object o)</code> Removes the specified element from this set if it is present (optional operation).
boolean	<code>removeAll(Collection<?> c)</code> Removes from this set all of its elements that are contained in the specified collection (optional operation).
boolean	<code>retainAll(Collection<?> c)</code> Retains only the elements in this set that are contained in the specified collection (optional operation).
int	<code>size()</code> Returns the number of elements in this set (its cardinality).

Method Summary (continued)	
Object []	toArray() Returns an array containing all of the elements in this set.
<T> T []	toArray(T[] a) Returns an array containing all of the elements in this set; the run-time type of the returned array is that of the specified array.

FIGURE 13.1 The operations in the `Set` interface

Like most collections, the elements of a set are defined using a generic type parameter (`E` in this case). The only objects that can be added to a set are those that are type compatible with the generic type established when a set object is instantiated.

Figure 13.2 illustrates the operations in the `Map` interface. Elements are added to a map using the `put` operation, which accepts both the key object and its corresponding value as parameters. A particular element is retrieved from the map using the `get` operation, which accepts the key object as a parameter.

Method Summary	
void	clear() Removes all of the mappings from this map (optional operation).
boolean	containsKey(Object key) Returns true if this map contains a mapping for the specified key.
boolean	containsValue(Object value) Returns true if this map maps one or more keys to the specified value.
Set<Map.Entry<K, V>>	entrySet() Returns a Set view of the mappings contained in this map.
boolean	equals(Object o) Compares the specified object with this map for equality.
V	get(Object key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
int	hashCode() Returns the hash code value for this map.
boolean	isEmpty() Returns true if this map contains no key-value mappings.
Set<K>	keySet() Returns a Set view of the keys contained in this map.

V	<code>put(K key, V value)</code> Associates the specified value with the specified key in this map (optional operation).
void	<code>putAll(Map<? extends K, ? extends V> m)</code> Copies all of the mappings from the specified map to this map (optional operation).
V	<code>remove(Object key)</code> Removes the mapping for a key from this map if it is present (optional operation).
int	<code>size()</code> Returns the number of key-value mappings in this map.
<code>Collection<V></code>	<code>values()</code> Returns a <code>Collection</code> view of the values contained in this map.

FIGURE 13.2 The operations in the `Map` interface

The `Map` interface has two generic type parameters, one for the key (`K`) and one for the value (`V`). When a class implementing a `Map` is instantiated, both types are established for that particular map, and all subsequent operations work in terms of those types.

The Java API provides two implementation classes for each interface: `TreeSet` and `HashSet` are two implementations of the `Set` interface; `TreeMap` and `HashMap` are two implementations of the `Map` interface. As the names imply, the classes use two different underlying implementation techniques: trees and hashing.

Next we'll explore some examples that use these classes to solve some problems, and then we will discuss each implementation strategy in more detail.

13.3 Using Sets: Domain Blocker

One of the primary purposes of a set is to test for membership in the set. Let's consider an example that tests web site domains against a list of blocked domains. We could use a simple list of blocked domains, but when we use a `TreeSet` instead, each check for a particular domain is accomplished in $\log n$ steps instead of in n steps.

Suppose that the following list of blocked domains is held in a text input file called `blockedDomains.txt`:

```
dontgothere.com
ohno.org
badstuff.com
badstuff.org
```

```

badstuff.net
whatintheworld.com
notinthislifetime.org
letsnot.com
eeewwwww.com

```

Listing 13.1 illustrates the `DomainBlocker` class, which keeps track of the blocked domains and checks candidates against them as needed. The constructor for this class reads the file and sets up a `TreeSet` containing all of the blocked domains. The `isBlocked` method that determines whether a given domain is in the set.

In this example, the set of blocked domains is represented by a `TreeSet` object. The domains themselves are simply character strings.

LISTING 13.1

```

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
import java.util.TreeSet;

/**
 * A URL domain blocker.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class DomainBlocker
{
    private TreeSet<String> blockedSet;

    /**
     * Sets up the domain blocker by reading in the blocked domain names from
     * a file and storing them in a TreeSet.
     * @throws FileNotFoundException
     */
    public DomainBlocker() throws FileNotFoundException
    {
        blockedSet = new TreeSet<String>();

        File inputFile = new File("blockedDomains.txt");
        Scanner scan = new Scanner(inputFile);

        while (scan.hasNextLine())
        {
            blockedSet.add(scan.nextLine());
        }
    }
}

```

LISTING 13.1*continued*

```
/**
 * Checks to see if the specified domain has been blocked.
 *
 * @param domain the domain to be checked
 * @return true if the domain is blocked and false otherwise
 */
public boolean domainIsBlocked(String domain)
{
    return blockedSet.contains(domain);
}
}
```

In Listing 13.2 we see the `DomainChecker` class. As the driver for this example, this class creates an instance of the `DomainBlocker` class and then allows the user to enter domains interactively to check to see whether they are blocked.

LISTING 13.2

```
import java.io.FileNotFoundException;
import java.util.Scanner;

/**
 * Domain checking driver.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class DomainChecker
{
    /**
     * Repeatedly reads a domain interactively from the user and checks to
     * see if that domain has been blocked.
     */
    public static void main(String[] args) throws FileNotFoundException
    {
        DomainBlocker blocker = new DomainBlocker();
        Scanner scan = new Scanner(System.in);
    }
}
```


LISTING 13.2 *continued*

```

String domain;

do
{
    System.out.print("Enter a domain (DONE to quit): ");
    domain = scan.nextLine();

    if (!domain.equalsIgnoreCase("DONE"))
    {
        if (blocker.domainIsBlocked(domain))
            System.out.println("That domain is blocked.");
        else
            System.out.println("That domain is fine.");
    }
} while (!domain.equalsIgnoreCase("DONE"));
}
}

```

13.4 Using Maps: Product Sales

Let's look at an example using the `TreeMap` class. What if we were trying to keep track of product sales? Suppose that each time a product is sold, its product code is entered into a sales file. Here's a sample of how that information might appear in a file. Note that there are duplicates in the list.

```

HR588
DX555
EW231
TT232
TJ991
HR588
TT232
GB637
BV693
CB329
NP466
CB329
EW231
BV693
DX555
GB637
VA838

```

Our system would need to read the sales file and update the product information for each entry. We could organize our collection by product code but then keep that separate from the actual product information. Listing 13.3 shows the `Product` class, and Listing 13.4 shows the `ProductSales` class.

In our previous collections, when we wanted to retrieve or find an object in the collection, we would have had to instantiate an object of the same type and with the same critical information in order to look for it. One of the advantages of using a `Map` is that we no longer have to do that. In this example, our key is a `String`. Therefore, we were able to search the `Map` using a `String` rather than having to create a dummy `Product` object.

LISTING 13.3

```
/**
 * Represents a product for sale.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class Product implements Comparable<Product>
{
    private String productCode;
    private int sales;

    /**
     * Creates the product with the specified code.
     *
     * @param productCode a unique code for this product
     */
    public Product(String productCode)
    {
        this.productCode = productCode;
        this.sales = 0;
    }

    /**
     * Returns the product code for this product.
     *
     * @return the product code
     */
    public String getProductCode()
    {
        return productCode;
    }
}
```

LISTING 13.3 *continued*

```

/**
 * Increments the sales of this product.
 */
public void incrementSales()
{
    sales++;
}

/**
 * Compares this product to the specified product based on the product
 * code.
 *
 * @param other the other product
 * @return an integer code result
 */
public int compareTo(Product obj)
{
    return productCode.compareTo(obj.getProductCode());
}

/**
 * Returns a string representation of this product.
 *
 * @return a string representation of the product
 */
public String toString()
{
    return productCode + "\t(" + sales + ")";
}
}

```

LISTING 13.4

```

import java.io.File;
import java.io.IOException;
import java.util.Scanner;
import java.util.TreeMap;

```

LISTING 13.4*continued*

```
/**
 * Demonstrates the use of a TreeMap to store a sorted group of Product
 * objects.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class ProductSales
{
    /**
     * Processes product sales data and prints a summary sorted by
     * product code.
     */
    public static void main(String[] args) throws IOException
    {
        TreeMap<String, Product> sales = new TreeMap<String, Product>();

        Scanner scan = new Scanner(new File("salesData.txt"));

        String code;
        Product product;
        while (scan.hasNext())
        {
            code = scan.nextLine();
            product = sales.get(code);
            if (product == null)
                sales.put(code, new Product(code));
            else
                product.incrementSales();
        }

        System.out.println("Products sold this period:");
        for (Product prod : sales.values())
            System.out.println(prod);
    }
}
```

OUTPUT

```
Products sold this period:
BR742 (67)
BV693 (69)
```

LISTING 13.4 *continued*

```
CB329 (67)
DX555 (67)
DX699 (72)
EW231 (66)
GB637 (56)
HR588 (66)
LF845 (69)
LH933 (59)
NP466 (67)
OB311 (50)
TJ991 (79)
TT232 (74)
UI294 (75)
VA838 (60)
WL023 (76)
WL310 (81)
WL812 (65)
YG904 (78)
```

In the `main` method, a `while` loop is used to read all values from the input file. For each product code, we attempt to get the corresponding `Product` object from the map using the product code as the key. If the result is null, then no sales of that product have been recorded yet, and a new `Product` object is created and added to the map. If it was successfully retrieved from the map, the `incrementSales` method is called.

The output of the program lists only unique product codes found in the input file, followed by the number of sales in parentheses. Note that the output shown in Listing 13.4 is based on a much larger input file than the sample shown earlier in the chapter.

The output is accomplished by a `for-each` loop in the `main` method, which retrieves a list of all `Product` objects stored in the map using a call to the `values` method. The values are returned in order by product code, because that's how `Product` objects rank themselves using the `compareTo` method of `Product`.

13.5 Using Maps: User Management

Suppose we wanted to create a system to manage users. Our system could maintain a map of users and allow searches for particular users based on a user id. Listing 13.5 illustrates our `User` class, representing an individual user, and Listing 13.6 represents the `Users` class, representing the collection of users.

LISTING 13.5

```
/**
 * Represents a user with a userid.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class User
{
    private String userId;
    private String firstName;
    private String lastName;

    /**
     * Sets up this user with the specified information.
     *
     * @param userId a user identification string
     * @param firstName the user's first name
     * @param lastName the user's last name
     */
    public User(String userId, String firstName, String lastName)
    {
        this.userId = userId;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    /**
     * Returns the user id of this user.
     *
     * @return the user id of the user
     */
    public String getUserId()
    {
        return userId;
    }

    /**
     * Returns a string representation of this user.
     *
     * @return a string representation of the user
     */
    public String toString()
    {
        return userId + ":\t" + lastName + ", " + firstName;
    }
}
```

LISTING 13.6

```
import java.util.HashMap;
import java.util.Set;

/**
 * Stores and manages a map of users.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class Users
{
    private HashMap<String, User> userMap;

    /**
     * Creates a user map to track users.
     */
    public Users()
    {
        userMap = new HashMap<String, User>();
    }

    /**
     * Adds a new user to the user map.
     *
     * @param user the user to add
     */
    public void addUser(User user)
    {
        userMap.put(user.getUserId(), user);
    }

    /**
     * Retrieves and returns the specified user.
     *
     * @param userId the user id of the target user
     * @return the target user, or null if not found
     */
    public User getUser(String userId)
    {
        return userMap.get(userId);
    }

    /**
     * Returns a set of all user ids.
     */
}
```

LISTING 13.6*continued*

```
    *
    * @return a set of all user ids in the map
    */
    public Set<String> getUserIds()
    {
        return userMap.keySet();
    }
}
```

In the `Users` class, individual `User` objects are stored in a `HashMap` object, using a user id (string) as a key. The `addUser` and `getUser` methods simply store and retrieve the `User` objects as needed. The `getUserIds` method returns a `Set` of user ids using a call to the `keySet` method of the map.

Listing 13.7 shows the `UserManagement` class that contains the main method of our program. The method creates and adds several users, allows the user to search for them interactively, and then prints all of the users in the collection.

LISTING 13.7

```
import java.io.IOException;
import java.util.Scanner;

/**
 * Demonstrates the use of a map to manage a set of objects.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class UserManagement
{
    /**
     * Creates and populates a group of users. Then prompts for interactive
     * searches, and finally prints all users.
     */
    public static void main(String[] args) throws IOException
    {
```


LISTING 13.7 *continued*

```

Users users = new Users();
users.addUser(new User("fziffle", "Fred", "Ziffle"));
users.addUser(new User("geoman57", "Marco", "Kane"));
users.addUser(new User("rover322", "Kathy", "Shear"));
users.addUser(new User("appleseed", "Sam", "Geary"));
users.addUser(new User("mon2016", "Monica", "Blankenship"));

Scanner scan = new Scanner(System.in);
String uid;
User user;

do
{
    System.out.print("Enter User Id (DONE to quit): ");
    uid = scan.nextLine();
    if (!uid.equalsIgnoreCase("DONE"))
    {
        user = users.getUser(uid);
        if (user == null)
            System.out.println("User not found.");
        else
            System.out.println(user);
    }
} while (!uid.equalsIgnoreCase("DONE"));

// print all users
System.out.println("\nAll Users:\n");
for (String userId : users.getUserIds())
    System.out.println(users.getUser(userId));
}

```

OUTPUT

```
Enter User Id (DONE to quit): DONE
```

```

All Users:
geoman57: Kane, Marco
appleseed: Geary, Sam
rover322: Shear, Kathy
fziffle: Ziffle, Fred
mon2016: Blankenship, Monica

```

13.6 Implementing Sets and Maps Using Trees

As the names imply, the `TreeSet` and `TreeMap` classes use an underlying tree structure to hold the elements in the set or map. In previous chapters, we explored trees as collections in their own right, first as general trees in Chapter 10, then as binary search trees in Chapter 11. As we discussed in those chapters, the Java API does not treat trees as collections, but only as a means to implement other collections.

The tree used to implement `TreeSet` and `TreeMap` is a red-black implementation of a balanced binary search tree.

Recall the discussion of red-black trees in Chapter 11. They guarantee that the search tree remains balanced as elements are added and removed, which in turn results in nearly all of the basic operations being executed with $O(\log n)$ efficiency. These trees use the so-called natural ordering of elements, based on the `Comparable` interface, unless an explicit `Comparator` object is provided.

Furthermore, it turns out that the `TreeSet` and `TreeMap` classes in the API don't have their own unique implementations of the underlying tree. The `TreeSet` class is built upon a backing instance of a `TreeMap`.

KEY CONCEPT

The Java API treats trees as implementing data structures rather than as collections.

KEY CONCEPT

Both `TreeSet` and `TreeMap` use a red-black balanced binary search tree.

KEY CONCEPT

In the Java API, `TreeSet` is built using an underlying `TreeMap`.

13.7 Implementing Sets and Maps Using Hashing

The `HashSet` and `HashMap` classes are implemented using an underlying technique called *hashing* as the means by which elements are stored and retrieved. First we will discuss hashing in general; then we will consider how it is used to implement sets and maps.

In all of our discussions of the implementations of collections, we have proceeded with one of two assumptions about the order of elements in a collection:

- Order is determined by the order in which elements are added to and/or removed from our collection, as in the case of stacks, queues, unordered lists, and indexed lists.
- Order is determined by comparing the values of the elements (or some key component of the elements) to be stored in the collection, as in the case of ordered lists and binary search trees.

With hashing, however, the order—and, more specifically, the location of an item within the collection—is determined by some function of the value of the

element to be stored, or some function of a key value of the element to be stored. In hashing, elements are stored in a *hash table*, with their location in the table determined by a *hashing function*. Each location in the table may be referred to as a *cell* or a *bucket*. A complete discussion of hashing functions is included in Appendix E, but we'll discuss the basics here.

Consider a simple example where we create an array that will hold 26 elements. Wishing to store names in our array, we create a hashing function that equates each name to the position in the array associated with the first letter of the name (for example, a first letter of A would be mapped to position 0 of the array, a first letter of D would be mapped to position 3 of the array, and so on). Figure 13.3 illustrates this scenario after several names have been added.

KEY CONCEPT

The situation in which two elements or keys map to the same location in a hash table is called a collision.

Note that, unlike our earlier implementations of collections, using a hashing approach results in the access time to a particular element being independent of the number of elements in the table. This means that all of the operations on an element of a hash table should be $O(1)$. This is the result of no longer having to do comparisons to find a particular element or to locate the appropriate position for a given element. Using hashing, we simply calculate where a particular element should be.

KEY CONCEPT

A hashing function that maps each element to a unique position in the hash table is said to be a perfect hashing function.

However, this efficiency is fully realized only if each element maps to a unique position in the table. Consider our example from Figure 13.3. What will happen if we attempt to store the name “Ann” and the name “Andrew”? This situation, where two elements or keys map to the same location in a hash table, is called a *collision*.

A hashing function that maps each element to a unique position in the hash table is said to be a *perfect hashing function*. Although it is possible in some situations to develop a perfect hashing function, a hashing function that does a good job of distributing the elements among the table positions will still result in constant time ($O(1)$) access to elements in the table and an improvement over our earlier algorithms that were either $O(n)$ in the case of our linear approaches or $O(\log n)$ in the case of search trees.

KEY CONCEPT

In hashing, elements are stored in a hash table, and their location in the table is determined by a hashing function.

A complete discussion of hashing is included in Appendix E. Now, let's consider how the Java API uses hashing to create a set implementation.

Just as the `TreeSet` class was built upon a backing `TreeMap` instance, the `HashSet` class is built upon a backing instance of the `HashMap` class. The `HashSet` class provides constant time ($O(1)$) access for the basic operations as long as the hash function does a reasonable job of distributing elements in the hash table. The two parameters to the constructor that affect the efficiency of the hash function are the *initial capacity* and *load factor*.

The initial capacity determines the initial size of the hash table. The load factor determines how full the table is allowed to be before its size is increased. The default for the initial capacity is 16, and the default for the load factor is 0.75. With these defaults, the table size would be doubled once 12 elements had been added.

When an element is added to a `HashSet`, the object's `hashCode` method is called to produce an integer hash code for the object. If the `hashCode` method has not been overridden, then the `hashCode` method of the `java.lang.Object` class is used. Whether it uses this method or an overridden version, the requirements of the `hashCode` method as stated in the Java API are the same:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided that no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

Ann
Doug
Elizabeth
Hal
Mary
Tim
Walter
Young

FIGURE 13.3 A simple hashing example

Summary of Key Concepts

- A set is a unique collection of objects generally used to determine whether a particular element is a member of the set.
- A map is a collection of objects that can be retrieved using a unique key.
- The Java API treats trees as implementing data structures rather than as collections.
- In the Java API, `TreeSet` is built using a backing `TreeMap`.
- Both `TreeSet` and `TreeMap` use a red-black balanced binary search tree.
- In hashing, elements are stored in a hash table, and their location in the table is determined by a hashing function.
- The situation in which two elements or keys map to the same location in the table is called a collision.
- A hashing function that maps each element to a unique position in the table is said to be a perfect hashing function.

Summary of Terms

set A unique set of objects generally used to determine whether a particular element is a member of the set.

map A collection of objects that can be retrieved using a unique key.

collision The situation in which two elements or keys map to the same location in a hash table.

hashing A technique by which elements are stored in, and retrieved from, a hash table, and their location in the table is determined by a hashing function.

hash table A table where elements are stored in the hashing technique.

hashing function In the hashing technique, the function that determines where elements are stored in a hash table.

cell A location in a hash table.

bucket A location in a hash table.

perfect hashing function A hashing function that maps each element to a unique position in a hash table.

initial capacity The parameter that determines the initial size of a hash table.

load factor The parameter that determines how full a hash table is allowed to be before its size is increased.

Self-Review Questions

- SR 13.1 What is a set?
- SR 13.2 What is a map?
- SR 13.3 How are sets and maps implemented in the Java API?
- SR 13.4 What is the relationship between a `TreeSet` and a `TreeMap`?
- SR 13.5 What is the relationship between a `HashSet` and a `HashMap`?
- SR 13.6 How does a hash table differ from the other implementation strategies we have discussed?
- SR 13.7 What is the potential advantage of a hash table over other implementation strategies?
- SR 13.8 Define the terms *collision* and *perfect hashing function*.

Exercises

- EX 13.1 Define the concept of a set. List additional operations that might be considered for a set.
- EX 13.2 The `TreeSet` class is built upon a backing instance of the `TreeMap` class. Discuss the advantages and disadvantages of this strategy for reuse.
- EX 13.3 Given the nature of a set, one could implement the `Set` interface using any one of a variety of other collections or data structures. Describe how you might implement the `Set` interface using a `LinkedList`. Discuss the advantages and disadvantages of this approach.
- EX 13.4 A bag is a very similar construct to a set except that duplicates are allowed in a bag. What changes would have to be made to extend a `TreeSet` to create an implementation of a bag?
- EX 13.5 Draw a UML diagram showing the relationships among the classes involved in the Product Sales example from this chapter.
- EX 13.6 Draw a UML diagram showing the relationships among the classes in the User Management example from this chapter.
- EX 13.7 Describe two hashing functions that might be appropriate for a data set organized by name (e.g. last name, first name, middle initial).
- EX 13.8 Explain when it might be preferable to use a map instead of a set.

Programming Projects

- PP 13.1 Create an array based implementation of a set called `ArraySet<T>` that implements the `Set` interface.
- PP 13.2 Create a linked implementation of a set call `LinkedSet<T>` that implements the `Set` interface.
- PP 13.3 Create a tree-based implementation of a `TreeBag<T>` class. Remember, the difference is that a bag allows duplicates.
- PP 13.4 Create a hash table based implementation of a `HashBag<T>` class. Remember, the difference is that a bag allows duplicates.
- PP 13.5 Extend the `TreeSet` class to create a class called `AlgebraicTreeSet`. In addition to the methods of the `Set` interface, this class will provide the basic algebraic set operations of union, intersection, and difference.
- PP 13.6 Create the `AlgebraicTreeSet` class of PP13.5 by extending the `HashSet` class.
- PP 13.7 Building upon PP 13.1, create an array implementation of a map.
- PP 13.8 Building upon PP 13.2, create a linked implementation of a map.
- PP 13.9 Using a `TreeMap` develop a rolodex application to keep track of `Contact` objects as described in Chapter 9.
- PP 13.10 Using a `HashMap` develop a new implementation of the `ProgramofStudy` application from Chapter 6.

Answers to Self-Review Questions

- SRA 13.1 A set is a unique set of objects generally used to determine whether a particular element is a member of the set.
- SRA 13.2 A map is a collection of objects that can be retrieved using a unique key.
- SRA 13.3 Sets and maps are implemented in the Java API both with Red/Black Binary Trees (`TreeSet` and `TreeMap`) and with hash-tables (`HashSet` and `HashMap`).
- SRA 13.4 A `TreeSet` is implemented using a backing instance of a `TreeMap`.
- SRA 13.5 A `HashSet` is implemented using a backing instance of a `HashMap`.

- SRA 13.6 Using a hash table, the location of an element in the table is determined using a hashing function. In this way, each element in the hash table can be accessed in equal, $O(1)$, time.
- SRA 13.7 Given the $O(1)$ access time to each element in a hash table, assuming a good hashing function, then a hash table has the potential to be more efficient than some of our other strategies. For example, a binary search tree might require $O(\log n)$ time to access a given element as opposed to the $O(1)$ access time of a hash table.
- SRA 13.8 A *collision* occurs in a hash-table when two or more different elements are hashed to the same location in the table. A *perfect hashing function* is one that does not produce any collisions.

This page is intentionally left blank.



Multi-Way Search Trees

14

When we first introduced the concept of efficiency of algorithms, we said that we were interested in issues such as processing time and memory. In this chapter, we explore multi-way trees that were specifically designed with a concern for the use of space and the effect that a particular use of space could have on the total processing time for an algorithm.

CHAPTER OBJECTIVES

- Examine 2-3 and 2-4 trees.
- Introduce the generic concept of a B-tree.
- Examine some specialized implementations of B-trees.

14.1 Combining Tree Concepts

In Chapter 10, we established the difference between a general tree, which has a varying number of children per node, and a binary tree, which has at most two children per node. Then, in Chapter 11, we discussed the concept of a search tree, which has a specific ordering relationship among the elements in the nodes to allow efficient searching for a target value. In particular, we focused on binary search trees. Now we can combine these concepts and extend them further.

KEY CONCEPT

A multi-way search tree can have more than two children per node and can store more than one element in each node.

In a *multi-way search tree*, each node might have more than two child nodes, and, because it is a search tree, there is a specific ordering relationship among the elements. Furthermore, a single node in a multi-way search tree may store more than one element.

This chapter examines three specific forms of a multi-way search tree:

- 2-3 trees
- 2-4 trees
- B-trees

14.2 2-3 Trees

A *2-3 tree* is a multi-way search tree in which each node has two children (referred to as a *2-node*) or three children (referred to as a *3-node*). A 2-node contains one element, and, as in a binary search tree, the left subtree contains elements that are less than that element, and the right subtree contains elements that are greater than or equal to that element. However, unlike the case in a binary search tree, a 2-node can have either no children or two children—it cannot have just one child.

A 3-node contains two elements, one designated as the smaller element and one designated as the larger element. A 3-node has either no children or three children. If a 3-node has children, then the left subtree contains elements that are less than the smaller element, and the right subtree contains elements that are greater than or equal to the larger element. The middle subtree contains elements that are greater than or equal to the smaller element and less than the larger element.

KEY CONCEPT

A 2-3 tree contains nodes that contain either one or two elements and have zero, two, or three children.

All of the leaves of a 2-3 tree are on the same level. Figure 14.1 illustrates a valid 2-3 tree.

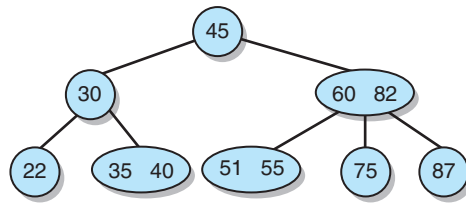


FIGURE 14.1 A 2-3 tree

Inserting Elements into a 2-3 Tree

Similar to a binary search tree, all insertions into a 2-3 tree occur at the leaves of the tree. That is, the tree is searched to determine where the new element will go; then it is inserted. Unlike a binary tree, however, the process of inserting an element into a 2-3 tree can have a ripple effect on the structure of the rest of the tree.

Inserting an element into a 2-3 tree has three cases. The first, and simplest, case is that the tree is empty. In this case, a new node is created containing the new element, and this node is designated as the root of the tree.

The second case occurs when we want to insert a new element at a leaf that is a 2-node. That is, we traverse the tree to the appropriate leaf (which may also be the root) and find that the leaf is a 2-node (containing only one element). In this case, the new element is added to the 2-node, making it a 3-node. Note that the new element may be less than or greater than the existing element. Figure 14.2 illustrates this case by inserting the value 27 into the tree shown in Figure 14.1. The leaf node containing 22 is a 2-node, so 27 is inserted into that node, making it a 3-node. Note that neither the number of nodes in the tree nor the height of the tree changed because of this insertion.

The third insertion situation occurs when we want to insert a new element at a leaf that is a 3-node (containing two elements). In this case, because the 3-node

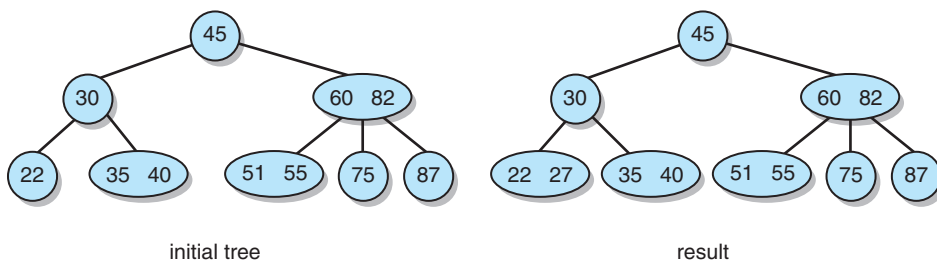


FIGURE 14.2 Inserting 27

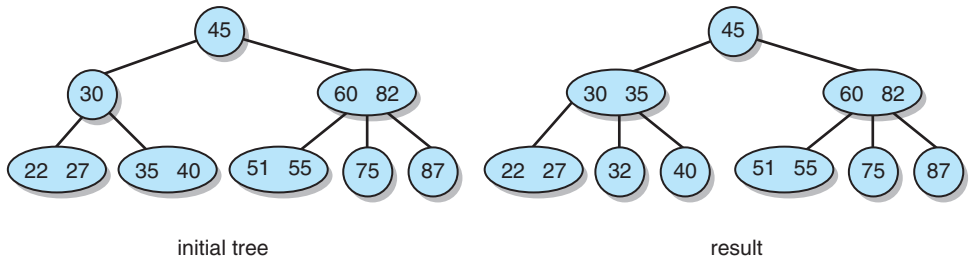


FIGURE 14.3 Inserting 32

cannot hold any more elements, it is split, and the middle element is moved up a level in the tree. The middle element that moves up a level can be either of the two elements that already existed in the 3-node, or it can be the new element being inserted. It depends on the relationship among those three elements.

Figure 14.3 shows the result of inserting the element 32 into the tree shown in Figure 14.2. Searching the tree, we reach the 3-node that contains the elements 35 and 40. That node is split, and the middle element (35) is moved up to join its parent node. Thus the internal node that contains 30 becomes a 3-node that contains both 30 and 35. Note that the act of splitting a 3-node results in two 2-nodes at the leaf level. In this example, we are left with one 2-node that contains 32 and another 2-node that contains 40.

Now consider the situation in which we must split a 3-node whose parent is already a 3-node. The middle element that is promoted causes the parent to split, moving an element up yet another level in the tree. Figure 14.4 shows the effect of inserting the element 57 into the tree shown in Figure 14.3. Searching the tree, we reach the 3-node leaf that contains 51 and 55. This node is split, causing the middle element 55 to move up a level. But that node is already a 3-node, containing the values 60 and 82, so we split that node as well, promoting the element 60,

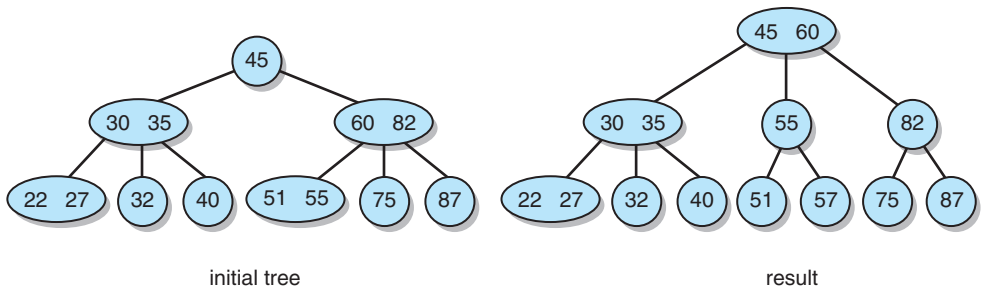


FIGURE 14.4 Inserting 57

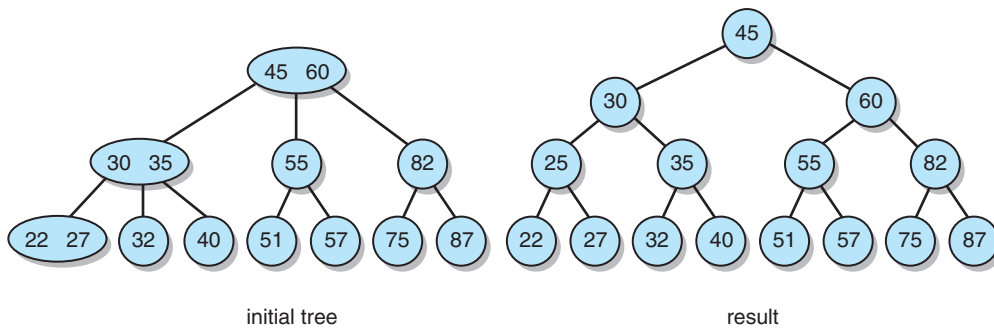


FIGURE 14.5 Inserting 25

which joins the 2-node containing 45 at the root. Therefore, inserting an element into a 2-3 tree can cause a ripple effect that changes several nodes in the tree.

If this effect propagates all the way to the root of the entire tree, a new 2-node root is created. For example, inserting the element 25 into the tree shown in Figure 14.4 results in the tree depicted in Figure 14.5. The 3-node containing 22 and 27 is split, promoting 25. This causes the 3-node containing 30 and 35 to split, promoting 30. This causes the 3-node containing 45 and 60 (which happens to be the root of the entire tree) to split, creating a new 2-node root that contains 45.

Note that when the root of the tree splits, the height of the tree increases by one. The insertion strategy for a 2-3 tree keeps all of the leaves at the same level.

KEY CONCEPT

If the propagation effect of a 2-3 tree insertion causes the root to split, the tree increases in height.

Removing Elements from a 2-3 Tree

Removal of elements from a 2-3 tree also has three cases. The first case is that the element to be removed is in a leaf that is a 3-node. In this case, removal is simply a matter of removing the element from the node. Figure 14.6 illustrates this process by removing the element 51 from the tree we began with in Figure 14.1. Note that the properties of a 2-3 tree are maintained.

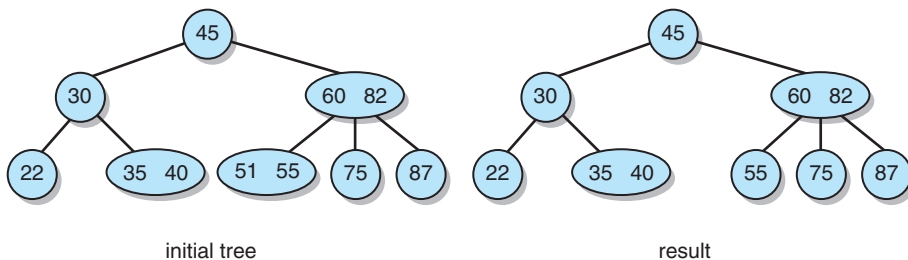


FIGURE 14.6 Removal from a 2-3 tree (case 1)

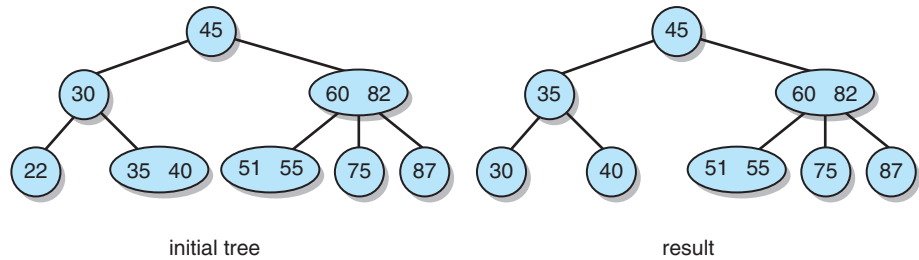


FIGURE 14.7 Removal from a 2-3 tree (case 2.1)



VideoNote

Inserting elements into, and removing elements from, a 2-3 tree

The second case is that the element to be removed is in a leaf that is a 2-node. This condition is called *underflow* and creates a situation in which we must rotate the tree and/or reduce the tree's height in order to maintain the properties of the 2-3 tree. This situation can be broken down into four subordinate cases that we will refer to as cases 2.1, 2.2, 2.3, and 2.4. Figure 14.7 illustrates case 2.1 and shows what happens if we remove the element 22 from our initial tree shown in Figure 14.1. In this case, because the parent node has a right child that is a 3-node, we can maintain the properties of a 2-3 tree by rotating the smaller element of the 3-node around the parent. The same process will work if the element being removed from a 2-node leaf is the right child and the left child is a 3-node.

What happens if we now remove the element 30 from the resulting tree in Figure 14.7? We can no longer maintain the properties of a 2-3 tree through a local rotation. Keep in mind that a node in a 2-3 tree cannot have just one child. Because the leftmost child of the right child of the root is a 3-node, we can rotate the smaller element of that node around the root to maintain the properties of a 2-3 tree. This process is illustrated in Figure 14.8 and represents case 2.2. Notice that the element 51 moves to the root, the element 45 becomes the larger element in a 3-node leaf, and then the smaller element of that leaf is rotated around its parent. Once element 51 was moved to the root and element 45 was moved to a 3-node leaf, we were back in the same situation as case 2.1.

Given the resulting 2-3 tree in Figure 14.8, what happens if we now remove element 55? None of the leaves of this tree is a 3-node. Thus, rotation from a leaf, even from a distance, is no longer an option. However, because the parent node is a 3-node, all that is required to maintain the properties of a 2-3 node is to change this 3-node to a 2-node by rotating the smaller element (60) into what will now be the left child of the node. Figure 14.9 illustrates case 2.3.

If we then remove element 60 (using case 1), the resulting tree contains nothing but 2-nodes. Now, if we remove another element, perhaps element 45, rotation is no longer an option. We must instead reduce the height of the tree in order to maintain the properties of a 2-3 tree. This is case 2.4. To accomplish this, we simply

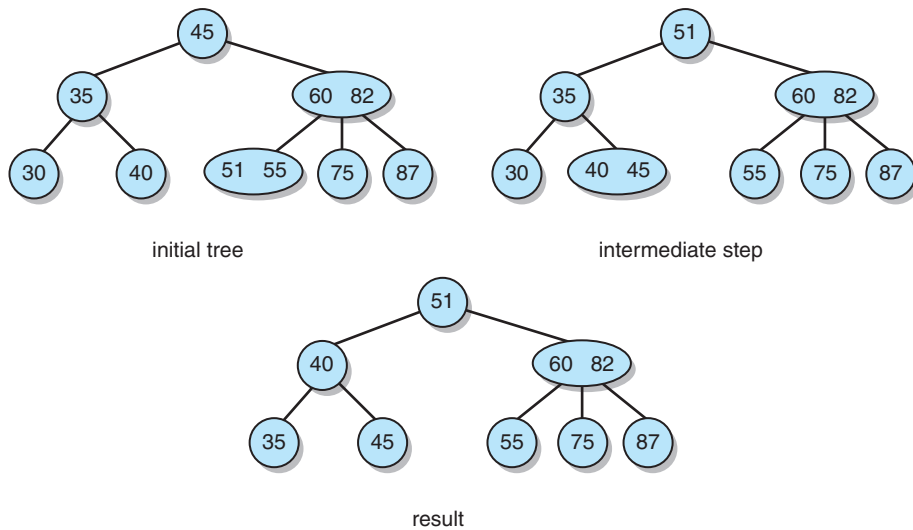


FIGURE 14.8 Removal from a 2-3 tree (case 2.2)

combine each of the leaves with its parent and siblings in order. If any of these combinations contains more than two elements, we split it into two 2-nodes and promote or propagate the middle element. Figure 14.10 illustrates this process for reducing the height of the tree.

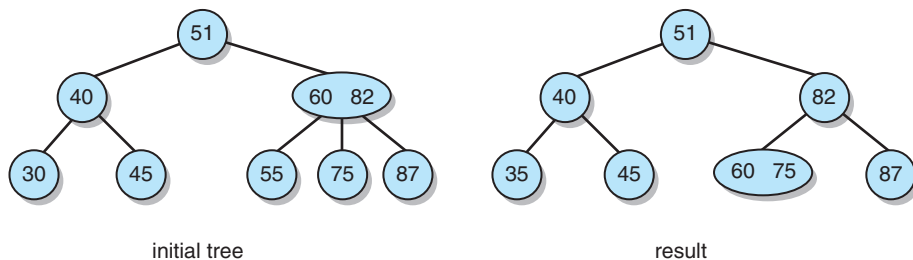


FIGURE 14.9 Removal from a 2-3 tree (case 2.3)

The third case is that the element to be removed is in an internal node. Just as we did with binary search trees, we can simply replace the element to be removed with its inorder successor. In a 2-3 tree, the inorder successor of an internal element will always be a leaf, which, if it is a 2-node, will bring us back to our first case, and if it is a 3-node, requires no further action. Figure 14.11 illustrates these possibilities by removing the element 30 from our original tree from Figure 14.1 and then by removing the element 60 from the resulting tree.

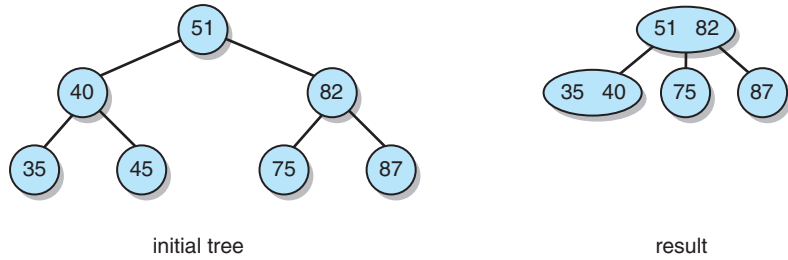


FIGURE 14.10 Removal from a 2-3 tree (case 2.4)

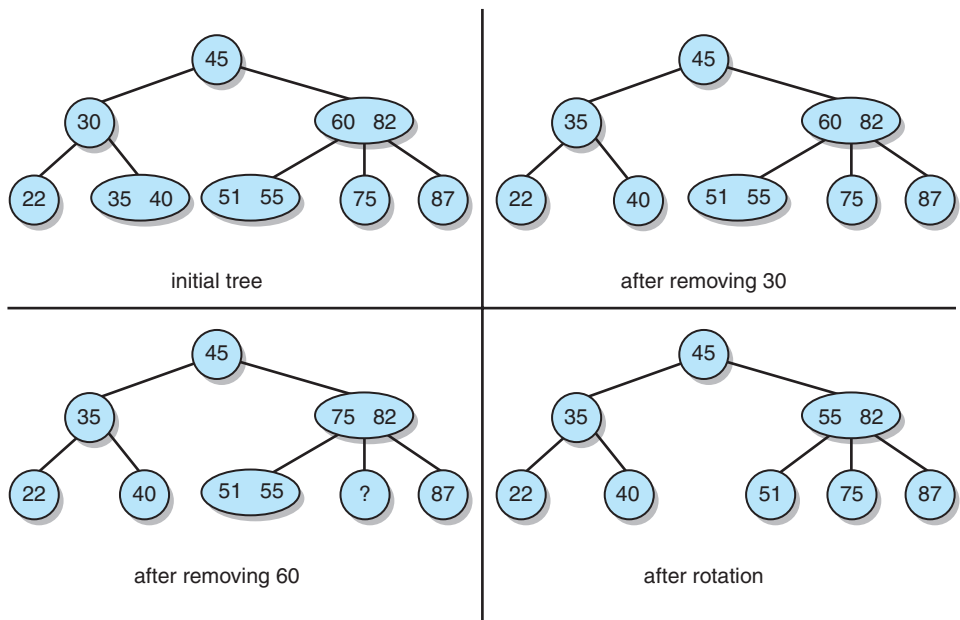


FIGURE 14.11 Removal from a 2-3 tree (case 3)

14.3 2-4 Trees

A *2-4 tree* is similar to a 2-3 tree, adding the characteristic that a node can contain three elements. Expanding on the same principles as a 2-3 tree, a *4-node* contains three elements and has either no children or four children. The same ordering property applies: The left child will be less than the leftmost element of a node, which will be less than or equal to the second child of the node, which will be less than the second element of the node, which will be less than or equal to the third

child of the node, which will be less than the third element of the node, which will be less than or equal to the fourth child of the node.

The same cases for insertion and removal of elements apply, with 2-nodes and 3-nodes behaving similarly on insertion and 3-nodes and 4-nodes behaving similarly on removal. Figure 14.12 illustrates a series of insertions into a 2-4 tree. Figure 14.13 illustrates a series of removals from a 2-4 tree.

KEY CONCEPT

A 2-4 tree expands on the concept of a 2-3 tree to include the use of 4-nodes.

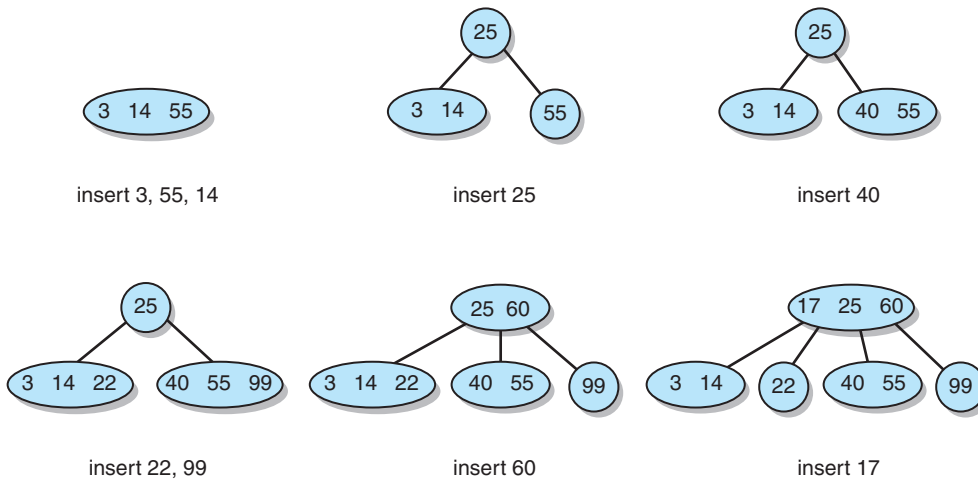


FIGURE 14.12 Insertions into a 2-4 tree

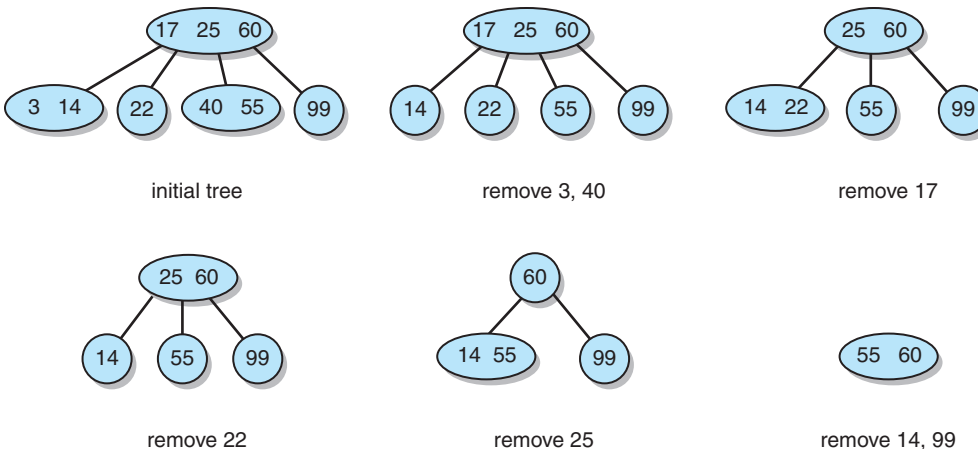


FIGURE 14.13 Removals from a 2-4 tree

14.4 B-Trees

Both 2-3 and 2-4 trees are examples of a larger class of multi-way search trees called *B-trees*. We refer to the maximum number of children of each node as the *order* of the B-tree. Thus 2-3 trees are B-trees of order 3, and 2-4 trees are B-trees of order 4.

KEY CONCEPT

A B-tree extends the concept of 2-3 and 2-4 trees so that nodes can have an arbitrary maximum number of elements.

B-trees of order m have the following properties:

- The root has at least two subtrees unless it is a leaf.
- Each non-root internal node n holds $k-1$ elements and k children, where $\lceil m/2 \rceil \leq k \leq m$.
- Each leaf n holds $k-1$ elements, where $\lceil m/2 \rceil \leq k \leq m$.
- All leaves are on the same level.

Figure 14.14 illustrates a B-tree of order 6.

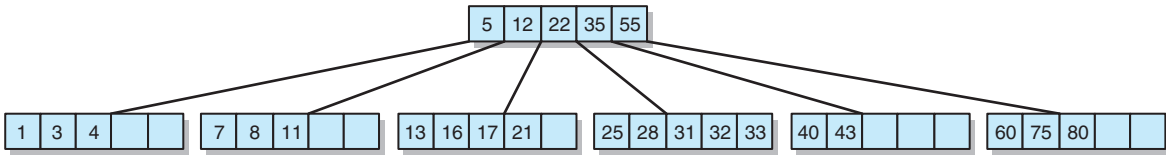


FIGURE 14.14 A B-tree of order 6

The reasoning behind the creation and use of B-trees is an interesting study in the effects of algorithm and data structure design. To understand this reasoning, we must understand the context of most all of the collections we have discussed thus far. Our assumption has always been that we were dealing with a collection in primary memory. However, what if the data set that we are manipulating is too large for primary memory? In that case, our data structure would be paged in and out of memory from a disk or some other secondary storage device. An interesting thing happens to time complexity once a secondary storage device is involved. No longer is the time to access an element of the collection simply a function of how many comparisons are needed to find the element. Now we must also consider the access time of the secondary storage device and how many separate accesses we will make to that device.

KEY CONCEPT

Access to secondary storage is very slow relative to access to primary storage, which is motivation to use structures such as B-trees.

In the case of a disk, this access time consists of seek time (the time it takes to position the read-write head over the appropriate track on the disk), rotational delay (the time it takes to spin the disk to the correct sector), and the transfer time (the time it takes to transfer a block of memory from the disk into primary memory).

Adding this “physical” complexity to the access time for a collection can be very costly. Access to secondary storage devices is very slow relative to access to primary storage.

Given this added time complexity, it makes sense to develop a structure that minimizes the number of times the secondary storage device must be accessed. A B-tree can be just such a structure. B-trees are typically tuned so that the size of a node is the same as the size of a block on secondary storage. In this way, we get the maximum amount of data for each disk access. Because B-trees can have many more elements per node than a binary tree, they are much flatter structures than binary trees. This reduces the number of nodes and/or blocks that must be accessed, thus improving performance.

We have already demonstrated the processes of insertion and removal of elements for 2-3 and 2-4 trees, both of which are B-trees. The process for any B-tree of order m is similar. Let’s now briefly examine some interesting variations of B-trees that were designed to solve specific problems.

B*-Trees

One of the potential problems with a B-tree is that even though we are attempting to minimize access to secondary storage, we have actually created a data structure that may be half empty. To minimize this problem, B*-trees were developed. *B*-trees* have all of the same properties as B-trees except that, instead of each node having k children where $\lceil m/2 \rceil \leq k \leq m$, in a B*-tree each node has k children where $\lceil (2m-1)/3 \rceil \leq k \leq m$. This means that each non-root node is at least two-thirds full.

This is accomplished by delaying splitting of nodes by rebalancing across siblings. Once siblings are full, instead of splitting one node into two, creating two half-full nodes, we split two nodes into three, creating three nodes that are two-thirds full.

B+-Trees

Another potential problem with B-trees is sequential access. As with any tree, we can use an inorder traversal to look at the elements of the tree sequentially. However, this means that we are no longer taking advantage of the blocking structure of secondary storage. In fact, we have made it much worse, because now we will access each block containing an internal node many separate times as we pass through it during the traversal.

B+-trees provide a solution to this problem. In a B-tree, each element appears only once in the tree, regardless of whether it appears in an internal node or in a

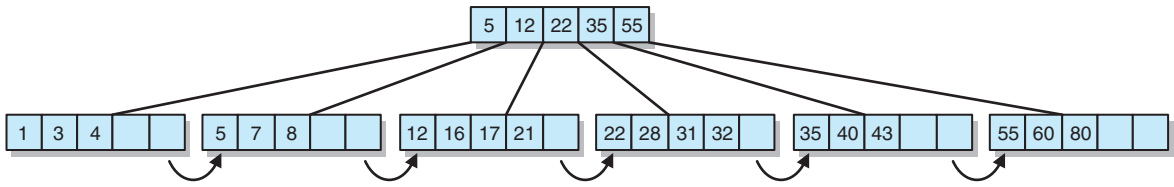


FIGURE 14.15 A B⁺-tree of order 6

leaf. In a B⁺-tree, each element appears in a leaf, regardless of whether or not it appears in an internal node. Elements appearing in an internal node will be listed again as the inorder successor (which is a leaf) of their position in the internal node. Additionally, each leaf node will maintain a pointer to the following leaf node. In this way, a B⁺-tree provides indexed access through the B-tree structure and sequential access through a linked list of leaves. Figure 14.15 illustrates this strategy.

Analysis of B-Trees

With balanced binary search trees, we were able to say that searching for an element in the tree was $O(\log_2 n)$. This is because, at worst, we had to search a single path from the root to a leaf in the tree and, at worst, the length of that path would be $\log_2 n$. Analysis of B-trees is similar. At worst, searching a B-tree, we will have to search a single path from the root to a leaf and, at worst, that path length will be $\log_m n$, where m is the order of the B-tree and n is the number of elements in the tree. However, finding the appropriate node is only part of the search. The other part of the search is finding the appropriate path from each node and then finding the target element in a given node. Because there are up to $m-1$ elements per node, it may take up to $m-1$ comparisons per node to find the appropriate path and/or to find the appropriate element. Thus, the analysis of a search of a B-tree yields $O((m-1)\log_m n)$. Because m is a constant for any given implementation, we can say that searching a B-tree is $O(\log n)$.

The analysis of insertion into and deletion from a B-tree is similar and is left as an exercise.

14.5 Implementation Strategies for B-Trees

We have already discussed insertion of elements into B-trees, removal of elements from B-trees, and the balancing mechanisms necessary to maintain the properties of a B-tree. What remains is to discuss strategies for storing B-trees. Keep in mind

that the B-tree structure was developed specifically to address the issue of a collection that must move in and out of primary memory from secondary storage. If we attempt to use object reference variables to create a linked implementation, we are actually storing a primary memory address for an object. Once that object is moved back to secondary storage, that address is no longer valid. Therefore, if interaction with secondary memory is part of your motivation to use a B-tree, then an array implementation may be a better solution.

A solution is to think of each node as a pair of arrays. The first array would be an array of $m-1$ elements, and the second array would be an array of m children. Next, if we think of the tree itself as one large array of nodes, then the elements stored in the array of children in each node would simply be integer indexes into this array of nodes.

In primary memory, this strategy works because when we use an array, as long as we know the index position of the element within the array, it does not matter to us where the array is loaded in primary memory. For secondary memory, this same strategy works because, given that each node is of fixed length, the address in memory of any given node is given by:

The base address of the file + (index of the node -1) \times length of a node

The array implementations of 2-3, 2-4, and larger B-trees are left as a programming project.

KEY CONCEPT

Arrays may provide a better solution both within a B-tree node and for collecting B-tree nodes, because they are effective in both primary memory and secondary storage.

Summary of Key Concepts

- A multi-way search tree can have more than two children per node and can store more than one element in each node.
- A 2-3 tree contains nodes that contain either one or two elements and have zero, two, or three children.
- Inserting an element into a 2-3 tree can have a ripple effect up the tree.
- If the propagation effect of a 2-3 tree insertion causes the root to split, the tree increases in height.
- A 2-4 tree expands on the concept of a 2-3 tree to include the use of 4-nodes.
- A B-tree extends the concept of 2-3 and 2-4 trees so that nodes can have an arbitrary maximum number of elements.
- Access to secondary storage is very slow relative to access to primary storage, which is motivation to use structures such as B-trees.
- Arrays may provide a better solution both within a B-tree node and for collecting B-tree nodes, because they are effective in both primary memory and secondary storage.

Summary of Terms

multi-way search tree A search tree where each node might have more than two child nodes and there is a specific ordering relationship among the elements.

2-3 tree A multi-way search tree in which each node has two children (referred to as a 2-node) or three children (referred to as a 3-node).

2-node A 2-node contains one element and, as in a binary search tree, the left subtree contains elements that are less than that element, and the right subtree contains elements that are greater than or equal to that element.

3-node A 3-node contains two elements, one designated as the smaller element and one designated as the larger element. A 3-node has either no children or three children. If a 3-node has children, the left subtree contains elements that are less than the smaller element and the right subtree contains elements that are greater than or equal to the larger element. The middle subtree contains elements that are greater than or equal to the smaller element and less than the larger element.

underflow A situation in which we must rotate the tree and/or reduce the tree's height in order to maintain the properties of the 2-3 tree.

2-4 tree A 2-4 tree is similar to a 2-3 tree, adding the characteristic that a node can contain three elements.

4-node A 4-node contains three elements and has either no children or four children.

B-tree A B-tree extends the concept of 2-3 and 2-4 trees so that nodes can have an arbitrary maximum number of elements.

B*-trees B*-trees have all of the same properties as B-trees except that, instead of each node having k children where $\lceil m/2 \rceil \leq k \leq m$, in a B*-tree each node has k children where $\lceil (2m-1)/3 \rceil \leq k \leq m$.

B⁺-trees In a B⁺-tree, each element appears in a leaf, regardless of whether or not it appears in an internal node. Elements appearing in an internal node will be listed again as the inorder successor (which is a leaf) of their position in the internal node. Additionally, each leaf node will maintain a pointer to the following leaf node. In this way, a B⁺-tree provides indexed access through the B-tree structure and sequential access through a linked list of leaves.

Self-Review Questions

- SR 14.1 Describe the nodes in a 2-3 tree.
- SR 14.2 When does a node in a 2-3 tree split?
- SR 14.3 How can splitting a node in a 2-3 tree affect the rest of the tree?
- SR 14.4 Describe the process of deleting an element from a 2-3 tree.
- SR 14.5 Describe the nodes in a 2-4 tree.
- SR 14.6 How do insertions and deletions in a 2-4 tree compare to insertions and deletions in a 2-3 tree?
- SR 14.7 When is rotation no longer an option for rebalancing a 2-3 tree after a deletion?

Exercises

- EX 14.1 Draw the 2-3 tree that results from adding the following elements into an initially empty tree:

34 45 3 87 65 32 1 12 17

EX 14.2 Using the resulting tree from Exercise 14.1, draw the resulting tree after removing each of the following elements:

3 87 12 17 45

EX 14.3 Repeat Exercise 14.1 using a 2-4 tree.

EX 14.4 Repeat Exercise 14.2 using the resulting 2-4 tree from Exercise 14.3.

EX 14.5 Draw the B-tree of order 8 that results from adding the following elements into an initially empty tree:

34 45 3 87 65 32 1 12 17 33 55 23 67 15 39 11 19 47

EX 14.6 Draw the B-tree that results from removing the following from the resulting tree from Exercise 14.5:

1 12 17 33 55 23 19 47

EX 14.7 Describe the complexity (order) of insertion into a B-tree.

EX 14.8 Describe the complexity (order) of deletion from a B-tree.

Programming Projects

PP 14.1 Create an implementation of a 2-3 tree using the array strategy discussed in Section 14.5.

PP 14.2 Create an implementation of a 2-3 tree using a linked strategy.

PP 14.3 Create an implementation of a 2-4 tree using the array strategy discussed in Section 14.5.

PP 14.4 Create an implementation of a 2-4 tree using a linked strategy.

PP 14.5 Create an implementation of a B-tree of order 7 using the array strategy discussed in Section 14.5.

PP 14.6 Create an implementation of a B⁺-tree of order 9 using the array strategy discussed in Section 14.5.

PP 14.7 Create an implementation of a B^{*}-tree of order 11 using the array strategy discussed in Section 14.5.

PP 14.8 Implement a graphical system to manage employees using an employee id, employee name, and years of service. The system should use a B-tree of order 7 to store employees, and it must provide the ability to add and remove employees. After each operation, your system must update a sorted list of employees sorted by name on the screen.

Answers to Self-Review Questions

- SRA 14.1 A 2-3 tree node can have either one element or two and can have no children, two children, or three children. If it has one element, then it is a 2-node and has either no children or two children. If it has two elements, then it is a 3-node and has either no children or three children.
- SRA 14.2 A 2-3 tree node splits when it has three elements. The smallest element becomes a 2-node, the largest element becomes a 2-node, and the middle element is promoted or propagated to the parent node.
- SRA 14.3 If the split and resulting propagation force the root node to split, then splitting the node will increase the height of the tree.
- SRA 14.4 Deletion from a 2-3 tree falls into one of three cases. Case 1, deletion of an element from a 3-node leaf, means simply removing the element and has no impact on the rest of the tree. Case 2, deletion of an element from a 2-node leaf, results in one of four cases. Case 2.1, deletion of an element from a 2-node that has a 3-node sibling, is resolved by rotating either the inorder predecessor or the inorder successor of the parent, depending upon whether the 3-node is a left child or a right child, around the parent. Case 2.2, deletion of an element from a 2-node when there is a 3-node leaf elsewhere in the tree, is resolved by rotating an element out of that 3-node and propagating that rotation until a sibling of the node being deleted becomes a 3-node; then this case becomes case 2.1. Case 2.3, deletion of a 2-node where there is a 3-node internal node, can be resolved through rotation as well. Case 2.4, deletion of a 2-node when there are no 3-nodes in the tree, is resolved by reducing the height of the tree.
- SRA 14.5 Nodes in a 2-4 tree are exactly like those in a 2-3 tree, except that 2-4 trees also allow 4-nodes, or nodes containing three elements and having four children.
- SRA 14.6 Insertions and deletions in a 2-4 tree are exactly like those in a 2-3 tree, except that splits occur when there are four elements, instead of three as in a 2-3 tree.
- SRA 14.7 If all of the nodes in a 2-3 tree are 2-nodes, then rotation is not an option for rebalancing.

References

- Bayer, R. “Symmetric Binary B-trees: Data Structure and Maintenance Algorithms.” *Acta Informatica* (1972): 290–306.
- Comer, D. “The Ubiquitous B-Tree.” *Computing Surveys* 11(1979): 121–137.
- Wedeking, H. “On the Selection of Access Paths in a Data Base System.” In *Data Base Management*, edited by J. W. Klimbie and K. L. Koffeman, pp. 385–397. Amsterdam: North-Holland, 1974.



Graphs

15

In Chapter 10, we introduced the concept of a tree, a non-linear structure defined by the concept that each node in the tree, other than the root node, has exactly one parent. If we were to violate that premise and allow each node in the tree to be connected to a variety of other nodes with no notion of parent or child, the result would be the concept of a graph, which we explore in this chapter. Graphs and graph theory make up entire subdisciplines of both mathematics and computer science. In this chapter, we introduce the basic concepts of graphs and their implementation.

CHAPTER OBJECTIVES

- Define *undirected graphs*.
- Define *directed graphs*.
- Define *weighted graphs or networks*.
- Explore common graph algorithms.

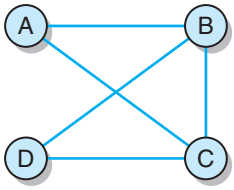


FIGURE 15.1 An example of an undirected graph

15.1 Undirected Graphs

Like trees, a graph is made up of nodes and the connections between those nodes. In graph terminology, we refer to the nodes as *vertices* and to the connections among them as *edges*. Vertices are typically identified by a name or a label. For example, we might label vertices A, B, C, and D. Edges are referred to by pairing the vertices that they connect. For example, we might have an edge (A, B), which means there is an edge from vertex A to vertex B.

An *undirected graph* is a graph where the pairings that represent the edges are unordered. Thus, listing an edge as (A, B) means that there is a connection between A and B that can be traversed in either direction. In an undirected graph, listing an edge as (A, B) means exactly the same thing as listing the edge as (B, A). Figure 15.1 illustrates the following undirected graph:

KEY CONCEPT

An undirected graph is a graph where the pairings that represent the edges are unordered.

KEY CONCEPT

Two vertices in a graph are adjacent if there is an edge connecting them.

KEY CONCEPT

An undirected graph is considered complete if it has the maximum number of edges connecting vertices.

Vertices: A, B, C, D

Edges: (A, B), (A, C), (B, C), (B, D), (C, D)

Two vertices in a graph are *adjacent* if there is an edge connecting them. For example, in the graph of Figure 15.1, vertices A and B are adjacent, and vertices A and D are not. Adjacent vertices are sometimes referred to as *neighbors*. An edge of a graph that connects a vertex to itself is called a *self-loop* or a *sling* and is represented by listing the vertex twice. For example, listing an edge (A, A) would mean that there is a sling connecting A to itself.

An undirected graph is considered *complete* if it has the maximum number of edges connecting vertices. For the first vertex, it requires $(n-1)$ edges to connect it to the other vertices. For the second vertex, it requires only $(n-2)$ edges because it is already connected to the first vertex. For the third vertex, it requires $(n-3)$ edges. This sequence continues until the final vertex requires no additional edges because all the other vertices have already been connected to it. Remember from Chapter 2 that the summation from 1 to n is

$$\sum_1^n i = n(n+1)/2$$

Thus, in this case, because we are summing only from 1 to $(n-1)$, the resulting summation is

$$\sum_1^{n-1} i = n(n-1)/2$$

This means that for any undirected graph with n vertices, it would require $n(n-1)/2$ edges to make the graph complete. This, of course, assumes that none of those edges is a sling.

A *path* is a sequence of edges that connects two vertices in a graph. For example, in our graph from Figure 15.1, A, B, D is a path from A to D. Notice that each sequential pair, (A, B) and then (B, D), is an edge. A path in an undirected graph is bi-directional. For example, A, B, D is the path from A to D, but because the edges are undirected, the inverse, D, B, A, is also the path from D to A. The *length* of a path is the number of edges in the path (or the number of vertices $- 1$). So for our previous example, the path length is 2. Notice that this definition of path length is identical to the definition that we used in discussing trees. In fact, trees are a special case of graphs.

An undirected graph is considered *connected* if for any two vertices in the graph, there is a path between them. Our graph from Figure 15.1 is connected. The same graph with a minor modification is not connected, as illustrated in Figure 15.2.

Vertices: A, B, C, D

Edges: (A, B), (A, C), (B, C)

A *cycle* is a path in which the first and last vertices are the same, and none of the edges is repeated. In Figure 15.2, we would say that the path A, B, C, A is a cycle. A graph that has no cycles is called *acyclic*. Earlier we mentioned the relationship between graphs and trees. Now that we have introduced these definitions, we can formalize that relationship. An undirected tree is a connected, acyclic, undirected graph with one element designated as the root.

KEY CONCEPT

A path is a sequence of edges that connects two vertices in a graph.

KEY CONCEPT

A cycle is a path in which the first and last vertices are the same, and none of the edges is repeated.

KEY CONCEPT

An undirected tree is a connected, acyclic, undirected graph with one element designated as the root.

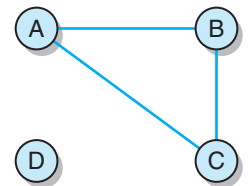


FIGURE 15.2 An example of an undirected graph that is not connected

15.2 Directed Graphs

A *directed graph*, sometimes referred to as a *digraph*, is a graph where the edges are ordered pairs of vertices. This means that the edges (A, B) and (B, A) are separate, directional edges in a directed graph. In our previous example, we had the following description for an undirected graph:

Vertices: A, B, C, D

Edges: (A, B), (A, C), (B, C), (B, D), (C, D)

Figure 15.3 shows what happens if we interpret this earlier description as a directed graph. We represent each of the edges now with the direction of traversal specified by the ordering of the vertices.

KEY CONCEPT

A directed graph, sometimes referred to as a digraph, is a graph where the edges are ordered pairs of vertices.

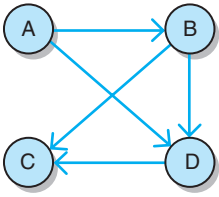


FIGURE 15.3 An example of a directed graph

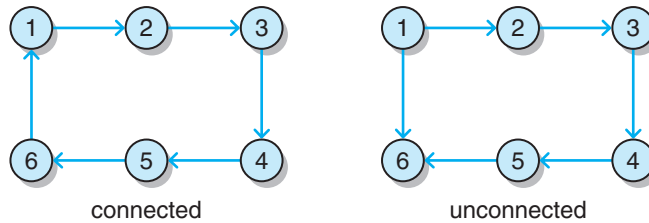


FIGURE 15.4 Examples of a connected directed graph and an unconnected directed graph

For example, the edge (A, B) allows traversal from A to B but not traversal in the other direction.

KEY CONCEPT

A path in a directed graph is a sequence of directed edges that connects two vertices in the graph.

Our previous definitions change slightly for directed graphs. For example, a path in a directed graph is a sequence of directed edges that connects two vertices in a graph. In our undirected graph, we listed the path A, B, D as the path from A to D , and that is still true in our directed interpretation of the graph description. However, paths in a directed graph are not bi-directional, so the inverse is no longer true: D, B, A is not a valid path from D to A , unless we add directional edges (D, B) and (B, A) .

Our definition for a connected directed graph sounds the same as it did for undirected graphs. A directed graph is connected if, for any two vertices in the graph, there is a path between them. However, keep in mind that our definition of a path is different. Look at the two graphs shown in Figure 15.4. The first one is connected. The second one is not connected, because there is no path from any other vertex to vertex 1.

If a directed graph has no cycles, it is possible to arrange the vertices such that vertex A precedes vertex B if an edge exists from A to B . The order of vertices resulting from this arrangement is called *topological order* and is very useful for examples such as course prerequisites.

As we discussed earlier, trees are graphs. In fact, most of our previous much work with trees actually focused on directed trees. A directed tree is a directed graph that has an element designated as the root and has the following properties:

- There are no connections from other vertices to the root.
- Every non-root element has exactly one connection to it.
- There is a path from the root to every other vertex.

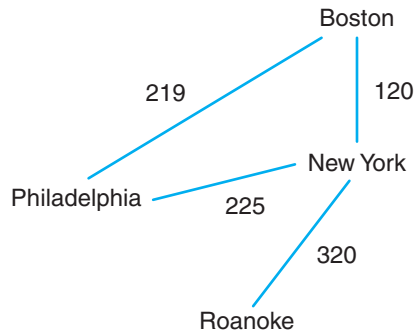


FIGURE 15.5 An undirected network

15.3 Networks

A *network*, or *weighted graph*, is a graph with weights or costs associated with each edge. Figure 15.5 shows an undirected network of the connections and the airfares between cities. This weighted graph, or network, could then be used to determine the cheapest path from one city to another. The weight of a path in a weighted graph is the sum of the weights of the edges in the path.

Networks may be either undirected or directed, depending on the need. Take our airfare example from Figure 15.5. What if the airfare to fly from New York to Boston is one price, but the airfare to fly from Boston to New York is a different price? This would be an excellent application of a directed network, as illustrated in Figure 15.6.

For networks, we represent each edge with a triple that includes the starting vertex, the ending vertex, and the weight. Keep in mind that for undirected networks,

KEY CONCEPT

A network, or weighted graph, is a graph with weights or costs associated with each edge.

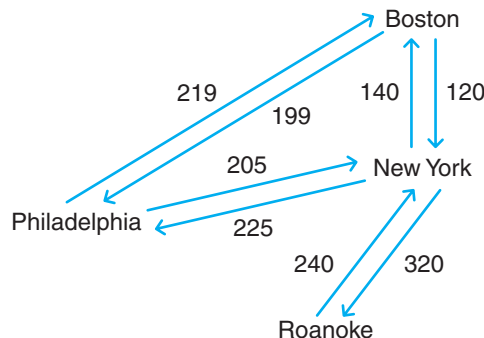


FIGURE 15.6 A directed network

the starting and ending vertices could be swapped with no impact. However, for directed networks, a triple must be included for every directional connection. For example, the network of Figure 15.6 would be represented as follows:

Vertices: Boston, New York, Philadelphia, Roanoke
 Edges: (Boston, New York, 120), (Boston, Philadelphia, 199),
 (New York, Boston, 140), (New York, Philadelphia, 225),
 (New York, Roanoke, 320), (Philadelphia, Boston, 219),
 (Philadelphia, New York, 205), (Roanoke, New York, 240)

15.4 Common Graph Algorithms

There are a number of common graph algorithms that may apply to undirected graphs, directed graphs, and/or networks. These include various traversal algorithms similar to what we explored with trees, as well as algorithms for finding the shortest path, algorithms for finding the least costly path in a network, and algorithms to answer simple questions about the graph, such as whether the graph is connected and what the shortest path is between two vertices.

Traversals

In our discussion of trees in Chapter 10, we defined four types of traversals and then implemented them as iterators: preorder traversal, inorder traversal, post-order traversal, and level-order traversal. Because we know that a tree is a graph, we know that for certain types of graphs, these traversals would still apply. Generally, however, we divide graph traversal into two categories: a *breadth-first* traversal, which behaves very much like the level-order traversal of a tree, and a *depth-first* traversal, which behaves very much like the preorder traversal of a tree. One difference here is that there is not a root node. Thus our traversal may start at any vertex in the graph.

We can construct a breadth-first traversal for a graph using a queue and an unordered list. We will use the queue (traversal-queue) to manage the traversal and the unordered list (result-list) to build our result. The first step is to enqueue the starting vertex into the traversal-queue and mark the starting vertex as visited. We then begin a loop that will continue until the traversal-queue is empty. Within this loop, we will take the first vertex off the traversal-queue and add that vertex to the rear of the result-list. Next, we will enqueue each of the vertices that are adjacent to the current one, and have not already been marked as visited, into the traversal-queue, mark each of them as visited, and then repeat the loop. We simply

repeat this process for each of the visited vertices until the traversal-queue is empty, meaning we can no longer reach any new vertices. The result-list now contains the vertices in breadth-first order from the given starting point. Very similar logic can be used to construct a breadth-first iterator. The `iteratorBFS` shows an iterative algorithm for this traversal for an array implementation of a graph. The determination of vertices that are adjacent to the current one depends on the implementation we choose to represent edges in a graph. This particular method assumes an implementation using an adjacency matrix. We will discuss this further in Section 15.5.

A depth-first traversal for a graph can be constructed using virtually the same logic by simply replacing the traversal-queue with a traversal-stack. One other difference in the algorithm, however, is that we do not want to mark a vertex as visited until it has been added to the result-list. The `iteratorDFS` method illustrates this algorithm for an array implementation of a graph.



VideoNote

Illustration of depth-first and breadth-first traversals of a graph

KEY CONCEPT

The only difference between a depth-first traversal of a graph and a breadth-first traversal is that the depth-first traversal uses a stack instead of a queue to manage the traversal.

```
/**
 * Returns an iterator that performs a breadth first
 * traversal starting at the given index.
 *
 * @param startIndex the index from which to begin the traversal
 * @return an iterator that performs a breadth first traversal
 */
public Iterator<T> iteratorBFS(int startIndex)
{
    Integer x;
    QueueADT<Integer> traversalQueue = new LinkedList<Integer>();
    UnorderedListADT<T> resultList = new ArrayUnorderedList<T>();

    if (!indexIsValid(startIndex))
        return resultList.iterator();

    boolean[] visited = new boolean[numVertices];
    for (int i = 0; i < numVertices; i++)
        visited[i] = false;

    traversalQueue.enqueue(new Integer(startIndex));
    visited[startIndex] = true;

    while (!traversalQueue.isEmpty())
    {
        x = traversalQueue.dequeue();
        resultList.addToRear(vertices[x.intValue()]);
    }
}
```

```

    // Find all vertices adjacent to x that have not been visited
    // and queue them up
    for (int i = 0; i < numVertices; i++)
    {
        if (adjMatrix[x.intValue()][i] && !visited[i])
        {
            traversalQueue.enqueue(new Integer(i));
            visited[i] = true;
        }
    }
}
return new GraphIterator(resultList.iterator());
}

/**
 * Returns an iterator that performs a depth first traversal
 * starting at the given index.
 *
 * @param startIndex the index from which to begin the traversal
 * @return an iterator that performs a depth first traversal
 */
public Iterator<T> iteratorDFS(int startIndex)
{
    Integer x;
    boolean found;
    StackADT<Integer> traversalStack = new LinkedStack<Integer>();
    UnorderedListADT<T> resultList = new ArrayUnorderedList<T>();
    boolean[] visited = new boolean[numVertices];

    if (!indexIsValid(startIndex))
        return resultList.iterator();

    for (int i = 0; i < numVertices; i++)
        visited[i] = false;

    traversalStack.push(new Integer(startIndex));
    resultList.addToRear(vertices[startIndex]);
    visited[startIndex] = true;

    while (!traversalStack.isEmpty())
    {
        x = traversalStack.peek();
        found = false;

        // Find a vertex adjacent to x that has not been visited
        // and push it on the stack

```

```
for (int i = 0; (i < numVertices) && !found; i++)
{
    if (adjMatrix[x.intValue()][i] && !visited[i])
    {
        traversalStack.push(new Integer(i));
        resultList.addToRear(vertices[i]);
        visited[i] = true;
        found = true;
    }
}
if (!found && !traversalStack.isEmpty())
    traversalStack.pop();
}
return new GraphIterator(resultList.iterator());
}
```

Let's look at an example. Figure 15.7 shows a sample undirected graph where each vertex is labeled with an integer. For a breadth-first traversal starting from vertex 9, we do the following:

1. Add 9 to the traversal-queue and mark it as visited.
2. Dequeue 9 from the traversal-queue.
3. Add 9 to the result-list.
4. Add 6, 7, and 8 to the traversal-queue, marking each of them as visited.
5. Dequeue 6 from the traversal-queue.
6. Add 6 to the result-list.

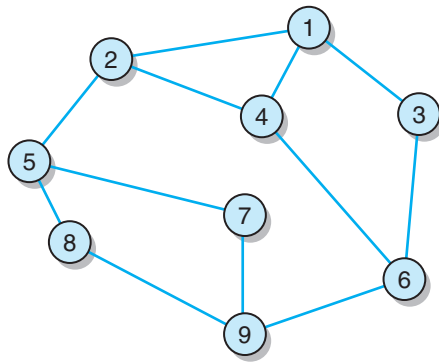


FIGURE 15.7 A traversal example

7. Add 3 and 4 to the traversal-queue, marking them both as visited.
8. Dequeue 7 from the traversal-queue and add it to the result-list.
9. Add 5 to the traversal-queue, marking it as visited.
10. Dequeue 8 from the traversal-queue and add it to the result-list. (We do not add any new vertices to the traversal-queue because there are no neighbors of 8 that have not already been visited.)
11. Dequeue 3 from the traversal-queue and add it to the result-list.
12. Add 1 to the traversal-queue, marking it as visited.
13. Dequeue 4 from the traversal-queue and add it to the result-list.
14. Add 2 to the traversal-queue, marking it as visited.
15. Dequeue 5 from the traversal-queue and add it to the result-list. (Because there are no unvisited neighbors, we continue without adding anything to the traversal-queue.)
16. Dequeue 1 from the traversal-queue and add it to the result-list. (Because there are no unvisited neighbors, we continue without adding anything to the traversal-queue.)
17. Dequeue 2 from the traversal-queue and add it to the result-list.

The result-list now contains the breadth-first order starting at vertex 9: 9, 6, 7, 8, 3, 4, 5, 1, and 2. Try tracing a depth-first search on the same graph from Figure 15.7.

Of course, both of these algorithms could be expressed recursively. For example, the following algorithm recursively defines a depth-first search:

```
DepthFirstSearch(node x)
{
    visit(x)
    result-list.addToRear(x)
    for each node y adjacent to x
        if y not visited
            DepthFirstSearch(y)
}
```

Testing for Connectivity

KEY CONCEPT

A graph is connected if and only if the number of vertices in the breadth-first traversal is the same as the number of vertices in the graph, regardless of the starting vertex.

In our earlier discussion, we defined a graph as *connected* if, for any two vertices in the graph, there is a path between them. This definition holds true for both undirected and directed graphs. Given the algorithm we just discussed, there is a simple solution to the question of whether a graph is connected: The graph is connected if and only if, for each vertex v in a graph containing n vertices, the size of the result of a breadth-first traversal starting at v is n .

Let's look at the examples of undirected graphs in Figure 15.8. We stated earlier that the graph on the left is connected and the graph on the right is not. Let's

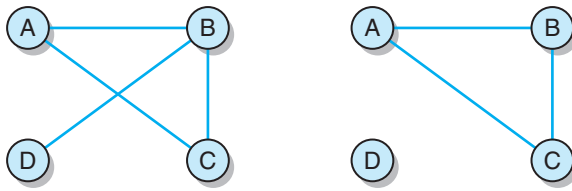


FIGURE 15.8 Connectivity in an undirected graph

Starting Vertex	Breadth-First Traversal
A	A, B, C, D
B	B, A, D, C
C	C, B, A, D
D	D, B, A, C

FIGURE 15.9 Breadth-first traversal for a connected undirected graph

confirm that by following our algorithm. Figure 15.9 shows the breadth-first traversals for the graph on the left using each of the vertices as a starting point. As you can see, all of the traversals yield $n = 4$ vertices, so the graph is connected. Figure 15.10 shows the breadth-first traversals for the graph on the right using each of the vertices as a starting point. Not only does none of the traversals contain $n = 4$ vertices, but the one starting at vertex D has only the one vertex. Thus the graph is not connected.

Starting Vertex	Breadth-First Traversal
A	A, B, C
B	B, A, C
C	C, B, A
D	D

FIGURE 15.10 Breadth-first traversal for an unconnected undirected graph

KEY CONCEPT

A spanning tree is a tree that includes all of the vertices of a graph and some, but possibly not all, of the edges.

KEY CONCEPT

A minimum spanning tree is a spanning tree where the sum of the weights of the edges is less than or equal to the sum of the weights for any other spanning tree for the same graph.

Minimum Spanning Trees

A *spanning tree* is a tree that includes all of the vertices of a graph and some, but possibly not all, of the edges. Because trees are also graphs, for some graphs the graph itself will be a spanning tree, and thus the only spanning tree for that graph will include all of the edges. Figure 15.11 shows a spanning tree for our graph from Figure 15.7.

One interesting application of spanning trees is to find a minimum spanning tree for a weighted graph. A *minimum spanning tree* is a spanning tree where the sum of the weights of the edges is less than or equal to the sum of the weights for any other spanning tree for the same graph.

The algorithm for developing a minimum spanning tree was developed by Prim (1957) and is quite elegant. As we discussed earlier, each edge is represented by a triple that includes the starting vertex, the ending vertex, and the weight. We then pick an arbitrary starting vertex (it does not matter which one) and add it to our minimum spanning tree (MST). Next we add all of the edges that include our starting vertex to a minheap ordered by weight. Keep in mind that if we are dealing with a directed network, we will add only edges that start at the given vertex.

Next we remove the minimum edge from the minheap and add the edge and the new vertex to our MST. Next we add to our minheap all of the edges that include this new vertex and whose other vertex is not already in our MST. We continue this process until either our MST includes all of the vertices in our original graph or the minheap is empty. Figure 15.12 shows a weighted network and its associated minimum spanning tree. The `getMST` method illustrates this algorithm.

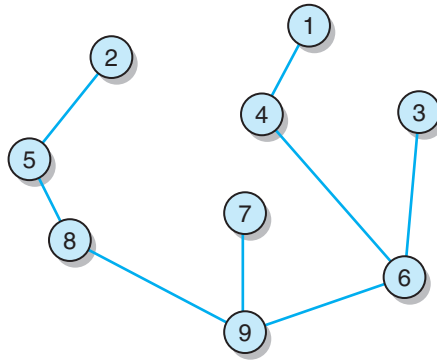


FIGURE 15.11 A spanning tree

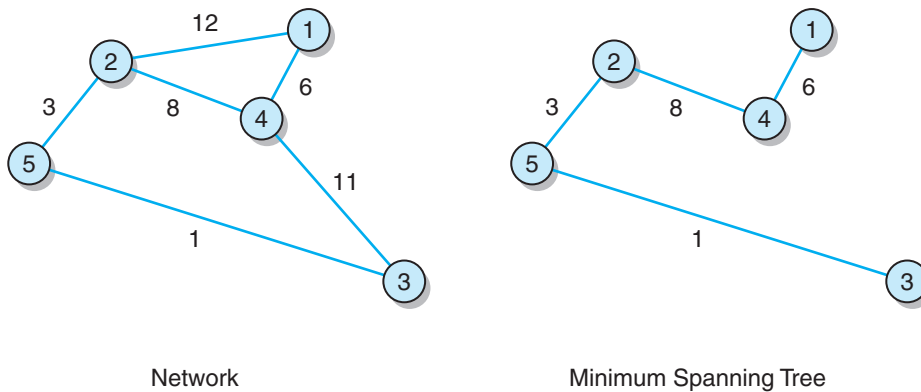


FIGURE 15.12 A network and its minimum spanning tree

```

/**
 * Returns a minimum spanning tree of the network.
 *
 * @return a minimum spanning tree of the network
 */
public Network mstNetwork()
{
    int x, y;
    int index;
    double weight;
    int[] edge = new int[2];
    HeapADT<Double> minHeap = new LinkedHeap<Double>();
    Network<T> resultGraph = new Network<T>();

    if (isEmpty() || !isConnected())
        return resultGraph;

    resultGraph.adjMatrix = new double[numVertices][numVertices];
    for (int i = 0; i < numVertices; i++)
        for (int j = 0; j < numVertices; j++)
            resultGraph.adjMatrix[i][j] = Double.POSITIVE_INFINITY;
    resultGraph.vertices = (T[]) (new Object[numVertices]);

    boolean[] visited = new boolean[numVertices];
    for (int i = 0; i < numVertices; i++)
        visited[i] = false;

    edge[0] = 0;
    resultGraph.vertices[0] = this.vertices[0];

```



```

resultGraph.numVertices++;
visited[0] = true;

// Add all edges, which are adjacent to the starting vertex,
// to the heap
for (int i = 0; i < numVertices; i++)
    minHeap.addElement(new Double(adjMatrix[0][i]));

while ((resultGraph.size() < this.size()) && !minHeap.isEmpty())
{
    // Get the edge with the smallest weight that has exactly
    // one vertex already in the resultGraph
    do
    {
        weight = (minHeap.removeMin()).doubleValue();
        edge = getEdgeWithWeightOf(weight, visited);
    } while (!indexIsValid(edge[0]) || !indexIsValid(edge[1]));

    x = edge[0];
    y = edge[1];
    if (!visited[x])
        index = x;
    else
        index = y;

    // Add the new edge and vertex to the resultGraph
    resultGraph.vertices[index] = this.vertices[index];
    visited[index] = true;
    resultGraph.numVertices++;

    resultGraph.adjMatrix[x][y] = this.adjMatrix[x][y];
    resultGraph.adjMatrix[y][x] = this.adjMatrix[y][x];

    // Add all edges, that are adjacent to the newly added vertex,
    // to the heap
    for (int i = 0; i < numVertices; i++)
    {
        if (!visited[i] && (this.adjMatrix[i][index] <
            Double.POSITIVE_INFINITY))
        {
            edge[0] = index;
            edge[1] = i;
            minHeap.addElement(new Double(adjMatrix[index][i]));
        }
    }
}
return resultGraph;
}

```

Determining the Shortest Path

There are two possibilities for determining the “shortest” path in a graph. The first, and perhaps simplest, possibility is to determine the literal shortest path between a starting vertex and a target vertex—that is, the least number of edges between the two vertices. This turns out to be a simple variation of our earlier breadth-first traversal algorithm.

To convert this algorithm to find the shortest path, we simply store two additional pieces of information for each vertex during our traversal: the path length from the starting vertex to this vertex, and the vertex that is the predecessor of this vertex in that path. Then we modify our loop to terminate when we reach our target vertex. The path length for the shortest path is simply the path length to the predecessor of the target + 1, and if we wish to output the vertices along the shortest path, we can simply backtrack along the chain of predecessors.

The second possibility for determining the shortest path is to look for the cheapest path in a weighted graph. Dijkstra (1959) developed an algorithm for this possibility that is similar to our previous algorithm. However, instead of using a queue of vertices that causes us to progress through the graph in the order in which we encounter vertices, we use a minheap or a priority queue storing vertex and weight pairs based on total weight (the sum of the weights from the starting vertex to this vertex) so that we always traverse through the graph following the cheapest path first. For each vertex, we must store the label of the vertex, the weight of the cheapest path (thus far) to that vertex from our starting point, and the predecessor of that vertex along that path. On the minheap, we will store vertex and weight pairs for each possible path that we have encountered but not yet traversed. As we remove a (vertex, weight) pair from the minheap, if we encounter a vertex with a weight less than the one already stored with the vertex, we update the cost.

15.5 Strategies for Implementing Graphs

Let us begin our discussion of implementation strategies by examining what operations will need to be available for a graph. Of course, we will need to be able to add vertices and edges to the graph and to remove them from it. There will need to be traversals (perhaps breadth-first and depth-first) beginning with a particular vertex, and these might be implemented as iterators, as we did for binary trees. Other operations like `size`, `isEmpty`, `toString`, and `find` will be useful as well. In addition to these, operations to determine the shortest path from a particular vertex to a particular target vertex, to determine the adjacency of two vertices, to construct a minimum spanning tree, and to test for connectivity will all probably need to be implemented.

Whatever storage mechanism we use for vertices must allow us to mark vertices as visited during traversals and other algorithms. This can be accomplished by simply adding a Boolean variable to the class representing the vertices.

Adjacency Lists

Because trees are graphs, perhaps the best introduction to how we might implement graphs is to consider the discussions and examples that we have already seen concerning the implementation of trees. One might immediately think of using a set of nodes where each node contains an element and $n-1$ links to other nodes. When we use this strategy with trees, the number of connections from any given node is limited by the order of the tree (e.g., a maximum of two directed edges starting at any particular node in a binary tree). Because of this limitation, we can specify, for example, that a binary node has a left and a right child pointer. Even if the binary node is a leaf, the pointer still exists. It is simply set to null.

In the case of a *graph node*, because each node could have up to $n-1$ edges connecting it to other nodes, it would be better to use a dynamic structure such as a linked list to store the edges within each node. This list is called an *adjacency list*. In the case of a network or a weighted graph, each edge would be stored as a triple including the weight. In the case of an undirected graph, an edge (A, B) would appear in the adjacency list of both vertex A and vertex B.

Adjacency Matrices

Keep in mind that we must somehow efficiently (in terms of both space and access time) store both vertices and edges. Because vertices are just elements, we can use any of our collections to store the vertices. In fact, we often talk about a “set of vertices,” the term *set* implying an implementation strategy. However, another solution for storing edges is motivated by our use of array implementations of trees, but instead of using a one-dimensional array, we will use a two-dimensional array that we call an *adjacency matrix*. In an adjacency matrix, each position of the two-dimensional array represents an intersection between two vertices in the graph. Each of these intersections is represented by a Boolean value indicating whether or not the two vertices are connected. Figure 15.13 shows the undirected graph that we began with at the beginning of this chapter. Figure 15.14 shows the adjacency matrix for this graph.

For any position (row, column) in the matrix, that position is true if and only if the edge ($v_{\text{row}}, v_{\text{column}}$) is in the graph. Because edges in an undirected graph are bidirectional, if (A, B) is an edge in the graph, then (B, A) is also in the graph.

Notice that this matrix is symmetrical—that is, each side of the diagonal is a mirror image of the other. This is because we are representing an undirected graph. For undirected graphs, it may not be necessary to represent the entire matrix; one side or the other of the diagonal may be enough.

	A	B	C	D
A	F	T	T	F
B	T	F	T	T
C	T	T	F	F
D	F	T	F	F

FIGURE 15.14 An adjacency matrix for an undirected graph

However, for directed graphs, because all of the edges are directional, the result can be quite different. Figure 15.15 shows a directed graph, and Figure 15.16 shows the adjacency matrix for this graph.

Adjacency matrices may also be used with networks or weighted graphs by simply storing an object at each position of the matrix to represent the weight of the edge. Positions in the matrix where edges do not exist would simply be set to null.

15.6 Implementing Undirected Graphs with an Adjacency Matrix

Like the other collections we have discussed, the first step in implementing a graph is to determine its interface. Listing 15.1 illustrates the `GraphADT` interface. Listing 15.2 illustrates the `NetworkADT` interface that extends the `GraphADT` interface. Note that our interfaces include methods to add and remove vertices, methods to add and remove edges, iterators for both breadth-first and depth-first traversals, methods to determine the shortest path between two vertices and to determine whether the graph is connected, and our usual collection of methods to determine the size of the collection, to determine whether it is empty, and to return a string representation of it.

	A	B	C	D
A	F	T	T	F
B	F	F	T	T
C	F	F	F	F
D	F	F	F	F

FIGURE 15.16 The adjacency matrix for the directed graph shown in Figure 15.15

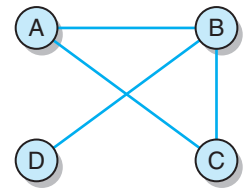


FIGURE 15.13 An undirected graph

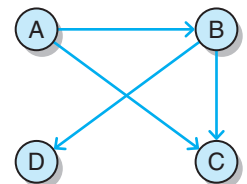


FIGURE 15.15 A directed graph

LISTING 15.1

```
package jsjf;

import java.util.Iterator;

/**
 * GraphADT defines the interface to a graph data structure.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public interface GraphADT<T>
{
    /**
     * Adds a vertex to this graph, associating object with vertex.
     *
     * @param vertex the vertex to be added to this graph
     */
    public void addVertex(T vertex);

    /**
     * Removes a single vertex with the given value from this graph.
     *
     * @param vertex the vertex to be removed from this graph
     */
    public void removeVertex(T vertex);

    /**
     * Inserts an edge between two vertices of this graph.
     *
     * @param vertex1 the first vertex
     * @param vertex2 the second vertex
     */
    public void addEdge(T vertex1, T vertex2);

    /**
     * Removes an edge between two vertices of this graph.
     *
     * @param vertex1 the first vertex
     * @param vertex2 the second vertex
     */
    public void removeEdge(T vertex1, T vertex2);
}
```

LISTING 15.1

continued

```
/**
 * Returns a breadth first iterator starting with the given vertex.
 *
 * @param startVertex the starting vertex
 * @return a breadth first iterator beginning at the given vertex
 */
public Iterator iteratorBFS(T startVertex);

/**
 * Returns a depth first iterator starting with the given vertex.
 *
 * @param startVertex the starting vertex
 * @return a depth first iterator starting at the given vertex
 */
public Iterator iteratorDFS(T startVertex);

/**
 * Returns an iterator that contains the shortest path between
 * the two vertices.
 *
 * @param startVertex the starting vertex
 * @param targetVertex the ending vertex
 * @return an iterator that contains the shortest path
 *         between the two vertices
 */
public Iterator iteratorShortestPath(T startVertex, T targetVertex);

/**
 * Returns true if this graph is empty, false otherwise.
 *
 * @return true if this graph is empty
 */
public boolean isEmpty();

/**
 * Returns true if this graph is connected, false otherwise.
 *
 * @return true if this graph is connected
 */
public boolean isConnected();

/**
 * Returns the number of vertices in this graph.
 *
 * @return the integer number of vertices in this graph
 */
public int size();
```

LISTING 15.1 *continued*

```

/**
 * Returns a string representation of the adjacency matrix.
 *
 * @return a string representation of the adjacency matrix
 */
public String toString();
}

```

LISTING 15.2

```

package jsjf;

import java.util.Iterator;

/**
 * NetworkADT defines the interface to a network.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public interface NetworkADT<T> extends GraphADT<T>
{

    /**
     * Inserts an edge between two vertices of this graph.
     *
     * @param vertex1 the first vertex
     * @param vertex2 the second vertex
     * @param weight the weight
     */
    public void addEdge(T vertex1, T vertex2, double weight);

    /**
     * Returns the weight of the shortest path in this network.
     *
     * @param vertex1 the first vertex
     * @param vertex2 the second vertex
     * @return the weight of the shortest path in this network
     */
    public double shortestPathWeight(T vertex1, T vertex2);
}

```

Of course, this interface could be implemented a variety of ways, but we will focus our discussion on an adjacency matrix implementation. The other implementations of undirected graphs and networks, as well as the implementations of directed graphs and networks, are left as programming projects. The header and instance data for our implementation are presented to provide context. Note that the adjacency matrix is represented by a two-dimensional Boolean array.

```
package jsjf;

import jsjf.exceptions.*;
import java.util.*;

/**
 * Graph represents an adjacency matrix implementation of a graph.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class Graph<T> implements GraphADT<T>
{
    protected final int DEFAULT_CAPACITY = 5;
    protected int numVertices; // number of vertices in the graph
    protected boolean[][] adjMatrix; // adjacency matrix
    protected T[] vertices; // values of vertices
    protected int modCount;
```

Our constructor simply initializes the number of vertices to zero, constructs the adjacency matrix, and sets up an array of generic objects (`T[]`) to represent the vertices.

```
/**
 * Creates an empty graph.
 */
public Graph()
{
    numVertices = 0;
    this.adjMatrix = new boolean[DEFAULT_CAPACITY][DEFAULT_CAPACITY];
    this.vertices = (T[]) (new Object[DEFAULT_CAPACITY]);
}
```


The addEdge Method

Once we have established our list of vertices and our adjacency matrix, adding an edge is simply a matter of setting the appropriate locations in the adjacency matrix to true. Our `addEdge` method uses the `getIndex` method to locate the proper indices and calls a different version of the `addEdge` method to make the assignments if the indices are valid.

```
/**
 * Inserts an edge between two vertices of the graph.
 *
 * @param vertex1 the first vertex
 * @param vertex2 the second vertex
 */
public void addEdge(T vertex1, T vertex2)
{
    addEdge(getIndex(vertex1), getIndex(vertex2));
}
```

```
/**
 * Inserts an edge between two vertices of the graph.
 *
 * @param index1 the first index
 * @param index2 the second index
 */
public void addEdge(int index1, int index2)
{
    if (indexIsValid(index1) && indexIsValid(index2))
    {
        adjMatrix[index1][index2] = true;
        adjMatrix[index2][index1] = true;
        modCount++;
    }
}
```

The addVertex Method

Adding a vertex to the graph involves adding the vertex in the next available position in the array and setting all of the appropriate locations in the adjacency matrix to false.

```

/**
 * Adds a vertex to the graph, expanding the capacity of the graph
 * if necessary. It also associates an object with the vertex.
 *
 * @param vertex the vertex to add to the graph
 */
public void addVertex(T vertex)
{
    if ((numVertices + 1) == adjMatrix.length)
        expandCapacity();

    vertices[numVertices] = vertex;
    for (int i = 0; i < numVertices; i++)
    {
        adjMatrix[numVertices][i] = false;
        adjMatrix[i][numVertices] = false;
    }
    numVertices++;
    modCount++;
}

```

The expandCapacity Method

The `expandCapacity` method for our adjacency matrix implementation of a graph is more interesting than the similar method in other array implementations. It is no longer just a case of expanding one array and copying the contents. Keep in mind that for our graph, we must not only expand the array of vertices and copy the existing vertices into the new array; we must also expand the capacity of the adjacency list and copy the old contents into the new list.

```

/**
 * Creates new arrays to store the contents of the graph with
 * twice the capacity.
 */
protected void expandCapacity()
{
    T[] largerVertices = (T[]) (new Object[vertices.length*2]);
    boolean[][] largerAdjMatrix =
        new boolean[vertices.length*2][vertices.length*2];

    for (int i = 0; i < numVertices; i++)
    {
        for (int j = 0; j < numVertices; j++)
        {

```

```

        largerAdjMatrix[i][j] = adjMatrix[i][j];
    }
    largerVertices[i] = vertices[i];
}

vertices = largerVertices;
adjMatrix = largerAdjMatrix;
}

```

Other Methods

The remaining methods for our graph implementation are left as programming projects, as is the implementation of a network.

Summary of Key Concepts

- An undirected graph is a graph where the pairings that represent the edges are unordered.
- Two vertices in a graph are adjacent if there is an edge connecting them.
- An undirected graph is considered complete if it has the maximum number of edges connecting vertices.
- A path is a sequence of edges that connects two vertices in a graph.
- A cycle is a path in which the first and last vertices are the same and none of the edges is repeated.
- An undirected tree is a connected, acyclic, undirected graph with one element designated as the root.
- A directed graph, sometimes referred as a digraph, is a graph where the edges are ordered pairs of vertices.
- A path in a directed graph is a sequence of directed edges that connects two vertices in the graph.
- A network, or a weighted graph, is a graph with weights or costs associated with each edge.
- The only difference between a depth-first traversal of a graph and a breadth-first traversal is that the depth-first traversal uses a stack instead of a queue to manage the traversal.

- A graph is connected if and only if the number of vertices in the breadth-first traversal is the same as the number of vertices in the graph, regardless of the starting vertex.
- A spanning tree is a tree that includes all of the vertices of a graph and some, but possibly not all, of the edges.
- A minimum spanning tree is a spanning tree where the sum of the weights of the edges is less than or equal to the sum of the weights for any other spanning tree for the same graph.

Summary of Terms

graph A graph is made up of nodes and the connections between those nodes.

vertices Nodes within a graph.

edges Connections between nodes in a graph.

undirected graph A graph where the pairings that represent the edges are unordered.

adjacent Two vertices are adjacent if there is an edge connecting them.

self-loop An edge of a graph that connects a vertex to itself.

complete An undirected graph is considered complete if it has the maximum number of edges connecting vertices.

path A sequence of edges that connects two vertices in a graph.

path length The number of edges in the path (or the number of vertices $- 1$).

connected An undirected graph is considered connected if, for any two vertices in the graph, there is a path between them.

cycle A path in which the first and last vertices are the same and none of the edges is repeated.

acyclic A graph that has no cycles.

directed graph (digraph) A graph where the edges are ordered pairs of vertices.

topological order The order of vertices for an acyclic directed graph where A precedes B if an edge exists from A to B.

network (weighted graph) A graph with weights or costs associated with each edge.

breadth-first traversal A traversal of a graph that behaves like a level-order traversal of a tree.

depth-first traversal A traversal of a graph that behaves like a preorder traversal of a tree.

spanning tree A tree that includes all of the vertices of a graph and some, but possibly not all, of the edges.

minimum spanning tree A spanning tree for a network where the sum of the weights of the edges is less than or equal to the sum of the weights for any other spanning tree.

adjacency list For any given node in a graph, the list of edges connecting it to other nodes. In the case of a network, each entry in the list also includes the weight or cost of the edge.

adjacency matrix A two-dimensional array where each location in the array represents the intersection between two vertices in the graph. In the case of an undirected graph, each position in the array is simply a Boolean. In the case of a weighted graph, the weight of the edge is stored in the array.

Self-Review Questions

- SR 15.1 What is the difference between a graph and a tree?
- SR 15.2 What is an undirected graph?
- SR 15.3 What is a directed graph?
- SR 15.4 What does it mean to say that a graph is complete?
- SR 15.5 What is the maximum number of edges for an undirected graph? What is the maximum number of edges for a directed graph?
- SR 15.6 Give the definition of a path and the definition of a cycle.
- SR 15.7 What is the difference between a network and a graph?
- SR 15.8 What is a spanning tree? What is a minimum spanning tree?

Exercises

- EX 15.1 Draw the undirected graph that is represented as follows:
 Vertices: 1, 2, 3, 4, 5, 6, 7
 Edges: (1, 2), (1, 4), (2, 3), (2, 4), (3, 7), (4, 7),
 (4, 6), (5, 6), (5, 7), (6, 7)
- EX 15.2 Is the graph from Exercise 15.1 connected? Is it complete?
- EX 15.3 List all of the cycles in the graph from Exercise 15.1.
- EX 15.4 Draw a spanning tree for the graph from Exercise 15.1.

- EX 15.5 Using the data in Exercise 15.1, draw the resulting directed graph.
- EX 15.6 Is the directed graph of Exercise 15.5 connected? Is it complete?
- EX 15.7 List all of the cycles in the graph of Exercise 15.5.
- EX 15.8 Draw a spanning tree for the graph of Exercise 15.5.
- EX 15.9 Consider the weighted graph shown in Figure 15.10. List all of the possible paths from vertex 2 to vertex 3, along with the total weight of each path.

Programming Projects

- PP 15.1 Implement an undirected graph using an adjacency list. Keep in mind that you must store both vertices and edges. Your implementation must implement the `GraphADT` interface.
- PP 15.2 Repeat Programming Project 15.1 for a directed graph.
- PP 15.3 Complete the implementation of a graph using an adjacency matrix that was presented in this chapter.
- PP 15.4 Extend the adjacency matrix implementation presented in this chapter to create an implementation of a weighted graph, or network.
- PP 15.5 Extend the adjacency matrix implementation presented in this chapter to create a directed graph.
- PP 15.6 Extend your implementation from Programming Project 15.1 to create a weighted, undirected graph.
- PP 15.7 Create a limited airline scheduling system that will allow a user to enter city-to-city connections and their prices. Your system should then allow a user to enter two cities and should return the shortest path and the cheapest path between the two cities. Your system should report if there is no connection between two cities. Assume an undirected network.
- PP 15.8 Repeat Programming Project 15.7 assuming a directed network.
- PP 15.9 Create a simple graphical application that will produce a textual representation of the shortest path and the cheapest path between two vertices in a network.
- PP 15.10 Create a network routing system that, given the point-to-point connections in the network and the costs of utilizing each, will produce cheapest-path connections from each point to each point in the network and will report any disconnected locations.

Answers to Self-Review Questions

- SRA 15.1 A graph is the more general concept, without the restriction that each node have one and only one parent except for the root, which does not have a parent. In the case of a graph, there is no root, and each vertex can be connected to up to $n-1$ other vertices.
- SRA 15.2 An undirected graph is a graph where the pairings that represent the edges are unordered.
- SRA 15.3 A directed graph, sometimes referred to as a digraph, is a graph where the edges are ordered pairs of vertices.
- SRA 15.4 A graph is considered complete if it has the maximum number of edges connecting vertices.
- SRA 15.5 The maximum number of edges for an undirected graph is $n(n-1)/2$. For a directed graph, it is $n(n-1)$.
- SRA 15.6 A path is a sequence of edges that connects two vertices in a graph. A cycle is a path in which the first and last vertices are the same and none of the edges is repeated.
- SRA 15.7 A network is a graph, either directed or undirected, with weights or costs associated with each edge.
- SRA 15.8 A spanning tree is a tree that includes all of the vertices of a graph and some, but possibly not all, of the edges. A minimum spanning tree is a spanning tree where the sum of the weights of the edges is less than or equal to the sum of the weights for any other spanning tree for the same graph.

References

- Collins, W. J. *Data Structures: An Object-Oriented Approach*. Reading, Mass.: Addison-Wesley, 1992.
- Dijkstra, E. W. "A Note on Two Problems in Connection with Graphs." *Numerische Mathematik 1* (1959): 269–271.
- Drosdek, A. *Data Structures and Algorithms in Java*. Pacific Grove, Cal.: Brooks/Cole, 2001.
- Prim, R. C. "Shortest Connection Networks and Some Generalizations." *Bell System Technical Journal 36* (1957): 1389–1401.



UML



Appendix

The Unified Modeling Language (UML)

Software engineering deals with the analysis, synthesis, and communication of ideas in the development of software systems. In order to facilitate the methods and practices necessary to accomplish these goals, software engineers have developed a wide variety of notations to capture and communicate information. Although numerous notations are available, only a few have become popular, and one in particular has become a de facto standard in the industry.

KEY CONCEPT

The Unified Modeling Language (UML) provides a notation with which we can capture and illustrate program designs.

The *Unified Modeling Language* (UML) was developed in the mid-1990s, but it is actually a synthesis of three separate and time-honored design notations, each popular in its own right. We use UML notation throughout this text to illustrate program designs, and this section describes the key aspects of UML diagrams. Keep in mind that UML is language-independent. It uses generic terms and contains some features

that are not relevant to the Java programming language. We focus on aspects of UML that are particularly appropriate for its use in this text.

UML is an object-oriented modeling language. It provides a convenient way to represent the relationships among classes and objects in a software system. We provide an overview of UML here, and use it throughout the text. The details of the underlying object-oriented concepts are discussed in Appendix B.

UML Class Diagrams

A UML *class diagram* describes the classes in the system, the static relationships among them, the attributes and operations associated with a class, and the constraints on the connections among objects. The terms *attribute* and *operation* are generic object-oriented terms. An *attribute* is any class-level data, including variables and constants. An *operation* is essentially equivalent to a method.

A class is represented in UML by a rectangle, which is usually divided into three sections containing the class name, its attributes, and its operations. Figure A.1

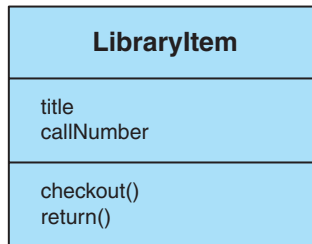


FIGURE A.1 `LibraryItem` class diagram

illustrates a class named `LibraryItem`. There are two attributes associated with the class, `title` and `callNumber`, and there are two operations associated with the class, `checkout` and `return`.

In the notation for a class, the attributes and operations are optional. Therefore, a class may be represented by a single rectangle containing only the class name, if desired. We can include the attributes and/or operations whenever they help convey important information in the diagram. If attributes or operations are included, then both sections are shown (though not necessarily filled) to make it clear which is which.

There are many additional pieces of information that can be included in the UML class notation. An annotation bracketed using `<` and `>` is called a *stereotype* in UML terminology. The `<abstract>` stereotype or the `<interface>` stereotype could be added above the name to indicate that it is representing an abstract class or an interface. The visibility of a class is assumed to be public by default, although nonpublic classes can be identified by using a property string in curly braces, such as `{private}`.

UML diagrams may be extraordinarily abstract, containing only the name of each class, or extremely detailed, including detailed information about each attribute and operation, or anywhere in between. For example, we may choose to provide only the attribute names, as we did in Figure A.1, or we may provide several pieces of additional information. The full syntax for showing an attribute is

visibility name:type = default-value

The visibility may be spelled out as `public`, `protected`, or `private`, or you may use the symbols `+` to represent public visibility, `#` for protected visibility, or `-` for private visibility. For example, we might have listed the title of a `LibraryItem` as

```
- title : String
```

indicating that the attribute `title` is a private variable of type `String`. A default value is not provided in this case. Also, the stereotype `<final>` may be added to an attribute to indicate that it is a constant.

Similarly, we may choose to provide only the name of each method, or we may choose to provide the full syntax for an operation:

visibility name (parameter-list): return-type {property-string}

As with the syntax for attributes, all of the items other than the name are optional. The visibility modifiers are the same as they are for attributes. The *parameter-list* can include the name and type of each parameter, separated by a colon. The *return-type* is the type of the value returned from the operation.

KEY CONCEPT

Various kinds of relationships can be represented in a UML class diagram.

The scope and detail of a UML class diagram is a choice. For example, we know that in Java, all classes are derived from the class `java.lang.Object`. Thus, we could include `java.lang.Object` in every UML class diagram for systems created in Java. In general, we include classes from the Java API in our class diagrams only when they are important to us. For example, if we derived a new kind of list from the `java.util.ArrayList` class, we would probably include the `ArrayList` class in our diagram, but not all of the other Java API classes with relationships to the `ArrayList` class.

UML Relationships

There are several kinds of relationships among classes that UML diagrams can represent. Usually they are shown as lines or arrows connecting one class to another. Specific types of lines and arrowheads have specific meaning in UML.

One type of relationship that can be shown between two classes in a UML diagram is *inheritance*. Figure A.2 shows two classes that are derived from the `LibraryItem` class. Inheritance is shown using an arrow with an open arrowhead pointing from the child class to the parent class. This example shows that both the `Book` class and the `Video` class inherit all the attributes and operations of `LibraryItem`, but they also extend that definition with attributes of their own. Note

KEY CONCEPT

The inheritance relationship is indicative of one class being derived from, or being a child of, another class.

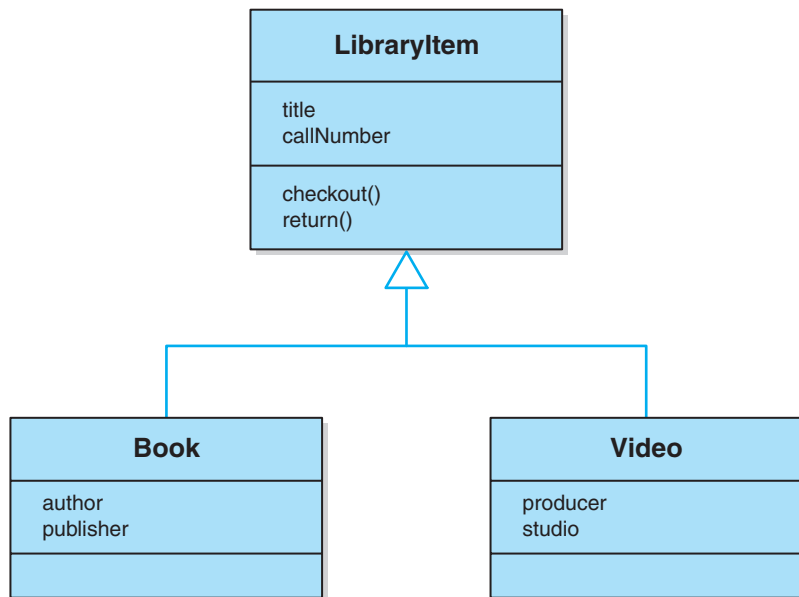


FIGURE A.2 A UML class diagram showing inheritance relationships

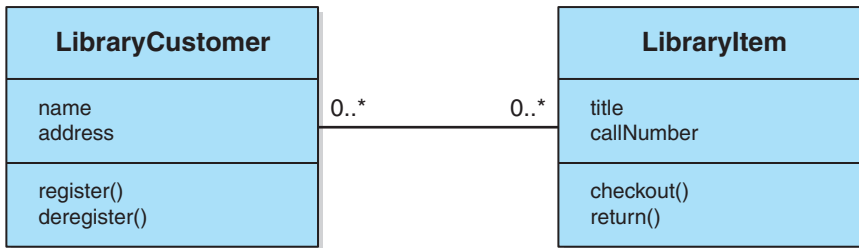


FIGURE A.3 A UML class diagram showing an association

that in this example, neither subclass has any operations other than those provided in the parent class.

Another relationship shown in a UML diagram is *association*, which represents relationships between instances (objects) of the classes. Association is indicated by a solid line between the two classes involved and can be annotated with the *cardinality* of the relationship on either side. For example, Figure A.3 shows an association between a `LibraryCustomer` and a `LibraryItem`. The cardinality of `0..*` means “zero or more,” in this case indicating that any given library customer may check out 0 or more items, and that any given library item may be checked out by multiple customers. The cardinality of an association may indicate other relationships, such as an exact number or a specific range. For example, if a customer were allowed to check out no more than five items, the cardinality could have been indicated by `0..5`.

A third type of relationship between classes is *aggregation*. This is the situation in which one class is essentially made up, at least in part, of other classes. For example, we can extend our library example to include a `CourseMaterials` class that is made up of books, course notes, and videos, as shown in Figure A.4. Aggregation is shown by using an open diamond on the aggregate end of the relationship.

A fourth type of relationship that we may wish to represent is *implementation*. This relationship occurs between an interface and any class that implements that interface. Figure A.5 shows an interface called `Copyrighted` that contains two abstract methods. The dotted arrow with the open arrowhead indicates that the `Book` class implements the `Copyrighted` interface.

A fifth type of relationship between classes is one class *using* another. Examples of this concept include an instructor using a chalkboard, a driver using a car, and a library customer using a computer. Figure A.6 illustrates this relationship, showing that a `LibraryCustomer` might use a `Computer`. The “uses” relationship is indicated by a dotted line with an open arrowhead that is generally annotated with the nature of the relationship.

KEY CONCEPT

The association relationship represents relationships between instances of classes.

KEY CONCEPT

The aggregation relationship represents one class being made up of other classes.

KEY CONCEPT

The implementation relationship represents a class implementing an interface.

KEY CONCEPT

The “uses” relationship represents one class using another.

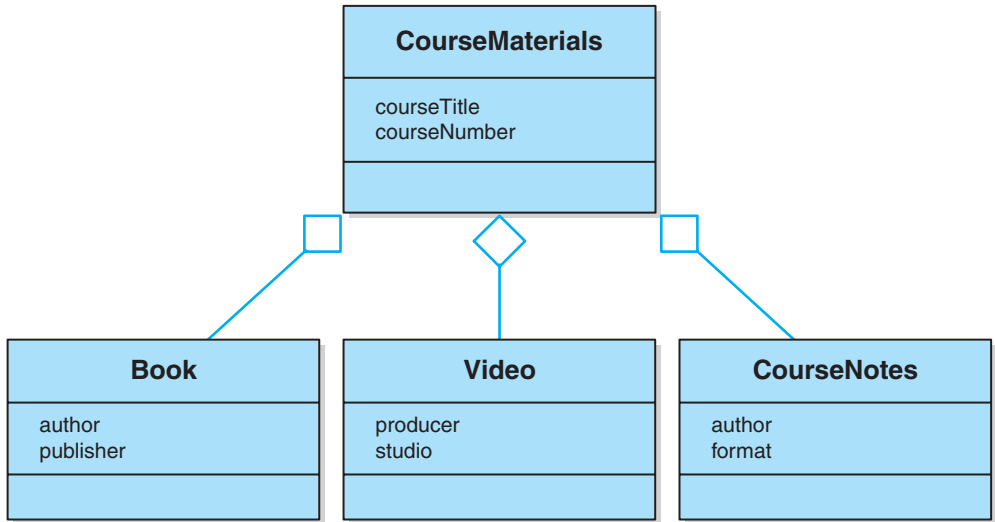


FIGURE A.4 One class shown as an aggregate of other classes

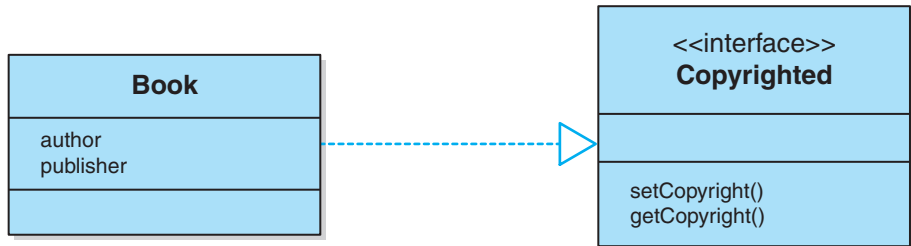


FIGURE A.5 A UML diagram showing a class implementing an interface



FIGURE A.6 A UML diagram showing the use of one class by another

Summary of Key Concepts

- The Unified Modeling Language (UML) provides a notation with which we can capture and illustrate program designs.
- Various kinds of relationships can be represented in a UML class diagram.
- The inheritance relationship is indicative of one class being derived from, or being a child of, another class.
- The association relationship represents relationships between instances of classes.
- The aggregation relationship represents one class being made up of other classes.
- The implementation relationship represents a class implementing an interface.
- The “uses” relationship represents one class using another.

Self-Review Questions

- SR A.1 What does a UML class diagram represent?
- SR A.2 What are the different types of relationships represented in a class diagram?

Exercises

- EX A.1 Create a UML class diagram for the organization of a university, where the university is made up of colleges, which are made up of departments, which contain faculty and students.
- EX A.2 Complete the UML class description for a library system outlined in this appendix.
- EX A.3 List and illustrate an example of each of the relationships discussed in this appendix.

Answers to Self-Review Questions

- SRA A.1 A class diagram describes the types of objects or classes in the system, the static relationships among them, the attributes and operations of a class, and the constraints on the connections among objects.
- SRA A.2 Relationships shown in a UML class diagram include inheritance, association, aggregation, the implementation of interfaces, and the use of one class by another.

This page is intentionally left blank.



Object-Oriented Design

B

Appendix

B.1 Overview of Object Orientation

Java is an object-oriented language. As the name implies, an *object* is a fundamental entity in a Java program. In addition to objects, a Java program also manages primitive data. *Primitive data* include common, fundamental values such as numbers and characters. An object usually represents something more specialized or complex, such as a bank account. An object often contains primitive values and is in part defined by them. For example, an object that represents a bank account might contain the account balance, which is stored as a primitive numeric value.

An object is defined by a *class*, which can be thought of as the data type of the object. The operations that can be performed on the object are defined by the methods in the class.

Once a class has been defined, multiple objects can be created from that class. For example, once we define a class to represent the concept of a bank account, we can create multiple objects that represent specific, individual bank accounts. Each bank account object would keep track of its own balance. This is an example of *encapsulation*, meaning that each object protects and manages its own information. The methods defined in the bank account class would allow us to perform operations on individual bank account objects. For instance, we might withdraw money from a particular account. We can think of these operations as services that the object performs. The act of invoking a method on an object is sometimes referred to as *sending a message* to the object, requesting that the service be performed.

Classes can be created from other classes using *inheritance*. That is, the definition of one class can be based on another class that already exists. Inheritance is a form of software *reuse*, capitalizing on the similarities among various kinds of classes that we may want to create. One class can be used to derive several new classes. Derived classes can then be used to derive even more classes. This creates a hierarchy of classes, where characteristics defined in one class are inherited by its children, which in turn pass them on to their children, and so on. For example, we might create a hierarchy of classes that represent various types of accounts. Common characteristics are defined in high-level classes, and specific differences are defined in derived classes.

Classes, objects, encapsulation, and inheritance are the primary ideas that make up the world of object-oriented software. They are depicted in Figure B.1 and are explored in more detail throughout this appendix.

B.2 Using Objects

The following `println` statement illustrates the process of using an object for the services it provides:

```
System.out.println("Whatever you are, be a good one.");
```

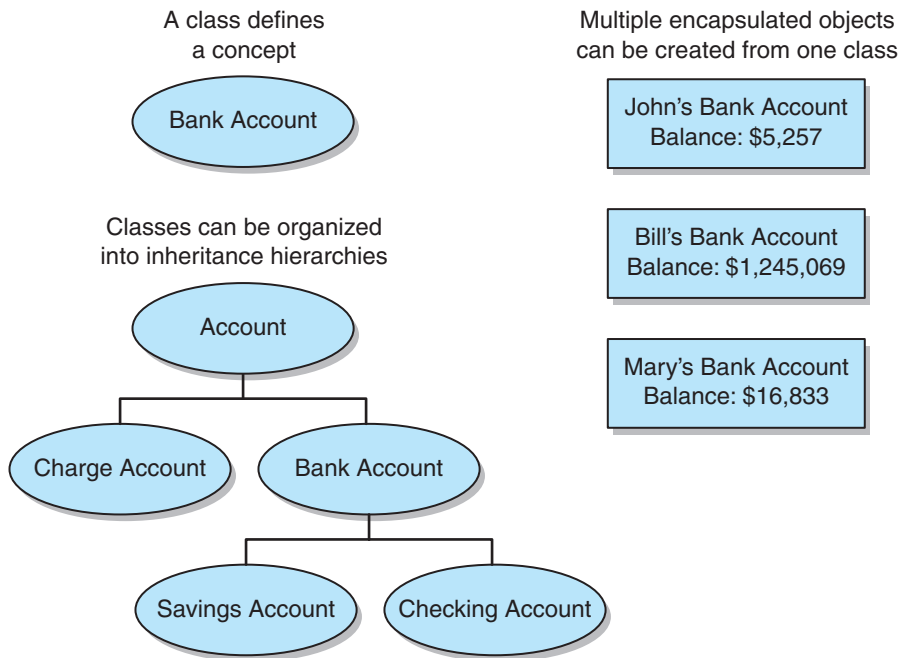


FIGURE B.1 Various aspects of object-oriented software

The `System.out` object represents an output device or file, which by default is the monitor screen. To be more precise, the object's name is `out`, and it is stored in the `System` class.

The `println` method represents a service that the `System.out` object performs for us. Whenever we request it, the object will print a string of characters to the screen. We can say that we send the `println` message to the `System.out` object to request that some text be printed.

Abstraction

An object is an *abstraction*, which means that the precise details of how it works are irrelevant from the point of view of the user of the object. We don't really need to know how the `println` method prints characters to the screen, as long as we can count on it to do its job. Of course, there are times when it is helpful to understand such information, but it is not necessary in order to *use* the object.

Sometimes it is important to hide or ignore certain details. A human being is capable of mentally managing around seven (plus or minus two) pieces of information in short-term memory. Beyond that, we start to lose track of some of the pieces.

However, if we group pieces of information together, then those pieces can be managed as one “chunk” in our minds. We don’t actively deal with all of the details in the chunk, but we can still manage it as a single entity. Therefore, we can deal with large quantities of information by organizing it into chunks. An object is a construct that organizes information and allows us to hide the details inside. An object is therefore a wonderful abstraction.

We use abstractions every day. Think about a car for a moment. You don’t necessarily need to know how a four-cycle combustion engine works in order to drive a car. You just need to know some basic operations: how to turn it on, how to put it in gear, how to make it move with the pedals and steering wheel, and how to stop it. These operations define the way a person interacts with the car. They mask the details of what is happening inside the car that enable it to function. When you are driving a car, you are not usually thinking about the spark plugs igniting the gasoline that drives the piston that turns the crankshaft that turns the axle that turns the wheels. If we had to worry about all of these underlying details, we would never be able to operate something as complicated as a car.

Initially, all cars had manual transmissions. The driver had to understand and deal with the details of changing gears with the stick shift. Eventually, automatic transmissions were developed, and the driver who bought a car so equipped no longer had to worry about shifting gears. Those details were hidden by raising the *level of abstraction*.

Of course, someone has to deal with the details. The car manufacturer has to know the details in order to design and build the car in the first place. A car mechanic relies on the fact that most people don’t have the expertise or tools necessary to fix a car when it breaks.

KEY CONCEPT

An abstraction hides details. A good abstraction hides the right details at the right time so that we can manage complexity.

The level of abstraction must be appropriate for each situation. Some people prefer to drive a car with a manual transmission. A race car driver, for instance, needs to control the shifting manually for optimum performance.

Likewise, someone has to create the code for the objects we use. Later in this appendix, we explore how to define objects by creating classes. For now, we can create and use objects from classes that have been defined for us already. Abstraction makes that possible.

Creating Objects

A Java variable can hold either a primitive value or a *reference to an object*. Like variables that hold primitive types, a variable that serves as an object reference must be declared. A class is used to define an object, and the class name can be thought of as the type of an object. The declarations of object references are similar in structure to the declarations of primitive variables.

The following declaration creates a reference to a `String` object:

```
String name;
```

That declaration is like the declaration of an integer, in that the type is followed by the variable name we want to use. However, no `String` object actually exists yet. To create an object, we use the `new` operator:

```
name = new String("James Gosling");
```

The act of creating an object by using the `new` operator is called *instantiation*. An object is said to be an *instance* of a particular class. After the `new` operator creates the object, a *constructor* is invoked to help set it up initially. A constructor has the same name as the class and is similar to a method. In this example, the parameter to the constructor is a string literal that specifies the characters that the `String` object will hold.

The acts of declaring the object reference variable and creating the object itself can be combined into one step by initializing the variable in the declaration, just as we do with primitive types:

```
String name = new String("James Gosling");
```

After an object has been instantiated, we use the *dot operator* to access its methods. The dot operator is appended directly after the object reference, followed by the method being invoked. For example, to invoke the `length` method defined in the `String` class, we use the dot operator on the `name` reference variable:

```
count = name.length();
```

An object reference variable (such as `name`) actually stores the address where the object is stored in memory. However, we don't usually care about the actual address value. We just want to access the object, wherever it is.

Even though they are not primitive types, strings are so fundamental and so frequently used that Java defines string literals delimited by double quotation marks, as we have seen in various examples. This is a shortcut notation. Whenever a string literal appears, a `String` object is created. Therefore, the following declaration is valid:

```
String name = "James Gosling";
```

That is, for `String` objects, the explicit use of the `new` operator, as well as the call to the constructor, can be eliminated. In most cases, this simplified syntax for strings is used.

KEY CONCEPT

The `new` operator returns a reference to a newly created object.

B.3 Class Libraries and Packages

A *class library* is a set of classes that supports the development of programs. A compiler often comes with a class library. Class libraries can also be obtained separately through third-party vendors. The classes in a class library contain methods that are often valuable to a programmer because of the special functionality they offer. In fact, programmers often become dependent on the methods in a class library and begin to think of them as part of the language. But technically, they are not in the language definition.

The `String` class, for instance, is not an inherent part of the Java language. It is part of the Java *standard class library* that can be found in any Java development environment. The classes that make up the library were created by employees at Sun Microsystems, the company that created the Java language.

KEY CONCEPT

The Java standard class library is a useful set of classes that anyone can use when writing Java programs.

KEY CONCEPT

A package is a Java language element used to group related classes under a common name.

The class library is made up of several clusters of related classes, which are sometimes called Java APIs. API stands for *application programmer interface*. For example, we may refer to the Java Database API when we are talking about the set of classes that helps us to write programs that interact with a database. Another example of an API is the Java Swing API, which consists of a set of classes that defines special graphical components used in a graphical user interface. Sometimes the entire standard library is referred to generically as the Java API.

The classes of the Java standard class library are also grouped into *packages*, which, like the APIs, enable us to group related classes under one name. Each class is part of a particular package. The `String` class and the `System` class, for example, are both part of the `java.lang` package.

The package organization is more fundamental and language-based than the API names. Although there is a general correspondence between package and API names, the groups of classes that make up a given API might cross packages. In this text, we refer to classes primarily in terms of their package organization.

The `import` Declaration

The classes of the package `java.lang` are automatically available for use when writing a program. To use classes from any other package, however, we must either *fully qualify* the reference or use an *import declaration*.

When you want to use a class from a class library in a program, you could use its fully qualified name, including the package name, every time it was referenced. For example, every time you wanted to refer to the `Random` class that is defined in the `java.util` package, you could write `java.util.Random`. However, completely

specifying the package and class name every time it is needed quickly becomes tiring. Java provides the `import` declaration to simplify these references.

The `import` declaration identifies the packages and classes that will be used in a program, so that the fully qualified name is not necessary with each reference. The following is an example of an `import` declaration:

```
import java.util.Random;
```

This declaration asserts that the `Random` class of the `java.util` package may be used in the program. Once this `import` declaration is made, it is sufficient to use the simple name `Random` when referring to that class in the program.

Another form of the `import` declaration uses an asterisk (*) to indicate that any class inside the package might be used in the program. Therefore, the declaration

```
import java.util.*;
```

allows all classes in the `java.util` package to be referenced in the program without the explicit package name. If only one class of a particular package will be used in a program, it is usually better to name the class specifically in the `import` declaration. However, if two or more classes will be used, the * notation is fine. Once a class is imported, it is just as if its code had been brought into the program. The code is not actually moved, but that is the effect.

The classes of the `java.lang` package are automatically imported because they are fundamental and can be thought of as basic extensions to the language. Therefore, any class in the `java.lang` package, such as `String`, can be used without an explicit `import` declaration. It is as if all programs automatically contained the following declaration:

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

B.4 State and Behavior

Think about objects in the world around you. How would you describe them? Let's use a ball as an example. A ball has particular characteristics, such as its diameter, color, and elasticity. Formally, we say the properties that describe an object, which are called *attributes*, define the object's *state of being*. We also describe a ball by what it does, such as the fact that it can be thrown, bounced, or rolled. These activities define the object's *behavior*.

All objects have a state and a set of behaviors. We can represent these characteristics in software objects as well. The values of an object's variables describe the object's state, and the methods that can be invoked using the object define the object's behaviors.

KEY CONCEPT

Each object has a state and a set of behaviors. The values of an object's variables define its state. The methods to which an object responds define its behaviors.

Consider a computer game that uses a ball. The ball could be represented as an object. It could have variables to store its size and location, and methods that draw it on the screen and calculate how it moves when thrown, bounced, or rolled. The variables and methods defined in the ball object establish the state and behavior that are relevant to the ball's use in the computerized ball game.

Each object has its own state. Each ball object has a particular location, for instance, which typically is different from the location of all other balls. Behaviors, though, tend to apply to all objects of a particular type. For instance, in general, any ball can be thrown, bounced, or rolled. The act of rolling a ball is generally the same for all balls.

The state of an object and that object's behaviors work together. How high a ball bounces depends on its elasticity. The action is the same, but the specific result depends on that particular object's state. An object's behavior often modifies its state. For example, when a ball is rolled, its location changes.

Any object can be described in terms of its state and behavior. Let's consider another example. In software that is used to manage a university, a student could be represented as an object. The collection of all such objects represents the entire student body at the university. Each student has a state. That is, each student object contains the variables that store information about a particular student, such as name, address, major, courses taken, grades, and grade point average. A student object also has behaviors. For example, the class of the student object may contain a method to add a new course.

Although software objects often represent tangible items, they don't have to. For example, an error message can be an object, with its state being the text of the message, and behaviors, including the process of issuing (perhaps printing) the error. A mistake that programmers new to the world of object-orientation often make is to limit the possibilities to tangible entities.

B.5 Classes

An object is defined by a class. A class is the model, pattern, or blueprint from which an object is created. Consider the blueprint that an architect creates when designing a house. The blueprint defines the important characteristics of the house: walls, windows, doors, electrical outlets, and so forth. Once the blueprint is created, several houses can be built using it.

In one sense, the houses built from the blueprint are different. They are in different locations, have different addresses, they contain different furniture, and different people live in them. Yet in many ways they are the "same" house. The layout of the rooms and other crucial characteristics are the same in each. To create a different house, we would need a different blueprint.

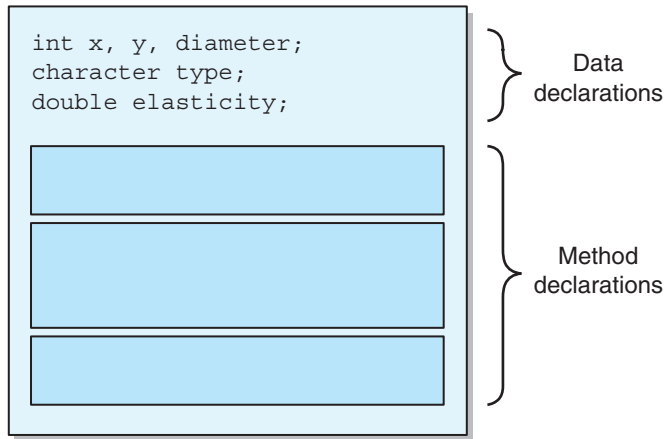


FIGURE B.2 The members of a class: data declarations and method declarations

A class is a blueprint of an object. But a class is not an object any more than a blueprint is a house. In general, no space to store data values is reserved in a class. To allocate space to store data values, we have to instantiate one or more objects from the class (static data are the exception to this rule and are discussed later in this appendix). Each object is an instance of a class. Each object has space for its own data, which is why each object can have its own state.

A class contains the declarations of the data that will be stored in each instantiated object, and the declarations of the methods that can be invoked using an object. Collectively these are called the *members* of the class. See Figure B.2.

Consider the class shown in Listing B.1, called `Coin`, which represents a coin that can be flipped and that, at any point in time, shows a face of either heads or tails.

In the `Coin` class, we have two integer constants, `HEADS` and `TAILS`, and one integer variable, `face`. The rest of the `Coin` class is composed of the `Coin` constructor and three regular methods: `flip`, `isHeads`, and `toString`.

Constructors are special methods that have the same name as the class. The `Coin` constructor gets called when the `new` operator is used to create a new instance of the `Coin` class. The rest of the methods in the `Coin` class define the various services provided by `Coin` objects.

A class we define can be used in multiple programs. This is no different from using the `String` class in whatever program we need it in. When designing a class, it is always good to look to the future to try to give the class behaviors that may be beneficial in other programs, not just behaviors that fit the specific purpose for which you are creating it at the moment.

KEY CONCEPT

A class is a blueprint for an object; it reserves no memory space for data. Each object has its own data space and thus its own state.

LISTING B.1

```
/**
 * Coin represents a coin with two sides that can be flipped.
 *
 * @author Lewis and Chase
 * @version 4.0
 */
public class Coin
{
    private final int HEADS = 0;
    private final int TAILS = 1;

    private int face;

    /**
     * Sets up the coin by flipping it initially.
     */
    public Coin()
    {
        flip();
    }

    /**
     * Flips the coin by randomly choosing a face value.
     */
    public void flip()
    {
        face = (int)(Math.random() * 2);
    }

    /**
     * Returns true if the current face of the coin is heads.
     *
     * @return true if the face is heads
     */
    public boolean isHeads()
    {
        return (face == HEADS);
    }
}
```

LISTING B.1*continued*

```
/**
 * Returns the current face of the coin as a string.
 *
 * @return the string representation of the current face value of this coin
 */
public String toString()
{
    String faceName;

    if (face == HEADS)
        faceName = "Heads";
    else
        faceName = "Tails";

    return faceName;
}
}
```

Instance Data

Note that in the `Coin` class, the constants `HEADS` and `TAILS` and the variable `face` are declared inside the class, but not inside any method. The location at which a variable is declared defines its *scope*, which is the area within a program in which that variable can be referenced. By being declared at the class level (not within a method), these variables and constants can be referenced in any method of the class.

Attributes declared at the class level are also called *instance data*, because memory space for the data is reserved for each instance of the class that is created. Each `Coin` object, for example, has its own `face` variable with its own data space. Therefore, at any point in time, two `Coin` objects can have their own states: one can be showing heads and the other can be showing tails, perhaps.

Java automatically initializes any variables declared at the class level. For example, all variables of numeric types such as `int` and `double` are initialized to zero. However, despite the fact that the language performs this automatic initialization, it is good practice to initialize variables explicitly (usually in a constructor) so that anyone reading the code will clearly understand the intent.

KEY CONCEPT

The scope of a variable, which determines where it can be referenced, depends on where it is declared.

B.6 Encapsulation

We can think about an object in one of two ways. The view we take depends on what we are trying to accomplish at the moment. First, when we are designing and implementing an object, we need to think about the details of how an object works. That is, we have to design the class; we have to define the variables that will be held in the object and define the methods that make the object useful.

However, when we are designing a solution to a larger problem, we have to think in terms of how the objects in the program interact. At that level, we have to think only about the services that an object provides, not about the details of how those services are provided. As we discussed earlier in this appendix, an object provides a level of abstraction that enables us to focus on the larger picture when we need to.

KEY CONCEPT

Objects should be encapsulated. The rest of a program should interact with an object only through a well-defined interface.

This abstraction works only if we are careful to respect its boundaries. An object should be *self-governing*, which means that the variables contained in an object should be modified only within the object. Only the methods within an object should have access to the variables in that object. We should make it difficult, if not impossible, for code outside of a class to “reach in” and change the value of a variable that is declared inside the class.

The object-oriented term for this characteristic is *encapsulation*. An object should be encapsulated from the rest of the system. It should interact with other parts of a program only through the specific set of methods that defines the services provided by that object. These methods define the *interface* between that object and the program that uses it.

The code that uses an object, which is sometimes called the *client* of an object, should not be allowed to access variables directly. The client should interact with the object’s methods, which in turn interact on behalf of the client with the data encapsulated within the object.

Visibility Modifiers

In Java, we accomplish object encapsulation using *modifiers*. A modifier is a Java reserved word that is used to specify particular characteristics of a programming language construct. For example, the `final` modifier is used to declare a constant. Java has several modifiers that can be used in various ways. Some modifiers can be used together, but some combinations are invalid.

Some Java modifiers are called *visibility modifiers* because they control access to the members of a class. The reserved words `public` and `private` are visibility modifiers that can be applied to the variables and methods of a class. If a member of a class has *public visibility*, then it can be directly referenced from outside of the object. If a member of a class has *private visibility*, it can be used anywhere

inside the class definition but cannot be referenced externally. A third visibility modifier, `protected`, is relevant only in the context of inheritance, which is discussed later in this appendix.

Public variables violate encapsulation. They allow code external to the class in which the data are defined to reach in and access or modify the value of the data. Therefore, instance data should be defined with private visibility. Data that are declared as private can be accessed only by the methods of the class, which makes the objects created from that class self-governing.

Which visibility we apply to a method depends on the purpose of that method. Methods that provide services to the client of the class must be declared with public visibility so that they can be invoked by the client. These methods are sometimes referred to as *service methods*. A private method cannot be invoked from outside the class. The only purpose of a private method is to help the other methods of the class do their job. Therefore, private methods are sometimes referred to as *support methods*.

The table in Figure B.3 summarizes the effects of public and private visibility on both variables and methods.

Note that a client can still access or modify `private` data by invoking service methods that change the data. A class must provide service methods for valid client operations. The code of those methods must be carefully designed to permit only appropriate access and valid changes.

Giving constants public visibility is generally considered acceptable. Although their values can be accessed directly, they cannot be changed because they were declared using the `final` modifier. Keep in mind that encapsulation means that data values should not be able to be *changed* directly by another part of the code. Because constants, by definition, cannot be changed, the encapsulation issue is largely moot.

KEY CONCEPT

Instance variables should be declared with private visibility to promote encapsulation.

	public	private
Variables	Violate encapsulation	Enforce encapsulation
Methods	Provide services to clients	Support other methods in the class

FIGURE B.3 The effects of public and private visibility

UML diagrams reflect the visibility of a class member with special notations. A member with public visibility is preceded by a plus sign (+), and a member with private visibility is preceded by a minus sign (-).

Local Data

As we noted earlier, the scope of a variable or constant is the part of a program in which a valid reference to that variable can be made. A variable can be declared inside a method, making it *local data* as opposed to instance data. Recall that instance data are declared in a class but not inside any particular method. The scope of local data is limited to the method in which they are declared. Any reference to local data of one method in any other method would cause the compiler to issue an error message. A local variable simply does not exist outside of the method in which it is declared. Instance data, declared at the class level, have a scope of the entire class. Any method of the class can refer to such data.

KEY CONCEPT

A variable declared in a method is local to that method and cannot be used outside of it.

Because local data and instance data operate at different levels of scope, it is possible to declare a local variable inside a method by using the same name as an instance variable declared at the class level. Referring to that name in the method will reference the local version of the variable. This naming practice obviously has the potential to confuse anyone reading the code, so it should be avoided.

The formal parameter names in a method header serve as local data for that method. They don't exist until the method is called, and they cease to exist when the method is exited.

B.7 Constructors

A constructor is similar to a method that is invoked when an object is instantiated. When we define a class, we usually define a constructor to help us set up the class. In particular, we often use a constructor to initialize the variables associated with each object.

A constructor differs from a regular method in two ways. First, the name of a constructor is the same as the name of the class. Therefore, the name of the constructor in the `Coin` class is `Coin`, and the name of the constructor in the `Account` class is `Account`. Second, a constructor cannot return a value and does not have a return type specified in the method header.

KEY CONCEPT

A constructor cannot have any return type, not even `void`.

A mistake that programmers often make is to put a `void` return type on a constructor. As far as the compiler is concerned, putting any return type on a constructor, even `void`, turns it into a regular method that happens to have the same name as the class. As such, it cannot be invoked as a constructor. This leads to error messages that are sometimes difficult to decipher.

A constructor is generally used to initialize the newly instantiated object. We don't have to define a constructor for every class. Each class has a *default constructor* that takes no parameters and is used if we don't provide our own. This default constructor generally has no effect on the newly created object.

B.8 Method Overloading

When a method is invoked, the flow of control transfers to the code that defines the method. After the method has been executed, control returns to the location of the call, and processing continues.

Often the method name is sufficient to indicate which method is being called by a specific invocation. But in Java, as in other object-oriented languages, you can use the same method name with different parameter lists for multiple methods. This technique is called *method overloading*. It is useful when you need to perform similar methods on different types of data.

The compiler must still be able to associate each invocation with a specific method declaration. If the method name for two or more methods is the same, then additional information is used to uniquely identify the version that is being invoked. In Java, a method name can be used for multiple methods as long as the number of parameters, the types of those parameters, or the order of the types of parameters is distinct. A method's name, along with the number, type, and order of its parameters, is called the method's *signature*. The compiler uses the complete method signature to *bind* a method invocation to the appropriate definition.

KEY CONCEPT

The versions of an overloaded method are distinguished by their signatures. The number, type, or order of their parameters must be distinct.

The compiler must be able to examine a method invocation, including the parameter list, to determine which specific method is being invoked. If you attempt to specify two method names with the same signature, the compiler will issue an appropriate error message and will not create an executable program. There can be no ambiguity.

Note that the return type of a method is not part of the method signature. That is, two overloaded methods cannot differ only by their return type. The reason is that the value returned by a method can be ignored by the invocation. The compiler would not be able to distinguish which version of an overloaded method is being referenced in such situations.

The `println` method is an example of a method that is overloaded several times, each accepting a single type. Here is a partial list of its various signatures:

```
println(String s)
println(int i)
println(double d)
println(char c)
println(boolean b)
```

The following two lines of code actually invoke different methods that have the same name.

```
System.out.println ("The total is: ");
System.out.println (count);
```

The first line invokes the `println` that accepts a string, and the second line, assuming `count` is an integer variable, invokes the version of `println` that accepts an integer. We often use a `println` statement that prints several distinct types, such as

```
System.out.println("The total is: " + count);
```

In this case, the plus sign is the string concatenation operator. First, the value in the variable `count` is converted into a string representation. Then the two strings are concatenated into one longer string, and the definition of `println` that accepts a single string is invoked.

Constructors are primary candidates for overloading. By providing multiple versions of a constructor, we provide several ways to set up an object.

B.9 References Revisited

In previous examples, we have declared *object reference variables* through which we access particular objects. Let's examine this relationship in more detail.

KEY CONCEPT

An object reference variable stores the address of an object.

An object reference variable and an object are two separate things. Remember that the declaration of the reference variable and the creation of the object that it refers to are separate steps. We often declare the reference variable and create an object for it to refer to on the same line, but keep in mind that we don't have to do so. In fact, in many cases, we won't want to.

The reference variable holds the address of an object even though the address never is disclosed to us. When we use the dot operator to invoke an object's method, we are actually using the address in the reference variable to locate the representation of the object in memory, look up the appropriate method, and invoke it.

The Null Reference

A reference variable that does not currently point to an object is called a *null reference*. When a reference variable is initially declared as an instance variable, it is a null reference. If we try to follow a null reference, a `NullPointerException` is thrown, indicating that there is no object to reference. For example, consider the following situation:

```
class NameIsNull
{
    String name; // not initialized, therefore null

    void printName()
    {
        System.out.println(name.length()); // causes an exception
    }
}
```

The declaration of the instance variable `name` asserts it to be a reference to a `String` object but does not create any `String` object for it to refer to. The variable `name`, therefore, contains a null reference. When the method attempts to invoke the `length` method of the object to which `name` refers, an exception is thrown because no object exists to execute the method.

Note that this situation can arise only in the case of instance variables. Suppose, for instance, that the following two lines of code were in a method:

```
String name;
System.out.println(name.length());
```

In this case, the variable `name` is local to whatever method it is declared in. The compiler would complain that we were using the `name` variable before it had been initialized. In the case of instance variables, however, the compiler can't determine whether a variable had been initialized or not. Therefore, the danger of attempting to follow a null reference is a problem.

The identifier `null` is a reserved word in Java and represents a null reference. We can explicitly set a reference to `null` to ensure that it doesn't point to any object. We can also use it to check whether a particular reference currently points to an object. For example, we could have used the following code in the `printName` method to keep us from following a null reference:

```
if (name == null)
    System.out.println("Invalid Name");
else
    System.out.println(name.length());
```

KEY CONCEPT

The reserved word `null` represents a reference that does not point to a valid object.

The `this` Reference

Another special reference for Java objects is called the `this` reference. The word `this` is a reserved word in Java. It allows an object to refer to itself. As we have discussed, a method is always invoked through a particular object or class. Inside that method, the `this` reference can be used to refer to the currently executing object.

KEY CONCEPT

The `this` reference always refers to the currently executing object.

For example, in the `ChessPiece` class, there could be a method called `move`, which could contain the line

```
if (this.position == piece2.position)
    result = false;
```

In this situation, the `this` reference is being used to clarify which position is being referenced. The `this` reference refers to the object through which the method was invoked. Therefore, when the following line is used to invoke the method, the `this` reference refers to `bishop1`:

```
bishop1.move();
```

But when another object is used to invoke the method, the `this` reference refers to it. Therefore, when the following invocation is used, the `this` reference in the `move` method refers to `bishop2`:

```
bishop2.move();
```

The `this` reference can also be used to distinguish the parameters of a constructor from their corresponding instance variables with the same names. For example, the constructor of a class called `Account` could be defined as follows:

```
public Account(String owner, long account, double initial)
{
    name = owner;
    acctNumber = account;
    balance = initial;
}
```

In this constructor, we deliberately came up with different names for the parameters to distinguish them from the instance variables `name`, `acctNumber`, and `balance`. This distinction is arbitrary. The constructor could have been written as follows using the `this` reference:

```
public Account(String name, long acctNumber, double balance)
{
    this.name = name;
    this.acctNumber = acctNumber;
    this.balance = balance;
}
```

In this version of the constructor, the `this` reference specifically refers to the instance variables of the object. The variables on the right-hand side of the assignment statements refer to the formal parameters. This approach eliminates the need to come up with different yet equivalent names. This situation sometimes occurs in other methods, but it comes up often in constructors.

Aliases

Because an object reference variable stores an address, programmers must be careful when managing objects. In particular, the semantics of an assignment statement for objects must be carefully understood. First, let's review the concept of assignment for primitive types. Consider the following declarations of primitive data:

```
int num1 = 5;
int num2 = 12;
```

In the following assignment statement, a copy of the value that is stored in `num1` is stored in `num2`:

```
num2 = num1;
```

The original value of 12 in `num2` is overwritten by the value 5. The variables `num1` and `num2` still refer to different locations in memory, and both of those locations now contain the value 5.

Now consider the following object declarations:

```
ChessPiece bishop1 = new ChessPiece();
ChessPiece bishop2 = new ChessPiece();
```

Initially, the references `bishop1` and `bishop2` refer to two different `ChessPiece` objects. The following assignment statement copies the value in `bishop1` into `bishop2`:

```
bishop2 = bishop1;
```

The key issue is that when an assignment like this is made, the address stored in `bishop1` is copied into `bishop2`. Originally the two references referred to different objects. After the assignment, both `bishop1` and `bishop2` contain the same address and therefore refer to the same object.

The `bishop1` and `bishop2` references are now *aliases* of each other, because they are two names that refer to the same object. All references to the object that was originally referenced by `bishop2` are now gone; that object cannot be used again in the program.

One important implication of aliases is that when we use one reference to change the state of the object, it is also changed for the other, because there is really only one object. If you change the state of `bishop1`, for instance, you change the state of `bishop2`, because they both refer to the same object. Aliases can produce undesirable effects unless they are managed carefully.

Another important aspect of references is the way they affect how we determine whether two objects are equal. The `==` operator that we use for primitive data can be used with object references, but it returns `true` only if the two

KEY CONCEPT

Several references can refer to the same object. These references are aliases of each other.

KEY CONCEPT

The `==` operator compares object references for equality, returning true if the references are aliases of each other.

references being compared are aliases of each other. It does not “look inside” the objects to see whether they contain the same data.

That is, the following expression is true only if `bishop1` and `bishop2` currently refer to the same object:

```
bishop1 == bishop2
```

A method called `equals` is defined for all objects, but unless we replace it with a specific definition when we write a class, it has the same semantics as the `==` operator. That is, the `equals` method returns a `boolean` value that, by default, will be true if the two objects being compared are aliases of each other. The `equals` method is invoked through one object and takes the other one as a parameter. Therefore, the expression

```
bishop1.equals(bishop2)
```

KEY CONCEPT

The `equals` method can be defined to determine equality between objects in any way we consider appropriate.

returns true if both references refer to the same object. However, we can define the `equals` method in the `ChessPiece` class to define equality for `ChessPiece` objects in any way we like. That is, we can define the `equals` method to return true under whatever conditions we think are appropriate to mean that one `ChessPiece` is equal to another.

The `equals` method has been given an appropriate definition in the `String` class. When comparing two `String` objects, the `equals` method returns true only if both strings contain the same characters. A common mistake is to use the `==` operator to compare strings, which compares the references for equality, when most of the time we want to compare the characters in the strings for equality. The `equals` method is discussed in more detail later in this appendix.

Garbage Collection

All interaction with an object occurs through a reference variable, so we can use an object only if we have a reference to it. When all references to an object are lost (perhaps by reassignment), that object can no longer participate in the program. The program can no longer invoke its methods or use its variables. At this point the object is called *garbage*, because it serves no useful purpose.

KEY CONCEPT

If an object has no references to it, a program cannot use it. Java performs automatic garbage collection by periodically reclaiming the memory space occupied by these objects.

Java performs *automatic garbage collection*. When the last reference to an object is lost, the object becomes a candidate for garbage collection. Occasionally, the Java run-time executes a method that “collects” all of the objects marked for garbage collection and returns their allocated memory to the system for future use. The programmer does not have to worry about explicitly returning memory that has become garbage.

If there is an activity that a programmer wants to accomplish in conjunction with the object being destroyed, the programmer can define a method called `finalize` in the object's class. The `finalize` method takes no parameters and has a `void` return type. It will be executed by the Java run time after the object is marked for garbage collection and before it is actually destroyed. The `finalize` method is not often used, because the garbage collector performs most normal cleanup operations. However, it is useful for performing activities that the garbage collector does not address, such as closing files.

Passing Objects as Parameters

Another important issue related to object references comes up when we want to pass an object to a method. Java passes all parameters to a method *by value*. That is, the current value of the actual parameter (in the invocation) is copied into the formal parameter in the method header. Essentially, parameter passing is like an assignment statement, assigning to the formal parameter a copy of the value stored in the actual parameter.

This issue must be considered when making changes to a formal parameter inside a method. The formal parameter is a separate copy of the value that is passed in, so any changes made to it have no effect on the actual parameter. After control returns to the calling method, the actual parameter will have the same value it had before the method was called.

However, when we pass an object to a method, we are actually passing a reference to that object. The value that gets copied is the address of the object. Therefore, the formal parameter and the actual parameter become aliases of each other. If we change the state of the object through the formal parameter reference inside the method, we are changing the object referenced by the actual parameter, because they refer to the same object. On the other hand, if we change the formal parameter reference itself (to make it point to a new object, for instance), we have not changed the fact that the actual parameter still refers to the original object.

KEY CONCEPT

When an object is passed to a method, the actual and formal parameters become aliases of each other.

B.10 The `static` Modifier

We have seen how visibility modifiers enable us to specify the encapsulation characteristics of variables and methods in a class. Java has several other modifiers that determine other characteristics. For example, the `static` modifier associates a variable or method with its class rather than with an object of the class.

Static Variables

So far, we have seen two categories of variables: local variables, which are declared inside a method, and instance variables, which are declared in a class but not inside a method. The term *instance variable* is used because an instance variable is accessed through a particular instance (an object) of a class. In general, each object has distinct memory space for each variable, so that each object can have a distinct value for that variable.

Another kind of variable, called a *static variable* or *class variable*, is shared among all instances of a class. There is only one copy of a static variable for all objects of a class. Therefore, changing the value of a static variable in one object changes it for all of the others. The reserved word `static` is used as a modifier to declare a static variable:

```
private static int count = 0;
```

KEY CONCEPT

A static variable is shared among all instances of a class.

Memory space for a static variable is established when the class that contains it is referenced for the first time in a program. A local variable declared within a method cannot be static.

Constants, which are declared using the `final` modifier, are also often declared using the `static` modifier as well. Because the value of constants cannot be changed, there might as well be only one copy of the value across all objects of the class.

Static Methods

A *static method* (also called a *class method*) can be invoked through the class name (all the methods of the `Math` class are static methods, for example). You don't have to instantiate an object of the class to invoke a static method. For example, the `sqrt` method is called through the `Math` class as follows:

```
System.out.println("Square root of 27: " + Math.sqrt(27));
```

KEY CONCEPT

A method is made static by using the `static` modifier in the method declaration.

A method is made static by using the `static` modifier in the method declaration. As we have seen, the `main` method of a Java program must be declared with the `static` modifier; this is so that `main` can be executed by the interpreter without instantiating an object from the class that contains `main`.

Because static methods do not operate in the context of a particular object, they cannot reference instance variables, which exist only in an instance of a class. The compiler will issue an error if a static method attempts to use a nonstatic variable. A static method can, however, reference static variables, because static variables exist independent of specific objects. Therefore, the `main` method can access only static or local variables.

The methods in the `Math` class perform basic computations based on values passed as parameters. There is no object state to maintain in these situations; therefore, there is no good reason to force us to create an object in order to request these services.

B.11 Wrapper Classes

In some object-oriented programming languages, everything is represented using classes and the objects that are instantiated from them. In Java there are primitive types (such as `int`, `double`, `char`, and `boolean`) in addition to classes and objects.

Having two categories of data to manage (primitive values and object references) can present a challenge in some circumstances. For example, we might create an object that serves as a collection to hold various types of other objects. But in a specific situation, we want the collection to hold simple integer values. In these cases we need to “wrap” a primitive type into a class so that it can be treated as an object.

A *wrapper class* represents a particular primitive type. For instance, the `Integer` class represents a simple integer value. An object created from the `Integer` class stores a single `int` value. The constructors of the wrapper classes accept the primitive value to store—for example,

```
Integer ageObj = new Integer(45);
```

Once this declaration and instantiation are performed, the `ageObj` object effectively represents the integer 45 as an object. It can be used wherever an object is called for in a program instead of a primitive type.

For each primitive type in Java there exists a corresponding wrapper class in the Java class library. All wrapper classes are defined in the `java.lang` package. There is even a wrapper class that represents the type `void`. However, unlike the other wrapper classes, the `Void` class cannot be instantiated. It simply represents the concept of a void reference.

The wrapper classes also provide various methods related to the management of the associated primitive type. For example, the `Integer` class contains methods that return the `int` value stored in the object, and that convert the stored value into other primitive types.

Wrapper classes also contain static methods that can be invoked independent of any instantiated object. For example, the `Integer` class contains a static method called `parseInt` to convert an integer that is stored in a `String` into its corresponding `int` value. If the `String` object `str` holds the string “987”, then the following line of code converts and stores the integer value 987 into the `int` variable `num`:

```
num = Integer.parseInt(str);
```

KEY CONCEPT

A wrapper class represents a primitive value so that it can be treated as an object.

The Java wrapper classes often contain static constants that are helpful as well. For example, the `Integer` class contains two constants, `MIN_VALUE` and `MAX_VALUE`, which hold the smallest and largest `int` values, respectively. The other wrapper classes contain similar constants for their types.

B.12 Interfaces

We have used the term *interface* to mean the public methods through which we can interact with an object. That definition is consistent with our use of it in this section, but now we are going to formalize this concept using a particular language construct in Java.

KEY CONCEPT

An interface is a collection of abstract methods. It cannot be instantiated.

A Java *interface* is a collection of constants and abstract methods. An *abstract method* is a method that does not have an implementation. That is, there is no body of code defined for an abstract method. The header of the method, including its parameter list, is simply followed by a semicolon. An interface cannot be instantiated.

The following interface, which is called `Complexity`, contains two abstract methods, `setComplexity` and `getComplexity`:

```
interface Complexity
{
    void setComplexity(int complexity);
    int getComplexity();
}
```

An abstract method can be preceded by the reserved word `abstract`, although in interfaces it usually is not. Methods in interfaces have public visibility by default.

KEY CONCEPT

A class implements an interface, which formally defines a set of methods used to interact with objects of that class.

A class *implements* an interface by providing method implementations for each of the abstract methods defined in the interface. A class that implements an interface uses the reserved word `implements` followed by the interface name in the class header. If a class asserts that it implements a particular interface, it must provide a definition for all methods in the interface. The compiler will produce errors if any of the methods in the interface is not given a definition in the class.

For example, a class called `Question` could be defined to represent a question that a teacher may ask on a test. If the `Question` class implements the `Complexity` interface, it must explicitly say so in the header and must define both methods from the `Complexity` interface:

```
class Questions implements Complexity
{
    int difficulty;
    // whatever else
```

```
void setComplexity(int complexity)
{
    difficulty = complexity;
}

int getComplexity()
{
    return difficulty;
}
}
```

Multiple classes can implement the same interface, providing alternative definitions for the methods. For example, we could implement a class called `Task` that also implemented the `Complexity` interface. In it we could choose to manage the complexity of a task in a different way (though it would still have to implement all the methods of the interface).

A class can implement more than one interface. In these cases, the class must provide an implementation for all methods in all interfaces listed. To show that a class implements multiple interfaces, they are listed in the `implements` clause, separated by commas. Here is an example:

```
class ManyThings implements Interface1, Interface2, Interface3
{
    // all methods of all interfaces
}
```

In addition to, or instead of, abstract methods, an interface can also contain constants, defined using the `final` modifier. When a class implements an interface, it gains access to all of the constants defined in it. This mechanism allows multiple classes to share a set of constants that are defined in a single location.

The Comparable Interface

The Java standard class library contains interfaces as well as classes. The `Comparable` interface, for example, is defined in the `java.lang` package. It contains only one method, `compareTo`, which takes an object as a parameter and returns an integer.

The intention of this interface is to provide a common mechanism for comparing one object to another. One object calls the method and passes another as a parameter:

```
if (obj1.compareTo(obj2) < 0)
    System.out.println("obj1 is less than obj2");
```

As specified by the documentation for the interface, the integer that is returned from the `compareTo` method should be negative if `obj1` is less than `obj2`, 0 if they are equal, and positive if `obj1` is greater than `obj2`. It is up to the designer of each class to decide what it means for one object of that class to be less than, equal to, or greater than another.

The `String` class contains a `compareTo` method that operates in this manner. Now we can clarify that the `String` class has this method because it implements the `Comparable` interface. The `String` class implementation of this method bases the comparison on the lexicographic ordering defined by the Unicode character set.

B.13 Inheritance

A class establishes the characteristics and behaviors of an object, but it reserves no memory space for variables (unless those variables are declared as static). Classes are the plan, and objects are the embodiment of that plan.

Many houses can be created from the same blueprint. They are essentially the same house in different locations with different people living in them. But suppose you want a house that is similar to another but has some different or additional features. You want to start with the same basic blueprint and then modify it to suit your needs and desires. Many housing developments are created this way. The houses in the development have the same core layout, but each house can be “customized” to include some unique features. For instance, they may all be split-level homes with the same bedroom, kitchen, and living room configuration, but some have a fireplace or full basement whereas others do not, and some have an attached garage instead of a carport.

It’s likely that the housing developer commissioned a master architect to create a single blueprint to establish the basic design of all houses in the development and then add a series of new blueprints that include variations designed to appeal to different buyers. The act of creating the series of blueprints was simplified because they all begin with the same underlying structure, while the variations give them unique characteristics that may be very important to the prospective owners.

Creating a new blueprint that is based on an existing blueprint is analogous to the object-oriented concept of *inheritance*, which allows a software designer to define a new class in terms of an existing one. It is a powerful software development technique and a defining characteristic of object-oriented programming.

Derived Classes

Inheritance is the process in which a new class is derived from an existing one. The new class automatically contains some or all of the variables and methods in the original class. Then, to tailor the class as needed, the programmer can add new variables and methods to the derived class, or modify the inherited ones.

KEY CONCEPT

Inheritance is the process of deriving a new class from an existing one.

In general, creating new classes via inheritance is faster, easier, and cheaper than writing them from scratch. At the heart of inheritance is the idea of *software reuse*. By using existing software components to create new ones, we capitalize on all of the effort that went into the design, implementation, and testing of the existing software.

Keep in mind that the word *class* comes from the idea of classifying groups of objects with similar characteristics. Classification schemes often use levels of classes that are related to one another. For example, all mammals share certain characteristics; they are warm-blooded, have hair, and bear live offspring. Now consider a subset of mammals, such as horses. All horses are mammals and have all the characteristics of mammals. But they also have unique features that make them different from other mammals.

If we map this idea into software terms, an existing class called `Mammal` will have certain variables and methods that describe the state and behavior of mammals. A `Horse` class can be derived from the existing `Mammal` class, automatically inheriting the variables and methods contained in `Mammal`. The `Horse` class can refer to the inherited variables and methods as if they had been declared locally in that class. New variables and methods can then be added to the derived class, to distinguish a horse from other mammals. Inheritance nicely models many situations found in the natural world.

The original class that is used to derive a new one is called the *parent class*, *superclass*, or *base class*. The derived class is called a *child class* or *subclass*. Java uses the reserved word `extends` to indicate that a new class is being derived from an existing class.

The derivation process should establish a specific kind of relationship between two classes: an *is-a relationship*. This type of relationship means that the derived class should be a more specific version of the original. For example, a horse is a mammal. Not all mammals are horses, but all horses are mammals.

Let's look at an example. The following class can be used to define a book:

```
class Book
{
    protected int numPages;

    protected void pages()
    {
        System.out.println("Number of pages: " + numPages);
    }
}
```

KEY CONCEPT

One purpose of inheritance is to reuse existing software.

KEY CONCEPT

Inherited variables and methods can be used in the derived class as if they had been declared locally.

KEY CONCEPT

Inheritance creates an is-a relationship between all parent and child classes.

To derive a child class that is based on the `Book` class, we use the reserved word `extends` in the header of the child class. For example, a `Dictionary` class can be derived from `Book` as follows:

```
class Dictionary extends Book
{
    private int numDefs;

    public void info()
    {
        System.out.println("Number of definitions: " + numDefs);
        System.out.println("Definitions per page: "
            + numDefs/numPages);
    }
}
```

Saying that the `Dictionary` class extends the `Book` class means that the `Dictionary` class automatically inherits the `numPages` variable and the `pages` method. Note that the `info` method uses the `numPages` variable explicitly.

Inheritance is a one-way street. The `Book` class cannot use variables or methods that are declared explicitly in the `Dictionary` class. For instance, if we created an object from the `Book` class, it could not be used to invoke the `info` method. This restriction makes sense, because a child class is a more specific version of the parent. A dictionary has pages, because all books have pages, but even though a dictionary has definitions, not all books do.

Inheritance relationships are represented in UML class diagrams using an arrow with an open arrowhead pointing from the child class to the parent class.

The protected Modifier

Not all variables and methods are inherited in a derivation. The visibility modifiers used to declare the members of a class determine which ones are inherited and which are not. Specifically, the child class inherits variables and methods that are declared `public`, and it does not inherit those that are declared `private`.

However, if we declare a variable with `public` visibility so that a derived class can inherit it, we violate the principle of encapsulation. Therefore, Java provides a third visibility modifier: `protected`. When a variable or method is declared with `protected` visibility, a derived class will inherit it, retaining some of its encapsulation properties. The encapsulation with `protected` visibility is not as tight as it would be if the variable or method were declared `private`, but it is better than if it were declared `public`. Specifically, a variable or method declared with `protected` visibility may be accessed by any class in the same package.

Each inherited variable or method retains the effect of its original visibility modifier. For example, if a method is public in the parent, it is public in the child.

Constructors are not inherited in a derived class, even though they have public visibility. This is an exception to the rule about public members being inherited. Constructors are special methods that are used to set up a particular type of object, so it wouldn't make sense for a class called `Dictionary` to have a constructor called `Book`.

KEY CONCEPT

Visibility modifiers determine which variables and methods are inherited. Protected visibility provides the best possible encapsulation that permits inheritance.

The `super` Reference

The reserved word `super` can be used in a class to refer to its parent class. Using the `super` reference, we can access a parent's members, even if they aren't inherited. Like the `this` reference, what the word `super` refers to depends on the class in which it is used. However, unlike the `this` reference, which refers to a particular instance of a class, `super` is a general reference to the members of the parent class.

One use of the `super` reference is to invoke a parent's constructor. If the following invocation is performed at the beginning of a constructor, the parent's constructor is invoked, passing any appropriate parameters:

```
super(x, y, z);
```

A child's constructor is responsible for calling its parent's constructor. Generally, the first line of a constructor should use the `super` reference call to a constructor of the parent class. If no such call exists, Java will automatically make a call to `super()` at the beginning of the constructor. This rule ensures that a parent class initializes its variables before the child class constructor begins to execute. Using the `super` reference to invoke a parent's constructor can be done only in the child's constructor and, if included, must be the first line of the constructor.

The `super` reference can also be used to reference other variables and methods defined in the parent's class.

KEY CONCEPT

A parent's constructor can be invoked using the `super` reference.

Overriding Methods

When a child class defines a method with the same name and signature as a method in the parent, we say that the child's version *overrides* the parent's version in favor of its own. The need for overriding occurs often in inheritance situations.

The object that is used to invoke a method determines which version of the method is actually executed. If it is an object of the parent

KEY CONCEPT

A child class can override (redefine) the parent's definition of an inherited method.

type, the parent's version of the method is invoked. If it is an object of the child type, the child's version is invoked. This flexibility allows two objects that are related by inheritance to use the same naming conventions for methods that accomplish the same general task in different ways.

A method can be defined with the `final` modifier. A child class cannot override a final method. This technique is used to ensure that a derived class uses a particular definition for a method.

The concept of method overriding is important to several issues related to inheritance. These issues are explored in later sections of this appendix.

B.14 Class Hierarchies

A child class derived from one parent can be the parent of its own child class. Furthermore, multiple classes can be derived from a single parent. Therefore, inheritance relationships often develop into *class hierarchies*. The UML class diagram in Figure B.4 shows a class hierarchy that incorporates the inheritance relationship between the classes `Mammal` and `Horse`.

There is no limit to the number of children a class can have, or to the number of levels to which a class hierarchy can extend. Two children of the same parent are called *siblings*. Although siblings share the characteristics passed on by their common parent, they are not related by inheritance, because one is not used to derive the other.

KEY CONCEPT

The child of one class can be the parent of one or more other classes, creating a class hierarchy.

KEY CONCEPT

Common features should be located as high in a class hierarchy as is reasonable, in order to minimize maintenance efforts.

In class hierarchies, common features should be kept as high in the hierarchy as reasonably possible. That way, the only characteristics explicitly established in a child class are those that make the class distinct from its parent and from its siblings. This approach maximizes the ability to reuse classes. It also facilitates maintenance activities, because when changes are made to the parent, they are automatically reflected in the descendants. Always remember to maintain the is-a relationship when building class hierarchies.

The inheritance mechanism is transitive. That is, a parent passes along a trait to a child class, that child class passes the trait along to its children, and so on. An inherited feature might have originated in the immediate parent, or possibly from several levels higher in a more distant ancestor class.

There is no single best hierarchy organization for all situations. The decisions made when designing a class hierarchy restrict and guide more detailed design decisions and implementation options, and they must be made carefully.

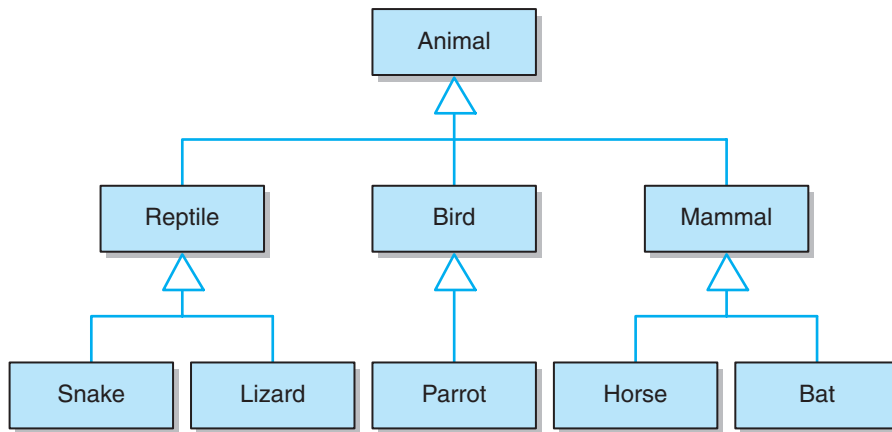


FIGURE B.4 A UML class diagram showing a class hierarchy

The Object Class

In Java, all classes are derived ultimately from the `Object` class. If a class definition doesn't use the `extends` clause to derive itself explicitly from another class, then that class is automatically derived from the `Object` class by default. Therefore, the following two class definitions are equivalent:

```
class Thing
{
    // whatever
}
```

and

```
class Thing extends Object
{
    // whatever
}
```

Because all classes are derived from `Object`, any public method of `Object` can be invoked through any object created in any Java program. The `Object` class is defined in the `java.lang` package of the standard class library.

The `toString` method, for instance, is defined in the `Object` class, so the `toString` method can be called on any object. When a `println` method is called with an object parameter, `toString` is called to determine what to print.

KEY CONCEPT

All Java classes are derived, directly or indirectly, from the `Object` class.

KEY CONCEPT

The `toString` and `equals` methods are defined in the `Object` class and therefore are inherited by every class in every Java program.

The definition for `toString` that is provided by the `Object` class returns a string containing the object's class name followed by a numeric value that is unique for that object. Usually, we override the `Object` version of `toString` to fit our own needs. The `String` class has overridden the `toString` method so that it returns its stored string value.

The `equals` method of the `Object` class is also useful. Its purpose is to determine whether two objects are equal. The definition of the `equals` method provided by the `Object` class returns `true` if the two object references actually refer to the same object (that is, if they are aliases). Classes often override the inherited definition of the `equals` method in favor of a more appropriate definition. For instance, the

`String` class overrides `equals` so that it returns `true` only if both strings contain the same characters in the same order.

Abstract Classes

An *abstract class* represents a generic concept in a class hierarchy. An abstract class cannot be instantiated and usually contains one or more abstract methods, which have no definition. In this sense, an abstract class is similar to an interface. Unlike interfaces, however, an abstract class can contain methods that are not abstract and can contain data declarations other than constants.

A class is declared as abstract by including the `abstract` modifier in the class header. Any class that contains one or more abstract methods must be declared as abstract. In abstract classes (unlike interfaces), the `abstract` modifier must be applied to each abstract method. A class declared as abstract does not have to contain abstract methods.

Abstract classes serve as placeholders in a class hierarchy. As the name implies, an abstract class represents an abstract entity that is usually not sufficiently defined to be useful by itself. Instead, an abstract class may contain a partial description that is inherited by all of its descendants in the class hierarchy. Its children, which are more specific, fill in the gaps.

KEY CONCEPT

An abstract class cannot be instantiated. It represents a concept on which other classes can build their definitions.

Consider the class hierarchy shown in Figure B.5. The `Vehicle` class at the top of the hierarchy may be too generic for a particular application. Therefore, we may choose to implement it as an abstract class. Concepts that apply to all vehicles can be represented in the `Vehicle` class and are inherited by its descendants. That way, each of its descendants doesn't have to define the same concept redundantly, and perhaps inconsistently.

For example, we may say that all vehicles have a particular speed. Therefore, we declare a `speed` variable in the `Vehicle` class, and all specific vehicles below it

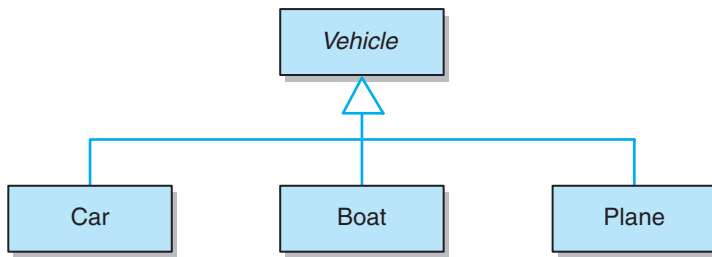


FIGURE B.5 A `Vehicle` class hierarchy

in the hierarchy automatically have that variable via inheritance. Any change we make to the representation of the speed of a vehicle is automatically reflected in all descendant classes. Similarly, we may declare an abstract method called `fuelConsumption`, whose purpose is to calculate how quickly fuel is being consumed by a particular vehicle. The details of the `fuelConsumption` method must be defined by each type of vehicle, but the `Vehicle` class establishes that all vehicles consume fuel and provides a consistent way to compute that value.

Some concepts don't apply to all vehicles, so we wouldn't represent those concepts at the `Vehicle` level. For instance, we wouldn't include a variable called `numberOfWheels` in the `Vehicle` class, because not all vehicles have wheels. The child classes for which wheels are appropriate can add that concept at the appropriate level in the hierarchy.

There are no restrictions on where in a class hierarchy an abstract class can be defined. Abstract classes are usually located at the upper levels of a class hierarchy. However, it is possible to derive an abstract class from a nonabstract parent.

Usually, a child of an abstract class will provide a specific definition for an abstract method inherited from its parent. Note that this is just a specific case of overriding a method, giving a different definition from the one the parent provides. If a child of an abstract class does not give a definition for every abstract method that it inherits from its parent, then the child class is also considered to be abstract.

Note that it would be a contradiction for an abstract method to be modified as `final` or `static`. Because a `final` method cannot be overridden in subclasses, an abstract `final` method would have no way of being given a definition in subclasses. A `static` method can be invoked using the class name without declaring an object of the class. Because abstract methods have no implementation, an abstract `static` method would make no sense.

Choosing which classes and methods to make abstract is an important part of the design process. Such choices should be made only after careful consideration. By using abstract classes wisely, we can create flexible, extensible software designs.

KEY CONCEPT

A class derived from an abstract parent must override all of its parent's abstract methods, or the derived class will also be considered abstract.

Interface Hierarchies

The concept of inheritance can be applied to interfaces as well as to classes. That is, one interface can be derived from another interface. These relationships can form an *interface hierarchy*, which is similar to a class hierarchy. Inheritance relationships between interfaces are shown in UML using the same connection (an arrow with an open arrowhead) that is used with classes.

KEY CONCEPT

Inheritance can be applied to interfaces, so that one interface can be derived from another interface.

When a parent interface is used to derive a child interface, the child inherits all abstract methods and constants of the parent. Any class that implements the child interface must implement all of the methods. There are no restrictions on the inheritance between interfaces, as there are with protected and private members of a class, because all members of an interface are public.

Class hierarchies and interface hierarchies do not overlap. That is, an interface cannot be used to derive a class, and a class cannot be used to derive an interface. A class and an interface interact only when a class is designed to implement a particular interface.

B.15 Polymorphism

Usually, the type of a reference variable exactly matches the class of the object it refers to. That is, if we declare a reference as follows:

```
ChessPiece bishop;
```

the `bishop` reference is used to refer to an object created by instantiating the `ChessPiece` class. However, the relationship between a reference variable and the object it refers to is more flexible than that.

The term *polymorphism* can be defined as “having many forms.” A *polymorphic reference* is a reference variable that can refer to different types of objects at different points in time. The specific method invoked through a polymorphic reference can change from one invocation to the next.

Consider the following line of code:

```
obj.doIt();
```

If the reference `obj` is polymorphic, it can refer to different types of objects at different times. If that line of code is in a loop or in a method that is called more than once, that line of code might call a different version of the `doIt` method each time it is invoked.

KEY CONCEPT

A polymorphic reference can refer to different types of objects over time.

At some point, the commitment is made to execute certain code to carry out a method invocation. This commitment is

referred to as *binding* a method invocation to a method definition. In most situations, the binding of a method invocation to a method definition can occur at compile time. For polymorphic references, however, the decision cannot be made until run-time. The method definition that is used is based on the object that is being referred to by the reference variable at that moment. This deferred commitment is called *late binding* or *dynamic binding*. It is less efficient than binding at compile time, because the decision has to be made during the execution of the program. This overhead is generally acceptable in light of the flexibility that a polymorphic reference provides.

There are two ways to create a polymorphic reference in Java: using inheritance and using interfaces. The following sections describe these approaches.

References and Class Hierarchies

In Java, a reference that is declared to refer to an object of a particular class also can be used to refer to an object of any class related to it by inheritance. For example, if the class `Mammal` is used to derive the class `Horse`, then a `Mammal` reference can be used to refer to an object of class `Horse`. This ability is shown in the code segment

```
Mammal pet;  
Horse secretariat = new Horse();  
pet = secretariat; // a valid assignment
```

The reverse operation, assigning the `Mammal` object to a `Horse` reference, is also valid, but it requires an explicit cast. Assigning a reference in this direction is generally less useful and more likely to cause problems, because although a horse has all the functionality of a mammal (because a horse *is-a* mammal), the reverse is not necessarily true.

This relationship works throughout a class hierarchy. If the `Mammal` class were derived from a class called `Animal`, then the following assignment would also be valid:

```
Animal creature = new Horse();
```

Carrying this to the extreme, an `Object` reference can be used to refer to any object, because ultimately all classes are descendants of the `Object` class. An `ArrayList`, for example, uses polymorphism in that it is designed to hold `Object` references. That's why an `ArrayList` can be used to store any kind of object. In fact, a particular `ArrayList` can be used to hold several different types of objects at one time, because, in essence, they are all `Object` objects.

KEY CONCEPT

A reference variable can refer to any object created from any class related to it by inheritance.

Polymorphism via Inheritance

The reference variable `creature`, as defined in the previous section, can be polymorphic, because at any point in time it could refer to an `Animal` object, a `Mammal` object, or a `Horse` object. Suppose that all three of these classes have a method called `move` and that it is implemented in a different way in each class (because the child class overrode the definition it inherited). The invocation

```
creature.move();
```

calls the `move` method, but the particular version of the method it calls is determined at run-time.

At the point when this line is executed, if `creature` currently refers to an `Animal` object, the `move` method of the `Animal` class is invoked. Likewise, if `creature` currently refers to a `Mammal` or `Horse` object, the `Mammal` or `Horse` version of `move` is invoked, respectively.

KEY CONCEPT

A polymorphic reference uses the type of the object, not the type of the reference, to determine which version of a method to invoke.

Of course, because `Animal` and `Mammal` represent general concepts, they may be defined as abstract classes. This situation does not eliminate the ability to have polymorphic references. Suppose the `move` method in the `Mammal` class is abstract and is given unique definitions in the `Horse`, `Dog`, and `Whale` classes (all derived from `Mammal`). A `Mammal` reference variable can be used to refer to any objects created from any of the `Horse`, `Dog`, and `Whale` classes, and it can be used to execute the `move` method on any of them.

Let's consider another situation. The class hierarchy shown in Figure B.6 contains classes that represent various types of employees that might work at a particular company.

Polymorphism could be used in this situation to pay various employees in different ways. One list of employees (of whatever type) could be paid using a single loop that invokes each employee's `pay` method. But the `pay` method that is invoked each time will depend on the specific type of employee that is executing the `pay` method during that iteration of the loop.

This is a classic example of polymorphism—allowing different types of objects to handle a similar operation in different ways.

Polymorphism via Interfaces

As we have seen, a class name is used to declare the type of an object reference variable. Similarly, an interface name can be used as the type of a reference variable as well. An interface reference variable can be used to refer to any object of any class that implements that interface.

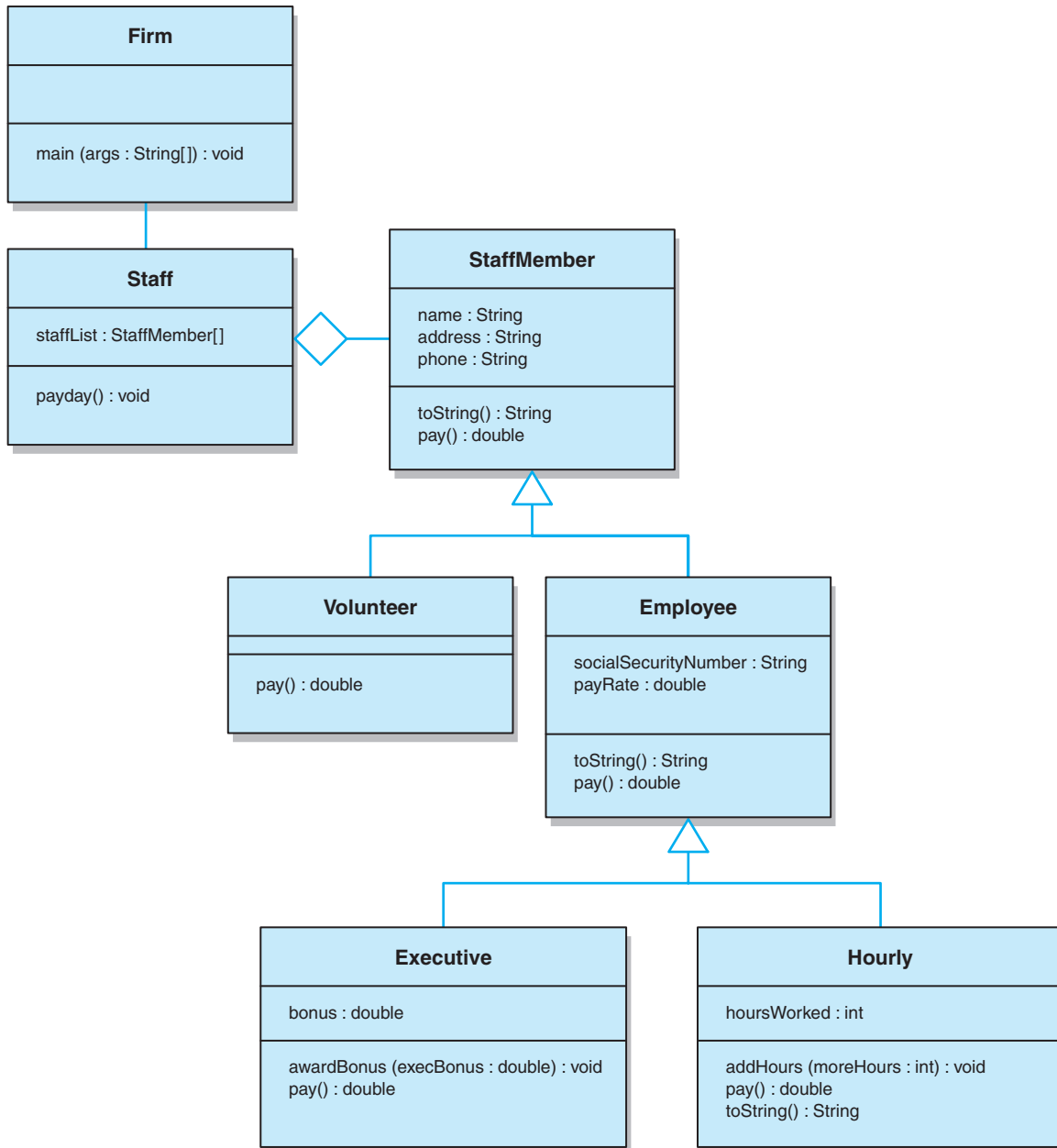


FIGURE B.6 A class hierarchy of employees

Suppose we declare an interface called `Speaker` as follows:

```
public interface Speaker
{
    public void speak();
    public void announce(String str);
}
```

KEY CONCEPT

An interface name can be used to declare an object reference variable. An interface reference can refer to any object of any class that implements the interface.

The interface name, `Speaker`, can now be used to declare an object reference variable:

```
Speaker current;
```

The reference variable `current` can be used to refer to any object of any class that implements the `Speaker` interface. For example, if we define a class called `Philosopher` such that it implements the `Speaker` interface, we can then assign a `Philosopher` object to a `Speaker` reference:

```
current = new Philosopher();
```

KEY CONCEPT

Interfaces enable us to make polymorphic references, in which the method that is invoked is based on the particular object being referenced at the time.

This assignment is valid because a `Philosopher` is, in fact, a `Speaker`.

The flexibility of an interface reference enables us to create polymorphic references. As we saw earlier in this appendix, by using inheritance, we can create a polymorphic reference that can refer to any one of a set of objects related by inheritance. Using interfaces, we can create similar polymorphic references, except that the objects being referenced are related by implementing the same interface instead of being related by inheritance.

For example, if we create a class called `Dog` that also implements the `Speaker` interface, it too can be assigned to a `Speaker` reference variable. The same reference, in fact, could at one point refer to a `Philosopher` object and then later refer to a `Dog` object. The following lines of code illustrate this:

```
Speaker guest;
guest = new Philosopher();
guest.speak();
guest = new Dog();
guest.speak();
```

In this code, the first time the `speak` method is called, it invokes the `speak` method defined in the `Philosopher` class. The second time it is called, it invokes the `speak` method of the `Dog` class. As with polymorphic references via inheritance, it is not the type of the reference that determines which method gets invoked, but rather it is the type of the object that the reference points to at the moment of invocation.

Note that when we are using an interface reference variable, we can invoke only the methods defined in the interface, even if the object it refers to has other methods to which it can respond. For example, suppose the `Philosopher` class also defined a public method called `pontificate`. The second line of the following code would generate a compiler error, even though the object can in fact respond to the `pontificate` method.

```
Speaker special = new Philosopher();
special.pontificate(); // generates a compiler error
```

The problem is that the compiler can determine only that the object is a `Speaker` and therefore can guarantee only that the object can respond to the `speak` and `announce` methods. Because the reference variable `special` could refer to a `Dog` object (which cannot `pontificate`), it does not allow the reference. If we know in a particular situation that such an invocation is valid, we can cast the object into the appropriate reference so that the compiler will accept it:

```
((Philosopher) special).pontificate();
```

Similar to polymorphic references based on inheritance, an interface name can be used as the type of a method parameter. In such situations, any object of any class that implements the interface can be passed into the method. For example, the following method takes a `Speaker` object as a parameter. Therefore, both a `Dog` object and a `Philosopher` object can be passed into it in separate invocations.

```
public void sayIt(Speaker current)
{
    current.speak();
}
```

B.16 Exceptions

Problems that arise in a Java program may generate exceptions or errors. An *exception* is an object that defines an unusual or erroneous situation. An exception is thrown by a program or the run-time environment, and it can be caught and handled appropriately if desired. An *error* is similar to an exception, except that an error generally represents an unrecoverable situation, and it should not be caught. Java has a predefined set of exceptions and errors that may occur during the execution of a program.

A program can be designed to process an exception in one of three ways:

- Not handle the exception at all.
- Handle the exception where it occurs.
- Handle the exception at another point in the program.

We explore each of these approaches in the following sections.

Exception Messages

If an exception is not handled at all by the program, the program will terminate (abnormally) and produce a message that describes what exception occurred and where in the program it was produced. The information associated with an exception is often helpful in tracking down the cause of a problem.

KEY CONCEPT

Errors and exceptions represent unusual or invalid processing.

Let's look at the output of an exception. An `ArithmeticException` is thrown when an invalid arithmetic operation, such as dividing by zero, is attempted. When that exception is thrown, if there is no code in the program to handle the exception explicitly, the program terminates and prints a message similar to the following:

```
Exception in thread "main" java.lang.ArithmeticException: / by
zero at Zero.main (Zero.java:17)
```

The first line of the exception output indicates which exception was thrown and provides some information about why it was thrown. The remaining line or lines are the *call stack trace*, which indicates where the exception occurred. In this case, there is only one line in the call stack trace, but in other cases there may be several, depending on where the exception originated in the program.

KEY CONCEPT

The messages printed by a thrown exception indicate the nature of the problem and provide a method call stack trace.

The first line of the trace indicates the method, file, and line number where the exception occurred. The other lines in the trace, if present, indicate the methods that were called to get to the method that produced the exception. In this program, there is only one method, and it produced the exception; therefore, there is only one line in the trace.

The call stack trace information can also be found by calling methods of the exception object that is being thrown. The method `getMessage` returns a string explaining why the exception was thrown. The method `printStackTrace` prints the call stack trace.

The try Statement

Let's now examine how we catch and handle an exception when it is thrown. A *try statement* consists of a `try` block followed by one or more `catch` clauses. The `try` block is a group of statements that may throw an exception. A `catch` clause defines how a particular kind of exception is handled. A `try` block can have several `catch` clauses associated with it, each dealing with a particular kind of exception. A `catch` clause is sometimes called an *exception handler*.

Here is the general format of a `try` statement:

```
try
{
    // statements in the try block
}
catch(IOException exception)
{
    // statements that handle the I/O problem
}
catch(NumberFormatException exception)
{
    // statements that handle the number format problem
}
```

When a `try` statement is executed, the statements in the `try` block are executed. If no exception is thrown during execution of the `try` block, processing continues with the statement following the `try` statement (after all of the `catch` clauses). This situation is the normal execution flow and should occur most of the time.

If an exception is thrown at any point during execution of the `try` block, control is immediately transferred to the appropriate exception handler if it is present. That is, control transfers to the first `catch` clause whose specified exception corresponds to the class of the exception that was thrown. After the statements in the `catch` clause are executed, control transfers to the statement after the entire `try` statement.

Exception Propagation

If an exception is not caught and handled where it occurs, control is immediately returned to the method that invoked the method that produced the exception. We can design our software so that the exception is caught and handled at this outer level. If it isn't caught there, control returns to the method that called it. This process is called *propagating the exception*.

Exception propagation continues until the exception is caught and handled, or until it is propagated out of the `main` method, which terminates the program and produces an exception message. To catch an exception at an outer level, the method that produces the exception must be invoked inside a `try` block that has an appropriate `catch` clause to handle it.

KEY CONCEPT

Each `catch` clause on a `try` statement handles a particular kind of exception that may be thrown within the `try` block.

KEY CONCEPT

If an exception is not caught and handled where it occurs, it is propagated to the calling method.

KEY CONCEPT

A programmer must carefully consider how exceptions should be handled, if at all, and at what level.

A programmer must pick the most appropriate level at which to catch and handle an exception. There is no single best answer. It depends on the situation and the design of the system. Sometimes the right approach will be not to catch an exception at all and let the program terminate.

The Exception Class Hierarchy

The classes that define various exceptions are related by inheritance, creating a class hierarchy that is shown in part in Figure B.7.

KEY CONCEPT

A new exception is defined by deriving a new class from the `Exception` class or one of its descendants.

The `Throwable` class is the parent of both the `Error` class and the `Exception` class. Many types of exceptions are derived from the `Exception` class, and these classes also have many children. Although these high-level classes are defined in the `java.lang` package, many child classes that define specific exceptions are part of several other packages. Inheritance relationships can span package boundaries.

We can define our own exceptions by deriving a new class from `Exception` or one of its descendants. The class we choose as the parent depends on what situation or condition the new exception represents.

After creating the class that defines the exception, we can create an object of that type as needed. The `throw` statement is used to throw the exception. For example:

```
throw new MyException();
```

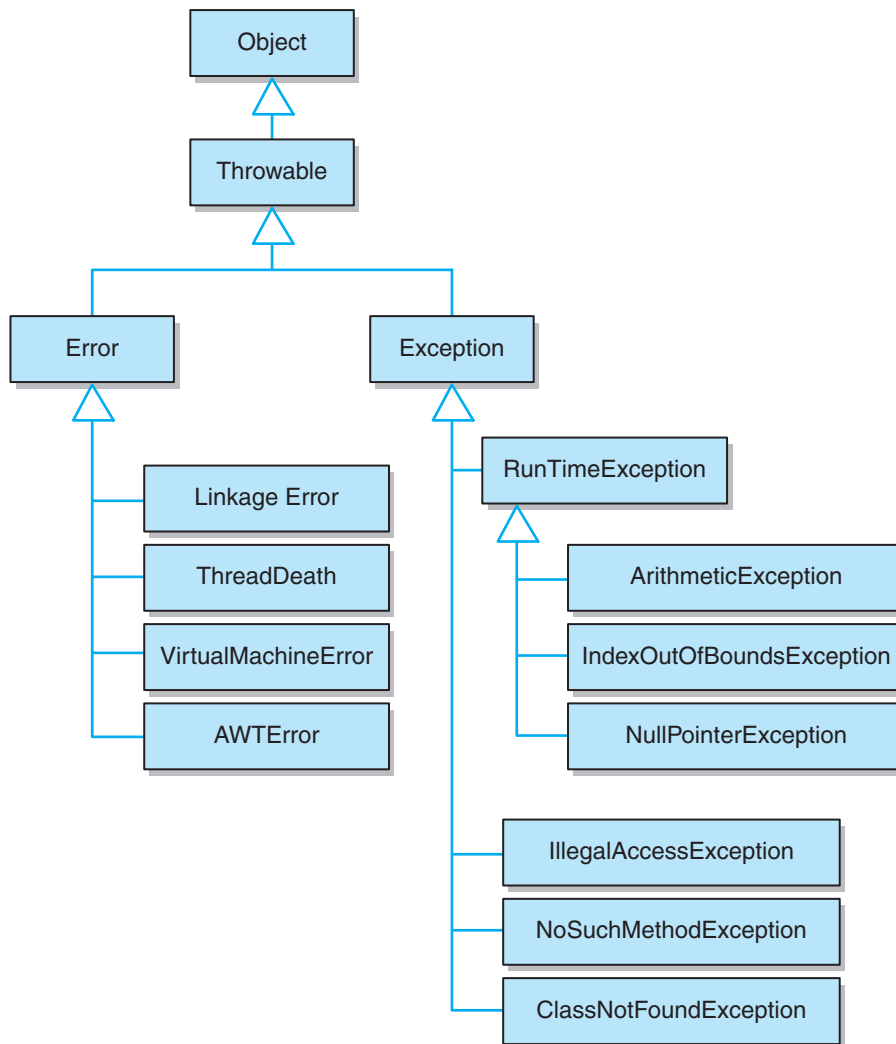


FIGURE B.7 Part of the `Error` and `Exception` class hierarchy

Summary of Key Concepts

- An abstraction hides details. A good abstraction hides the right details at the right time so that we can manage complexity.
- The `new` operator returns a reference to a newly created object.
- The Java standard class library is a useful set of classes that anyone can use when writing Java programs.
- A package is a Java language element used to group related classes under a common name.
- Each object has a state and a set of behaviors. The values of an object's variables define its state. The methods to which an object responds define its behaviors.
- A class is a blueprint for an object; it reserves no memory space for data. Each object has its own data space and thus its own state.
- The scope of a variable, which determines where it can be referenced, depends on where it is declared.
- Objects should be encapsulated. The rest of a program should interact with an object only through a well-defined interface.
- Instance variables should be declared with private visibility to promote encapsulation.
- A variable declared in a method is local to that method and cannot be used outside of it.
- A constructor cannot have any return type, even `void`.
- The versions of an overloaded method are distinguished by their signatures. The number, type, or order of their parameters must be distinct.
- An object reference variable stores the address of an object.
- The reserved word `null` represents a reference that does not point to a valid object.
- The `this` reference always refers to the currently executing object.
- Several references can refer to the same object. These references are aliases of each other.
- The `==` operator compares object references for equality, returning `true` if the references are aliases of each other.
- The `equals` method can be defined to determine equality between objects in any way we consider appropriate.

- If an object has no references to it, a program cannot use it. Java performs automatic garbage collection by periodically reclaiming the memory space occupied by these objects.
- When an object is passed to a method, the actual and formal parameters become aliases of each other.
- A static variable is shared among all instances of a class.
- A method is made static by using the `static` modifier in the method declaration.
- A wrapper class represents a primitive value so that it can be treated as an object.
- An interface is a collection of abstract methods. It cannot be instantiated.
- A class implements an interface, which formally defines a set of methods used to interact with objects of that class.
- Inheritance is the process of deriving a new class from an existing one.
- One purpose of inheritance is to reuse existing software.
- Inherited variables and methods can be used in the derived class as if they had been declared locally.
- Inheritance creates an is-a relationship between all parent and child classes.
- Visibility modifiers determine which variables and methods are inherited. Protected visibility provides the best possible encapsulation that permits inheritance.
- A parent's constructor can be invoked using the `super` reference.
- A child class can override (redefine) the parent's definition of an inherited method.
- The child of one class can be the parent of one or more other classes, creating a class hierarchy.
- Common features should be located as high in a class hierarchy as is reasonable, in order to minimize maintenance efforts.
- All Java classes are derived, directly or indirectly, from the `Object` class.
- The `toString` and `equals` methods are defined in the `Object` class and therefore are inherited by every class in every Java program.
- An abstract class cannot be instantiated. It represents a concept on which other classes can build their definitions.
- A class derived from an abstract parent must override all of its parent's abstract methods, or the derived class will also be considered abstract.

- Inheritance can be applied to interfaces, so that one interface can be derived from another interface.
- A polymorphic reference can refer to different types of objects over time.
- A reference variable can refer to any object created from any class related to it by inheritance.
- A polymorphic reference uses the type of the object, not the type of the reference, to determine which version of a method to invoke.
- An interface name can be used to declare an object reference variable. An interface reference can refer to any object of any class that implements the interface.
- Interfaces enable us to make polymorphic references, in which the method that is invoked is based on the particular object being referenced at the time.
- Errors and exceptions represent unusual or invalid processing.
- The messages printed by a thrown exception indicate the nature of the problem and provide a method call stack trace.
- Each `catch` clause on a `try` statement handles a particular kind of exception that may be thrown within the `try` block.
- If an exception is not caught and handled where it occurs, it is propagated to the calling method.
- A programmer must carefully consider how exceptions should be handled, if at all, and at what level.
- A new exception is defined by deriving a new class from the `Exception` class or one of its descendants.

Self-Review Questions

- SR B.1 What is the difference between an object and a class?
- SR B.2 Objects should be self-governing. Explain.
- SR B.3 Describe each of the following:
- a. public method
 - b. private method
 - c. public variable
 - d. private variable
- SR B.4 What are constructors used for? How are they defined?
- SR B.5 How are overloaded methods distinguished from each other?

- SR B.6 What is an aggregate object?
- SR B.7 What is the difference between a static variable and an instance variable?
- SR B.8 What is the difference between a class and an interface?
- SR B.9 Describe the relationship between a parent class and a child class.
- SR B.10 What relationship should every class derivation represent?
- SR B.11 What is the significance of the `Object` class?
- SR B.12 What is polymorphism?
- SR B.13 How is overriding related to polymorphism?
- SR B.14 How can polymorphism be accomplished using interfaces?

Exercises

- EX B.1 Identify each of the following as a class, an object, or a method.
 - > `superman`
 - > `breakChain`
 - > `SuperHero`
 - > `saveLife`
- EX B.2 Identify each of the following as a class, an object, or a method.
 - > `Beverage`
 - > `pepsi`
 - > `drink`
 - > `refill`
 - > `coke`
- EX B.3 Explain why a static method cannot refer to an instance variable.
- EX B.4 Can a class implement two interfaces that both contain the same method signature? Explain.
- EX B.5 Describe the relationship between a parent class and a child class.
- EX B.6 Draw and annotate a class hierarchy that represents various types of faculty at a university. Show what characteristics would be represented in the various classes of the hierarchy. Explain how polymorphism could play a role in the process of assigning courses to each faculty member.

Programming Projects

- PP B.1 Design and implement a class called `Sphere` that contains instance data that represent the sphere's diameter. Define the `Sphere` constructor to accept and initialize the diameter, and

include `getter` and `setter` methods for the diameter. Include methods that calculate and return the volume and surface area of the sphere (see Programming Project 3.2 for the formulas). Include a `toString` method that returns a one-line description of the sphere. Create a driver class called `MultiSphere`, whose `main` method instantiates and updates several `Sphere` objects.

- PP B.2 Design and implement a class called `Dog` that contains instance data that represent the dog's name and age. Define the `Dog` constructor to accept and initialize instance data. Include `getter` and `setter` methods for the name and age. Include a method to compute and return the age of the dog in "person years" (seven times the dog's age). Include a `toString` method that returns a one-line description of the dog. Create a driver class called `Kennel`, whose `main` method instantiates and updates several `Dog` objects.
- PP B.3 Design and implement a class called `Box` that contains instance data that represent the height, width, and depth of the box. Also include a `boolean` variable called `full` as instance data that represent whether the box is full or not. Define the `Box` constructor to accept and initialize the height, width, and depth of the box. Each newly created `Box` is empty (the constructor should initialize `full` to `false`). Include `getter` and `setter` methods for all instance data. Include a `toString` method that returns a one-line description of the box. Create a driver class called `BoxTest`, whose `main` method instantiates and updates several `Box` objects.
- PP B.4 Design and implement a class called `Book` that contains instance data for the title, author, publisher, and copyright date. Define the `Book` constructor to accept and initialize these data. Include `getter` and `setter` methods for all instance data. Include a `toString` method that returns a nicely formatted, multi-line description of the book. Create a driver class called `Bookshelf`, whose `main` method instantiates and updates several `Book` objects.
- PP B.5 Design and implement a class called `Flight` that represents an airline flight. It should contain instance data that represent the airline name, the flight number, and the flight's origin and destination cities. Define the `Flight` constructor to accept and initialize all instance data. Include `getter` and `setter` methods for all instance data. Include a `toString` method that returns a one-line description of the flight. Create a driver class called

- `FlightTest`, whose main method instantiates and updates several `Flight` objects.
- PP B.6 Design a Java interface called `Priority` that includes two methods: `setPriority` and `getPriority`. The interface should define a way to establish numeric priority among a set of objects. Design and implement a class called `Task` that represents a task (such as on a to-do list) that implements the `Priority` interface. Create a driver class to exercise some `Task` objects.
- PP B.7 Design a Java interface called `Lockable` that includes the following methods: `setKey`, `lock`, `unlock`, and `locked`. The `setKey`, `lock`, and `unlock` methods take an integer parameter that represents the key. The `setKey` method establishes the key. The `lock` and `unlock` methods lock and unlock the object, but only if the key passed in is correct. The `locked` method returns a boolean that indicates whether or not the object is locked. A `Lockable` object represents an object whose regular methods are protected: If the object is locked, the methods cannot be invoked; if it is unlocked, they can be invoked. Redesign and implement a version of the `Coin` class from Chapter 5 so that it is `Lockable`.
- PP B.8 Design and implement a set of classes that define the employees of a hospital: doctor, janitor, nurse, administrator, surgeon, receptionist, and so on. Include methods in each class that are named according to the services provided by that person and that print an appropriate message. Create a main driver class to instantiate and exercise several of the classes.
- PP B.9 Design and implement a set of classes that define various types of reading material: books, novels, magazines, technical journals, textbooks, and so on. Include data values that describe various attributes of the material, such as the number of pages and the names of the primary characters. Include methods that are named appropriately for each class and that print an appropriate message. Create a main driver class to instantiate and exercise several of the classes.
- PP B.10 Design and implement a set of classes that keeps track of demographic information about a set of people, such as age, nationality, occupation, income, and so on. Design each class to focus on a particular aspect of data collection. Create a main driver class to instantiate and exercise several of the classes.
- PP B.11 Design and implement a program that creates an exception class called `StringTooLongException`, designed to be thrown when a

string is discovered that has too many characters in it. In the `main` driver of the program, read strings from the user until the user enters “DONE”. If a string is entered that has too many characters (say 20), throw the exception. Allow the thrown exception to terminate the program.

- PP B.12 Modify the solution to Programming Project 10.1 such that it catches and handles the exception if it is thrown. Handle the exception by printing an appropriate message, and then continue processing more strings.

Answers to Self-Review Questions

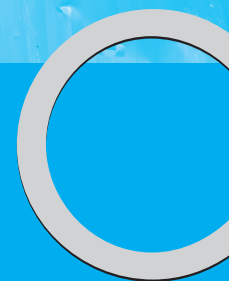
- SRA B.1 A class is the implementation of the blueprint for an object. An object is a specific instance of a class.
- SRA B.2 To say that objects should be self-governing means that only the methods of a particular object should be able to access or modify the object’s variables.
- SRA B.3
- A public method is a method (within a class) that has public visibility and may be called by a method of any other class that has declared a variable of the first class.
 - A private method is a method (within a class) that has private visibility and may be accessed only by methods within the class.
 - A public variable is a variable (within a class) that has public visibility and may be accessed by any method of any class that has declared a variable of the first class.
 - A private variable is a variable (within a class) that has private visibility and may be accessed only by methods within the class.
- SRA B.4 A constructor is the method that is called in the creation of an instance of a class. The constructor will typically initialize object attributes. Multiple constructors may be provided for various initialization strategies (e.g., no parameters for a default initialization or one or more parameters for more specific initializations).
- SRA B.5 Overloaded methods are distinguished from each other by their signatures, including the number and type of the parameters.
- SRA B.6 An aggregate object is an object that is made up of other objects.

- SRA B.7 A static variable is shared among all instances of a class, whereas an instance variable is unique to a particular instance.
- SRA B.8 A class provides implementations for all of its methods (unless it is an abstract class), whereas an interface simply provides the headings for each method.
- SRA B.9 The relationship between a child class and its parent class is called an is-a relationship. For example, if class B is derived from class A, then B is an instance of A with whatever additional information and methods are provided in class B.
- SRA B.10 Is-a.
- SRA B.11 The `java.lang.Object` class is the root of the class hierarchy for the Java language. This means that all classes in Java are ultimately derived from the `Object` class.
- SRA B.12 *Polymorphism* means “having many forms.” In object-oriented programming, we refer to an object reference as polymorphic if it can refer to objects of multiple classes.
- SRA B.13 Overriding is related to polymorphism because a parent class reference can point to objects of any of its descendant classes. One or more of these classes may have overridden methods from the parent class, causing the behavior of such a method to be dependent on the type of the object referenced by the call.
- SRA B.14 Polymorphism can be accomplished using interfaces because reference variables can be created using the interface type. Then those references can point to objects of any class that implements the interface.

This page is intentionally left blank.



Java Graphics



In Appendix D, we will cover the issues related to developing a graphical user interface (GUI) for a Java program. This appendix is provided to introduce the concepts and techniques used to manage Java graphics, draw shapes, and manage color.

Appendix

C.1 Pixels and Coordinates

A picture is represented on a computer by breaking it down into *pixels*, a term that is short for “picture elements.” A complete picture is stored by storing the color of each individual pixel. The more pixels used to represent a picture, the more realistic it looks when it is reproduced. The number of pixels used to represent a picture is called the *picture resolution*. The number of pixels that can be displayed by a monitor is called the *monitor resolution*.

When drawn, each pixel is mapped to a pixel on the monitor screen. Each computer system and programming language defines a coordinate system so that we can refer to particular pixels.

A traditional two-dimensional Cartesian coordinate system has two axes that meet at the origin. Values on either axis can be negative or positive. The Java programming language has a relatively simple coordinate system in which all of the visible coordinates are positive. Figure C.1 compares a traditional coordinate system to the Java coordinate system.

Each point in the Java coordinate system is represented using an (x, y) pair of values. Each graphical component in a Java program, such as a panel, has its own coordinate system, with the origin in the top-left corner at coordinates $(0, 0)$. The x -axis coordinates get larger as you move to the right, and the y -axis coordinates get larger as you move down.

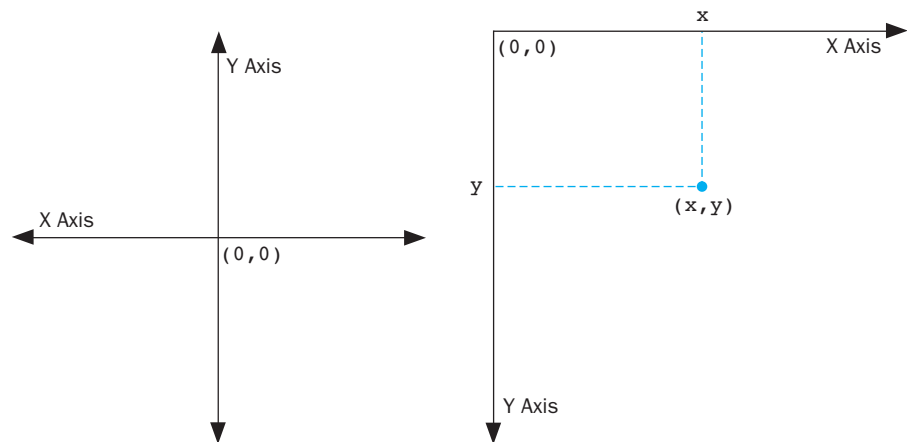


FIGURE C.1 A traditional coordinate system and the Java coordinate system

C.2 Representing Color

There are various ways to represent the color of a pixel. In the Java programming language, every color is represented as a mix of what the Java language refers to as the three *primary colors*: red, green, and blue. A color is specified using three numbers that are collectively referred to as an *RGB value*. RGB stands for Red-Green-Blue. Each number represents the relative contribution of a primary color.

Using 1 byte (8 bits) to store each of the three components of an RGB value, the numbers can range from 0 to 255. The level of each primary color determines the overall color. For example, high values of red and green combined with a low level of blue result in a shade of yellow.

In Java, a programmer uses the `Color` class, which is part of the `java.awt` package, to define and manage colors. Each object of the `Color` class represents a single color. The class contains several instances of itself to provide a basic set of predefined colors. Figure C.2 lists the predefined colors of the `Color` class. It also contains methods to define and manage many other colors.

Color	Object	RGB Value
black	<code>Color.black</code>	0, 0, 0
blue	<code>Color.blue</code>	0, 0, 255
cyan	<code>Color.cyan</code>	0, 255, 255
gray	<code>Color.gray</code>	128, 128, 128
dark gray	<code>Color.darkGray</code>	64, 64, 64
light gray	<code>Color.lightGray</code>	192, 192, 192
green	<code>Color.green</code>	0, 255, 0
magenta	<code>Color.magenta</code>	255, 0, 255
orange	<code>Color.orange</code>	255, 200, 0
pink	<code>Color.pink</code>	255, 175, 175
red	<code>Color.red</code>	255, 0, 0
white	<code>Color.white</code>	255, 255, 255
yellow	<code>Color.yellow</code>	255, 255, 0

FIGURE C.2 Predefined colors in the `Color` class

C.3 Drawing Shapes

The Java standard class library provides many classes that enable us to present and manipulate graphical information. The `Graphics` class, which is defined in the `java.awt` package, is fundamental to all such processing.

The `Graphics` class contains various methods that enable us to draw shapes, including lines, rectangles, and ovals. Figure C.3 lists some of the

```
void drawLine (int x1, int y1, int x2, int y2)
    Paints a line from point (x1, y1) to point (x2, y2).

void drawRect (int x, int y, int width, int height)
    Paints a rectangle with upper left corner (x, y) and dimensions width and
    height.

void drawOval (int x, int y, int width, int height)
    Paints an oval bounded by the rectangle with an upper left corner of (x, y) and
    dimensions width and height.

void drawString (String str, int x, int y)
    Paints the character string str at point (x, y), extending to the right.

void drawArc (int x, int y, int width, int height, int
startAngle, int arcAngle)
    Paints an arc along the oval bounded by the rectangle defined by x, y, width,
    and height. The arc starts at startAngle and extends for a distance defined by
    arcAngle.

void fillRect (int x, int y, int width, int height)
    Same as their draw counterparts, but filled with the current foreground color.

void fillOval (int x, int y, int width, int height)

void fillArc (int x, int y, int width, int height,
int startAngle, int arcAngle)

Color getColor ()
    Returns this graphics context's foreground color.

void setColor (Color color)
    Sets this graphics context's foreground color to the specified color.
```

FIGURE C.3 Some methods of the `Graphics` class

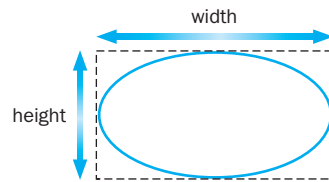


FIGURE C.4 An oval and its bounding rectangle

fundamental drawing methods of the `Graphics` class. These methods also let us draw circles and squares, which are just specific types of ovals and rectangles, respectively.

The methods of the `Graphics` class allow us to specify whether we want a shape filled or unfilled. An unfilled shape shows only the outline of the shape and is otherwise transparent (you can see any underlying graphics). A filled shape is solid between its boundaries and covers any underlying graphics.

Many of these methods accept parameters that specify the coordinates at which the shape should be drawn. Shapes drawn at coordinates that are outside the visible area will not be seen.

Many of the `Graphics` drawing methods are self-explanatory, but some require a little more discussion. Note, for instance, that an oval drawn by the `drawOval` method is defined by the coordinate of the upper-left corner and dimensions that specify the width and height of a *bounding rectangle*. Shapes with curves, such as ovals, are often defined by a rectangle that encompasses their perimeters. Figure C.4 depicts a bounding rectangle for an oval.

An arc can be thought of as a segment of an oval. To draw an arc, we specify the oval of which the arc is a part and the portion of the oval in which we're interested. The starting point of the arc is defined by the *start angle*, and the ending point of the arc is defined by the *arc angle*. The arc angle does not indicate where the arc ends, but rather its range. The start angle and the arc angle are measured in degrees. The origin for the start angle is an imaginary horizontal line passing through the center of the oval and can be referred to as 0° , as shown in Figure C.5.

Every graphics context has a current *foreground color* that is used whenever shapes or strings are drawn. Every surface that can be drawn on has a *background color*. The foreground color is set using the `setColor` method of the `Graphics` class, and the background color is set using the `setBackground` method of the component on which we are drawing, such as the panel.

Listing C.1 shows a program that uses various drawing and color methods to draw a winter scene featuring a snowman. The drawing is done on a `JPanel`, defined by the `SnowmanPanel` class, which is shown in Listing C.2.

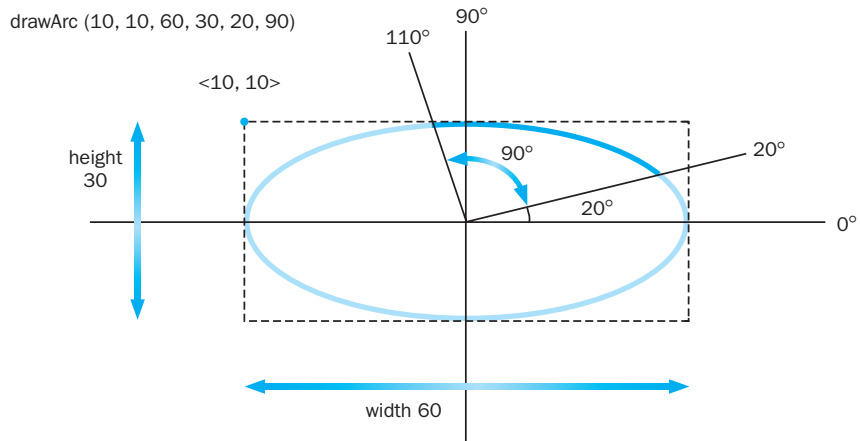


FIGURE C.5 An arc defined by an oval, a start angle, and an arc angle

The `paintComponent` method of a graphical component is called automatically when the component is rendered on the screen. Note that the `paintComponent` method accepts a `Graphics` object as a parameter. A `Graphics` object defines a particular *graphics context* with which we can interact. The graphics context passed into a panel's `paintComponent` method represents the graphics context in which the panel is drawn.

LISTING C.1

```
import javax.swing.JFrame;

/**
 * Snowman.java
 *
 * Demonstrates the use of basic drawing methods.
 */
public class Snowman
{
    /**
     * Displays a winter scene featuring a snowman.
     */
}
```

LISTING C.1*continued*

```
public static void main(String[] args)
{
    JFrame frame = new JFrame("Snowman");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    frame.getContentPane().add(new SnowmanPanel());

    frame.pack();
    frame.setVisible(true);
}
}
```

DISPLAY

LISTING C.2

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

/**
 * SnowmanPanel.java
 *
 * Represents the primary drawing panel for the Snowman application.
 */
public class SnowmanPanel extends JPanel
{
    private final int MID = 150;
    private final int TOP = 50;

    /**
     * Sets up the snowman panel.
     */
    public SnowmanPanel()
    {
        setPreferredSize(new Dimension(300, 225));
        setBackground(Color.cyan);
    }

    /**
     * Draws a snowman.
     */
    public void paintComponent(Graphics page)
    {
        super.paintComponent (page);
        page.setColor(Color.blue);
        page.fillRect(0, 175, 300, 50); // ground

        page.setColor(Color.yellow);
        page.fillOval(-40, -40, 80, 80); // sun

        page.setColor(Color.white);
        page.fillOval(MID-20, TOP, 40, 40); // head
        page.fillOval(MID-35, TOP+35, 70, 50); // upper torso
        page.fillOval(MID-50, TOP+80, 100, 60); // lower torso
    }
}
```

LISTING C.2

continued

```
page.setColor(Color.black);
page.fillOval(MID-10, TOP+10, 5, 5); // left eye
page.fillOval(MID+5, TOP+10, 5, 5); // right eye

page.drawArc(MID-10, TOP+20, 20, 10, 190, 160); // smile

page.drawLine(MID-25, TOP+60, MID-50, TOP+40); // left arm
page.drawLine(MID+25, TOP+60, MID+55, TOP+60); // right arm

page.drawLine(MID-20, TOP+5, MID+20, TOP+5); // brim of hat
page.fillRect(MID-15, TOP-20, 30, 25); // top of hat
}
}
```

The drawing of the snowman figure is based on two constant values called `MID` and `TOP`, which define the midpoint of the snowman (left to right) and the top of the snowman's head. The entire snowman figure is drawn relative to these values. Using constants like these makes it easier to create the snowman and to make modifications later. For example, to shift the snowman to the right or left in our picture, we need change only one constant declaration.

The call to the super `paintComponent` method as the first line in the `paintComponent` method ensures that the background color will be painted. The version of `paintComponent` defined in the `JPanel` class handles the display of the panel's background. The examples in Appendix D, which add graphical components (such as buttons) to a panel, do not need this call. If a panel contains graphical components, the parent's `paintComponent` method is automatically called. This is a key distinction between drawing on a component and adding a component to a container.

LISTING C.3

```
import javax.swing.JFrame;

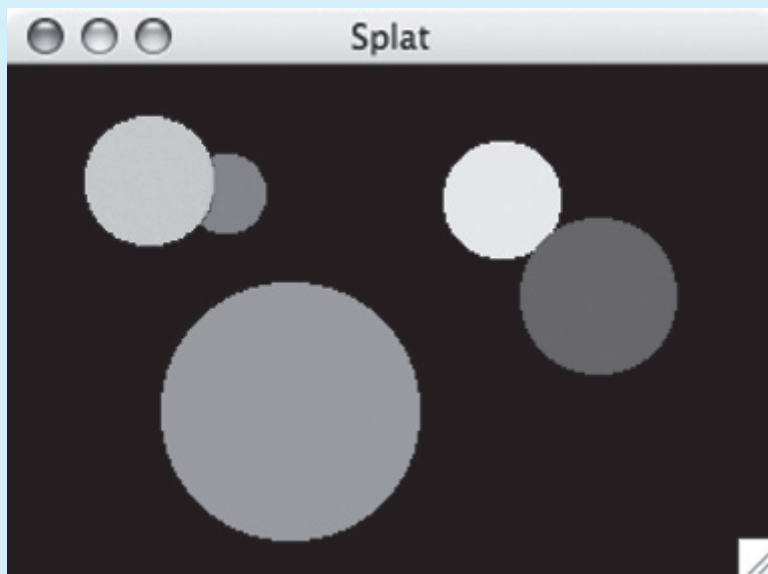
/**
 * Splat.java
 *
 * Demonstrates the use of graphical objects.
 */
```

LISTING C.3*continued*

```
public class Splat
{
    /**
     * Presents a set of circles.
     */
    public static void main(String[] args)
    {
        JFrame frame = new JFrame("Splat");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        frame.getContentPane().add(new SplatPanel());

        frame.pack();
        frame.setVisible(true);
    }
}
```

DISPLAY

Let's look at another example. The `Splat` class shown in Listing C.3 simply draws a few filled circles. The interesting thing about this program is not what it does but how it does it—each circle drawn in this program is represented by its own object.

The main method instantiates a `SplatPanel` object and adds it to the frame. The `SplatPanel` class is shown in Listing C.4. It is derived from `JPanel`, and it holds as instance data five `Circle` objects, which are instantiated in the panel's constructor. The `paintComponent` method in the `SplatPanel` class draws the panel by calling the `draw` method of each circle.

The `Circle` class is shown in Listing C.5. It defines instance data to store the size of the circle, its (x, y) location, and its color. The `draw` method of the `Circle` class simply draws the circle based on the values of its instance data.

The design of the `Splat` program embodies fundamental object-oriented thinking. Each circle manages itself and will draw itself in whatever graphics context you pass it. The `Circle` class is defined in a way that can be used in other situations and programs. There is a clean separation between the object being drawn and the component on which it is drawn.

LISTING C.4

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

/**
 * SplatPanel.java
 *
 * Demonstrates the use of graphical objects.
 */
public class SplatPanel extends JPanel
{
    private Circle circle1, circle2, circle3, circle4, circle5;

    /**
     * Creates five Circle objects.
     */
}
```

LISTING C.4 *continued*

```

public SplatPanel()
{
    circle1 = new Circle(30, Color.red, 70, 35);
    circle2 = new Circle(50, Color.green, 30, 20);
    circle3 = new Circle(100, Color.cyan, 60, 85);
    circle4 = new Circle(45, Color.yellow, 170, 30);
    circle5 = new Circle(60, Color.blue, 200, 60);

    setPreferredSize(new Dimension(300, 200));
    setBackground(Color.black);
}

/**
 * Draws this panel by requesting that each circle draw itself.
 */
public void paintComponent(Graphics page)
{
    super.paintComponent(page);

    circle1.draw(page);
    circle2.draw(page);
    circle3.draw(page);
    circle4.draw(page);
    circle5.draw(page);
}
}

```

LISTING C.5

```

import java.awt.*;

/**
 * Circle.java
 *
 * Represents a circle with a particular position, size, and color.
 */

```

LISTING C.5*continued*

```
public class Circle
{
    private int diameter, x, y;
    private Color color;

    /**
     * Sets up this circle with the specified values.
     */
    public Circle(int size, Color shade, int upperX, int upperY)
    {
        diameter = size;
        color = shade;
        x = upperX;
        y = upperY;
    }

    /**
     * Draws this circle in the specified graphics context.
     */
    public void draw(Graphics page)
    {
        page.setColor(color);
        page.fillOval(x, y, diameter, diameter);
    }
}
```

C.4 Polygons and Polylines

A polygon is a multi-sided shape that is defined in Java using a series of (x, y) points that indicate the vertices of the polygon. Arrays are often used to store the list of coordinates.

Polygons are drawn using methods of the `Graphics` class, in a manner similar to the way we draw rectangles and ovals. Like these other shapes, a polygon can be drawn filled or unfilled. The methods used to draw a polygon are called `drawPolygon` and `fillPolygon`. Both of these methods are overloaded. One version uses

arrays of integers to define the polygon, and the other uses an object of the `Polygon` class to define the polygon. We discuss the `Polygon` class later in this appendix.

In the version that uses arrays, the `drawPolygon` and `fillPolygon` methods take three parameters. The first is an array of integers representing the x coordinates of the points in the polygon, the second is an array of integers representing the corresponding y coordinates of those points, and the third is an integer that indicates how many points are used from each of the two arrays. Taken together, the first two parameters represent the (x, y) coordinates of the vertices of the polygons.

A polygon is always closed. A line segment is always drawn from the last point in the list to the first point in the list.

Much like a polygon, a *polyline* contains a series of points connected by line segments. Polylines differ from polygons in that the first and last coordinates are not automatically connected when they are drawn. Because a polyline is not closed, it cannot be filled. Therefore, there is only one method, called `drawPolyline`, used to draw a polyline. The parameters of the `drawPolyline` method are similar to those of the `drawPolygon` method.

The program shown in Listing C.6 uses polygons to draw a rocket. In the `RocketPanel` class, shown in Listing C.7 on page 531, the arrays called `xRocket` and `yRocket` define the points of the polygon that make up the main body of the rocket. The first point in the arrays is the upper tip of the rocket, and the points progress clockwise from there. The `xWindow` and `yWindow` arrays specify the points for the polygon that form the window in the rocket. Both the rocket and the window are drawn as filled polygons.

LISTING C.6

```
import javax.swing.JFrame;

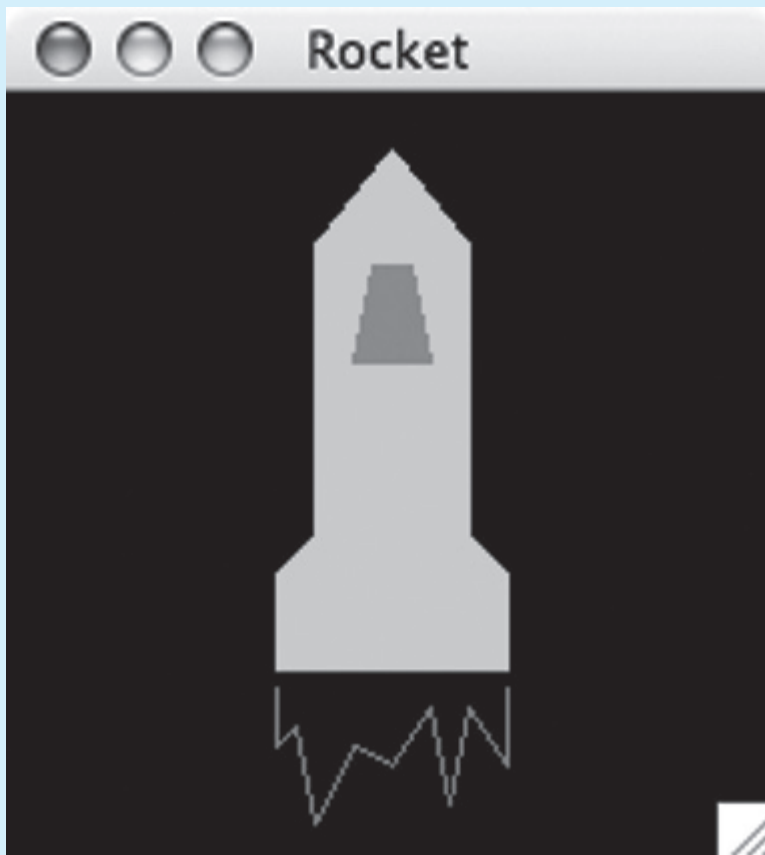
/**
 * Rocket.java
 *
 * Demonstrates the use of polygons and polylines.
 */
public class Rocket
{
    /**
     * Displays a rocket in flight.
     */
}
```

LISTING C.6 *continued*

```
public static void main(String[] args)
{
    JFrame frame = new JFrame("Rocket");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    frame.getContentPane().add(new RocketPanel());

    frame.pack();
    frame.setVisible(true);
}
}
```

DISPLAY

```

Polygon ()
    Constructor: Creates an empty polygon.

Polygon (int[] xpoints, int[] ypoints, int npoints)
    Constructor: Creates a polygon using the (x, y) coordinate pairs
    in corresponding entries of xpoints and ypoints.

void addPoint (int x, int y)
    Appends the specified point to this polygon.

boolean contains (int x, int y)
    Returns true if the specified point is contained in this polygon.

boolean contains (Point p)
    Returns true if the specified point is contained in this polygon.

Rectangle getBounds ()
    Gets the bounding rectangle for this polygon.

void translate (int deltaX, int deltaY)
    Translates the vertices of this polygon by deltaX along the x axis
    and deltaY along the y axis.

```

FIGURE C.6 Some methods of the `Polygon` class

The `xFlame` and `yFlame` arrays define the points of a polyline that are used to create the image of flame shooting out of the tail of the rocket. Because it is drawn as a polyline, not as a polygon, the flame is not closed or filled.

The `Polygon` Class

A polygon can also be defined explicitly using an object of the `Polygon` class, which is defined in the `java.awt` package of the Java standard class library. Two versions of the overloaded `drawPolygon` and `fillPolygon` methods take a single `Polygon` object as a parameter.

A `Polygon` object encapsulates the coordinates of the polygon sides. The constructors of the `Polygon` class allow the creation of an initially empty polygon, or one defined by arrays of integers representing the point coordinates. The `Polygon` class contains methods to add points to the polygon and to determine whether a given point is contained within the polygon shape. It also contains methods to get a representation of a bounding rectangle for the polygon, as well as a method to translate all of the points in the polygon to another position. Figure C.6 lists these methods.

LISTING C.7

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

/**
 * RocketPanel.java
 *
 * Demonstrates the use of polygons and polylines.
 */
public class RocketPanel extends JPanel
{
    private int[] xRocket = {100, 120, 120, 130, 130, 70, 70, 80, 80};
    private int[] yRocket = {15, 40, 115, 125, 150, 150, 125, 115, 40};

    private int[] xWindow = {95, 105, 110, 90};
    private int[] yWindow = {45, 45, 70, 70};

    private int[] xFlame = {70, 70, 75, 80, 90, 100, 110, 115, 120,
        130, 130};
    private int[] yFlame = {155, 170, 165, 190, 170, 175, 160, 185,
        160, 175, 155};

    /**
     * Sets up the basic characteristics of this panel.
     */
    public RocketPanel()
    {
        setBackground(Color.black);
        setPreferredSize(new Dimension(200, 200));
    }

    /**
     * Draws a rocket using polygons and polylines.
     */
    public void paintComponent(Graphics page)
    {
        super.paintComponent(page);
        page.setColor(Color.cyan);
        page.fillPolygon(xRocket, yRocket, xRocket.length);
    }
}
```

LISTING C.7 *continued*

```
page.setColor(Color.gray);
page.fillPolygon(xWindow, yWindow, xWindow.length);
page.setColor(Color.red);
page.drawPolyline(xFlame, yFlame, xFlame.length);
}
}
```

Exercises

- EX C.1 Compare and contrast a traditional coordinate system and the coordinate system used by Java graphical components.
- EX C.2 How many bits are needed to store a color picture that is 400 pixels wide and 250 pixels high? Assume that color is represented using the RGB technique described in this appendix and that no special compression is done.
- EX C.3 Assuming you have a `Graphics` object called `page`, write a statement that will draw a line from point (20, 30) to point (50, 60).
- EX C.4 Assuming you have a `Graphics` object called `page`, write a statement that will draw a rectangle with height 70 and width 35, such that its upper-left corner is at point (10, 15).
- EX C.5 Assuming you have a `Graphics` object called `page`, write a statement that will draw a circle centered on point (50, 50) with a radius of 20 pixels.
- EX C.6 The following lines of code draw the eyes of the snowman in the `Snowman` program. The eyes seem centered on the face when drawn, but the first parameters of each call are not equally offset from the midpoint. Explain.
- ```
page.fillOval(MID-10, TOP+10, 5, 5);
page.fillOval(MID+5, TOP+10, 5, 5);
```
- EX C.7 Write a method called `randomColor` that creates and returns a `Color` object that represents a random color.
- EX C.8 Write a method called `drawCircle` that draws a circle based on the method's parameters: a `Graphics` object through which to draw the circle, two integer values representing the (x, y) coordinates of the center of the circle, another integer that represents the circle's radius, and a `Color` object that defines the circle's color. The method does not return anything.

## Programming Projects

- PP C.1 Create a revised version of the `Snowman` program with the following modifications:
- Add two red buttons to the upper torso.
  - Make the snowman frown instead of smile.
  - Move the sun to the upper-right corner of the picture.

- Display your name in the upper-left corner of the picture.
  - Shift the entire snowman 20 pixels to the right.
- PP C.2 Write a program that writes your name using the `drawString` method.
- PP C.3 Write a program that draws the Big Dipper. Add some extra stars in the night sky.
- PP C.4 Write a program that draws some balloons tied to strings. Make the balloons various colors.
- PP C.5 Write a program that draws the Olympic logo. The circles in the logo should be colored (from left to right) blue, yellow, black, green, and red.
- PP C.6 Write a program that displays a business card of your own design. Include both graphics and text.
- PP C.7 Write a program that shows a pie chart with eight equal slices, all colored differently.
- PP C.8 Write a program that draws a house with a door (and doorknob), windows, and a chimney. Add some smoke coming out of the chimney and some clouds in the sky.
- PP C.9 Modify the program from Programming Project C.8 to include a simple fence with vertical, equally spaced slats backed by two horizontal support boards. Make sure the house is visible between the slats in the fence.
- PP C.10 Write a program that draws 20 horizontal, evenly spaced parallel lines of random length.
- PP C.11 Write a program that draws the side view of stair steps from the lower left to the upper right.
- PP C.12 Write a program that draws 100 circles of random color and random diameter in random locations. Ensure that, in each case, the entire circle appears in the visible area of the applet.
- PP C.13 Write a program that draws 10 concentric circles of random radius.
- PP C.14 Write a program that draws a brick wall pattern in which each row of bricks is offset from the row above and the row below it.
- PP C.15 Design and implement a program that draws a rainbow. Use tightly spaced concentric arcs to draw each part of the rainbow in a particular color.
- PP C.16 Design and implement a program that draws 20,000 points in random locations within the visible area. Make the points on the

- left half of the panel appear in red and the points on the right half of the panel appear in green. Draw each point by drawing a line with a length of only one pixel.
- PP C.17 Design and implement a program that draws 10 circles of random radius in random locations. Fill in the largest circle in red.
- PP C.18 Write a program that draws a quilt in which a simple pattern is repeated in a grid of squares.
- PP C.19 Modify the program from Programming Project C.18 such that it draws a quilt using a separate class called `Pattern` that represents a particular pattern. Allow the constructor of the `Pattern` class to vary some characteristics of the pattern, such as its color scheme. Instantiate two separate `Pattern` objects and incorporate them in a checkerboard layout in the quilt.
- PP C.20 Design and implement a class called `Building` that represents a graphical depiction of a building. Allow the parameters to the constructor to specify the building's width and height. Each building should be colored black and should contain a few random windows of yellow. Create a program that draws a random skyline of buildings.
- PP C.21 Write a program that displays a graphical seating chart for a dinner party. Create a class called `Diner` (as in one who dines) that stores the person's name, gender, and location at the dinner table. A diner is graphically represented as a circle, color-coded by gender, with the person's name printed in the circle.
- PP C.22 Create a class called `Crayon` that represents one crayon of a particular color and length (height). Design and implement a program that draws a box of crayons.
- PP C.23 Create a class called `Star` that represents a graphical depiction of a star. Let the constructor of the star accept the number of points in the star (4, 5, or 6), the radius of the star, and the center point location. Write a program that draws a sky containing various types of stars.
- PP C.24 Design and implement an application that displays an animation of a horizontal line segment moving across the screen, eventually passing across a vertical line. As the vertical line is passed, the horizontal line should change color. The change of color should occur while the horizontal line crosses the vertical line; therefore, while it is crossing, the horizontal line will be two different colors.



- PP C.25 Create a class that represents a spaceship, which can be drawn (side view) in any particular location. Use it to create a program that displays the spaceship so that it follows the movement of the mouse. When the mouse button is pressed down, have a laser beam shoot out of the front of the spaceship (one continuous beam, not a moving projectile) until the mouse button is released.



# Graphical User Interfaces

# D

**M**any programs provide a graphical user interface (GUI) through which a user interacts with the program. As the name implies, a GUI makes use of graphical screen components such as windows, buttons, check boxes, menus, and text fields. GUIs often provide a more natural and rich experience for the user, compared to a simple text-based, command-line environment. This appendix explores the various issues related to developing a GUI in Java.

## Appendix

## D.1 GUI Elements

The text-based programs we've seen in previous examples are *command-line applications*, which interact with the user through simple prompts and feedback. This type of interface is straightforward to understand, but it lacks the rich user experience possible when a true *graphical user interface* (GUI) is used. With a GUI, the user is not limited to responding to prompts in a particular order and receiving feedback in one place. Instead, the user can interact as needed with various components such as buttons and text fields. This chapter explores the many issues involved in developing a GUI in Java.

Let's start with an overview of the concepts that underlie every GUI-based program. At least three kinds of objects are needed to create a GUI in Java:

- components
- events
- listeners

A GUI *component* is an object that defines a screen element used to display information or allow the user to interact with a program in a certain way. Examples of GUI components include push buttons, text fields, labels, scroll bars, and menus. A *container* is a special type of component that is used to hold and organize other components.

An *event* is an object that represents some occurrence in which we may be interested. Often, events correspond to user actions, such as pressing a mouse button or typing a key on the keyboard. Most GUI components generate events to indicate a user action related to that component. For example, a button component will generate an event to indicate that the button has been pushed. A program that is oriented around a GUI, responding to events from the user, is called *event-driven*.

A *listener* is an object that “waits” for an event to occur and responds in some way when it does. A big part of designing a GUI-based program is establishing the relationships among the listener, the event it listens for, and the component that will generate the event.

### KEY CONCEPT

A GUI is made up of components, events that represent user actions, and listeners that respond to those events.

For the most part, we will use components and events that are predefined by classes in the Java class library. We will tailor the behavior of the components, but their basic roles have been established already. We will, however, write our own listener classes to perform whatever actions we desire when events occur.

To create a Java program that uses a GUI, then, we must

- instantiate and set up the necessary components,
- implement listener classes that define what happens when particular events occur, and
- establish the relationship between the listeners and the components that generate the events of interest.

Java components and other GUI-related classes are defined primarily in two packages: `java.awt` and `javax.swing`. (Note the `x` in `javax.swing`.) The *Abstract Windowing Toolkit* (AWT) was the original Java GUI package. It still contains many important classes that we will use. The *Swing* package was added later and provides components that are more versatile than those of the AWT package. Both packages are needed for GUI development, but we will use Swing components whenever there is an option.

In some respects, once you have a basic understanding of event-driven programming, the rest is just detail. There are many types of components you can use that produce many types of events that you may want to acknowledge. But they all work in the same basic way. They all have the same core relationships to one another.

Let's look at a simple example that contains all of the basic GUI elements. The `PushCounter` class shown in Listing D.1 contains the driver of a program that presents the user with a single push button (labeled "Push Me!"). Each time the button is pushed, a counter is updated and displayed.

The components used in this program include a button, a label to display the count, a panel to hold the button and label, and a frame to display the panel. The panel is defined by the `PushCounterPanel` class, which is shown in Listing D.2. Let's look at each of these pieces in more detail.

## Frames and Panels

A *frame* is a container that is used to display GUI-based Java applications. A frame is displayed as a separate window with its own title bar. It can be repositioned on the screen and resized as needed by dragging it with the mouse. It contains small buttons in the corner of the frame that allow the frame to be minimized, maximized, and closed. A frame is defined by the `JFrame` class.

A *panel* is also a container. However, unlike a frame, it cannot be displayed on its own. A panel must be added to another container for it to be displayed. Generally, a panel doesn't move unless you move the container that it's in. Its primary role is to help organize the other components in a GUI. A panel is defined by the `JPanel` class.

### KEY CONCEPT

A frame is displayed as a separate window, but a panel can be displayed only as part of another container.

We can classify containers as either heavyweight or lightweight. A *heavyweight container* is one that is managed by the underlying operating system on which the program is run, whereas a *lightweight container* is managed by the Java program itself. A frame is a heavyweight component, and a panel is a lightweight component. Another heavyweight container is an *applet*, which is used to display and execute a Java program through a web browser.

## LISTING D.1

```
import javax.swing.JFrame;

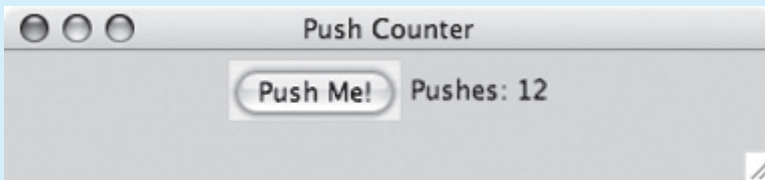
/**
 * PushCounter.java
 *
 * Demonstrates a graphical user interface and an event listener.
 */
public class PushCounter
{
 /**
 * Creates and displays the main program frame.
 */

 public static void main(String[] args)
 {
 JFrame frame = new JFrame("Push Counter");
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

 PushCounterPanel panel = new PushCounterPanel();
 frame.getContentPane().add(panel);

 frame.pack();
 frame.setVisible(true);
 }
}
```

## DISPLAY



Heavyweight components are more complex than lightweight components in general. A frame, for example, has multiple *panes*, which are responsible for various characteristics of the frame window. All visible elements of a Java interface are displayed in a frame's *content pane*.

## LISTING D.2

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

/**
 * PushCounterPanel.java
 *
 * Demonstrates a graphical user interface and an event listener.
 */
public class PushCounterPanel extends JPanel
{
 private int count;
 private JButton push;
 private JLabel label;

 /**
 * Constructor: Sets up the GUI.
 */
 public PushCounterPanel()
 {
 count = 0;

 push = new JButton("Push Me!");
 push.addActionListener(new ButtonListener());

 label = new JLabel("Pushes: " + count);

 add(push);
 add(label);

 setBackground(Color.cyan);
 setPreferredSize(new Dimension(300, 40));
 }
}
```

## LISTING D.2

*continued*

```

/**
 * Represents a listener for button push (action) events.
 */
private class ButtonListener implements ActionListener
{
 /**
 * Updates the counter and label when the button is pushed.
 */
 public void actionPerformed(ActionEvent event)
 {
 count++;
 label.setText("Pushes: " + count);
 }
}
}

```

Generally, we can create a Java GUI-based application by creating a frame in which the program interface is displayed. The interface is often organized onto a primary panel, which is added to the frame's content pane. The components in the primary panel are sometimes organized using other panels as needed.

In the main method of the `PushCounter` class, the frame for the program is constructed, set up, and displayed. The `JFrame` constructor takes a string as a parameter, which it displays in the title bar of the frame. The call to the `setDefaultCloseOperation` method determines what will happen when the close button in the corner of the frame is clicked. In most cases we'll simply let that button terminate the program, as indicated by the `EXIT_ON_CLOSE` constant.

The content pane of the frame is obtained using the `getContentPane` method, immediately after which the `add` method of the content pane is called to add the panel. The `pack` method of the frame sets its size appropriately based on its contents—in this case the frame is sized to accommodate the size of the panel it contains. This is a better approach than trying to set the size of the frame explicitly, which should change as the components within the frame change. The call to the `setVisible` method causes the frame to be displayed on the monitor screen.

You can interact with the frame itself in various ways. You can move the entire frame to another point on the desktop by grabbing the title bar of the frame and dragging it with the mouse. You can also resize the frame by dragging the bottom-right corner of the frame.

A panel is created by instantiating the `JPanel` class. In the case of the `PushCounter` program, the panel is represented by the `PushCounterPanel` class, which is derived from `JPanel`. Thus a `PushCounterPanel` is a `JPanel`, inheriting all of its methods and attributes. This is a common technique for creating panels.

The constructor of the `PushCounterPanel` class makes calls to several methods inherited from `JPanel`. For example, the background color of the panel is set using the `setBackground` method (the `Color` class is described in Appendix C). The `setPreferredSize` method accepts a `Dimension` object as a parameter, which is used to indicate the width and height of the component in pixels. The size of many components can be set this way, and most also have methods called `setMinimumSize` and `setMaximumSize` to help control the look of the interface.

A panel's `add` method allows a component to be added to the panel. In the `PushCounterPanel` constructor, a newly created button and label are added to the panel and are, from that point on, considered part of that panel. The order in which components are added to a container often matters. In this case, it determines that the button appears before the label.

A container is governed by a *layout manager*, which determines exactly how the components added to the panel will be displayed. The default layout manager for a panel simply displays components in the order in which they are added, with as many components on one line as possible. Layout managers are discussed in detail later in this appendix.

## Buttons and Action Events

The `PushCounter` program displays a button and a label. A *label*, created from the `JLabel` class, is a component that displays a line of text in a GUI. A label can also be used to display an image, as shown in later examples. In the `PushCounter` program, the label displays the number of times the button has been pushed.

Labels can be found in most GUI-based programs. They are very useful for displaying information or for labeling other components in the GUI. However, labels are not interactive. That is, the user does not interact with a label directly. The component that makes the `PushCounter` program interactive is the button that the user pushes with the mouse.

A *push button* is a component that allows the user to initiate an action with a press of the mouse. A push button is defined by the `JButton` class. A call to the `JButton` constructor takes a `String` parameter that specifies the text shown on the button.



A `JButton` generates an *action event* when it is pushed. There are several types of events defined in the Java standard class library, and we explore many of them throughout this appendix. Different components generate different types of events.

The only event of interest in this program occurs when the button is pushed. To respond to the event, we must create a listener object for that event, so we must write a class that represents the listener. In this case, we need an action event listener.

In the `PushButton` program, the `ButtonListener` class represents the action listener. We could write the `ButtonListener` class in its own file, or even in the same file but outside of the `PushCounterPanel` class. However, then we would have to set up a way to communicate between the listener and the components of the GUI that the listener updates. Instead, we define the `ButtonListener` class as an *inner class*, which is a class defined within another class. As such, it automatically has access to the members of the class that contains it. You should create inner classes only in situations in which there is an intimate relationship between the two classes, and the inner class is not accessed by any other class. The relationship between a listener and its GUI is one of the few situations in which an inner class is appropriate.

### KEY CONCEPT

Listeners are often defined as inner classes because of the intimate relationship between the listener and the GUI components.

Listener classes are written by implementing an *interface*, which is a list of methods that the implementing class must define. The Java standard class library contains interfaces for many types of events. An action listener is created by implementing the `ActionListener` interface; therefore, we include the `implements` clause in the `ButtonListener` class. Interfaces are discussed in more detail in Appendix B.

The only method listed in the `ActionListener` interface is the `actionPerformed` method, so that's the only method that the `ButtonListener` class must implement. The component that generates the action event (in this case the button) will call the `actionPerformed` method when the event occurs, passing in an `ActionEvent` object that represents the event. Sometimes we will use this event object, and other times it is sufficient just to know that the event occurred. In this case, we have no need to interact with the event object. When the event occurs, the listener increments the count and resets the text of the label by using the `setText` method.

Remember, we not only have to create a listener for an event, we must also set up the relationship between the listener and the component that will generate the event. To do so, we add the listener to the component by calling the appropriate method. In the `PushCounterPanel` constructor, we call the `addActionListener` method, passing in a newly instantiated `ButtonListener` object.

Review this example carefully, noting how it accomplishes the three key steps to creating an interactive GUI-based program. It creates and sets up the GUI components, creates the appropriate listener for the event of interest, and sets up the relationship between the listener and the component that will generate the event.

## Determining Event Sources

Let's look at an example in which one listener object is used to listen to two different components. The program represented by the `LeftRight` class, shown in Listing D.3, displays a label and two buttons. When the Left button is pressed, the label displays the word Left, and when the Right button is pressed, the label displays the word Right.

### LISTING D.3

```
import javax.swing.JFrame;

/**
 * LeftRight.java
 *
 * Demonstrates the use of one listener for multiple buttons.
 */
public class LeftRight
{
 /**
 * Creates and displays the main program frame.
 */
 public static void main(String[] args)
 {
 JFrame frame = new JFrame("Left Right");
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

 frame.getContentPane().add(new LeftRightPanel());

 frame.pack();
 frame.setVisible(true);
 }
}
```

### DISPLAY



The `LeftRightPanel` class, shown in Listing D.4, creates one instance of the `ButtonListener` class and then adds that listener to both buttons. Therefore, when either button is pressed, the `actionPerformed` method of the `ButtonListener` class is invoked.

#### LISTING D.4

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

/**
 * LeftRightPanel.java
 *
 * Demonstrates the use of one listener for multiple buttons.
 */
public class LeftRightPanel extends JPanel
{
 private JButton left, right;
 private JLabel label;
 private JPanel buttonPanel;

 /**
 * Constructor: Sets up the GUI.
 */
 public LeftRightPanel()
 {
 left = new JButton("Left");
 right = new JButton("Right");

 ButtonListener listener = new ButtonListener();
 left.addActionListener(listener);
 right.addActionListener(listener);

 label = new JLabel("Push a button");

 buttonPanel = new JPanel();
 buttonPanel.setPreferredSize(new Dimension(200, 40));
 buttonPanel.setBackground(Color.blue);
 buttonPanel.add(left);
 buttonPanel.add(right);
 }
}
```

## LISTING D.4

*continued*

```
 setPreferredSize(new Dimension(200, 80));
 setBackground(Color.cyan);
 add(label);
 add(buttonPanel);
 }

 /**
 * Represents a listener for both buttons.
 */
 private class ButtonListener implements ActionListener
 {

 /**
 * Determines which button was pressed and sets the label
 * text accordingly.
 */
 public void actionPerformed(ActionEvent event)
 {
 if (event.getSource() == left)
 label.setText("Left");
 else
 label.setText("Right");
 }
 }
}
```

On each invocation, the `actionPerformed` method uses an `if-else` statement to determine which button generated the event. The `getSource` method is called on the `ActionEvent` object that the button passes into the `actionPerformed` method. The `getSource` method returns a reference to the component that generated the event. The condition of the `if` statement compares the event source to the reference to the left button. If they don't match, then the event must have been generated by the right button.

We could have created two separate listener classes, one to listen to the left button and another to listen to the right button. In that case, the `actionPerformed` method would not have to determine the source of the event. Whether to have

multiple listeners or to determine the event source when it occurs is a design decision that should be made depending on the situation.

Note that the two buttons are put on the same panel called `buttonPanel`, which is separate from the panel represented by the `LeftRightPanel` class. By putting both buttons on one panel, we can guarantee their visual relationship to each other even when the frame is resized in various ways. For buttons labeled `Left` and `Right`, that is certainly important.

## D.2 More Components

In addition to push buttons, there are a variety of other interactive components that can be used in a GUI, each with a particular role to play. Let's examine a few more.

### Text Fields

A *text field* allows the user to enter typed input from the keyboard. The `Fahrenheit` program shown in Listing D.5 presents a GUI that includes a text field into which the user can type a Fahrenheit temperature. When the user presses the `Enter` (or `Return`) key, the equivalent Celsius temperature is displayed.

The interface for the `Fahrenheit` program is set up in the `FahrenheitPanel` class, shown in Listing D.6. The text field is an object of the `JTextField` class. The `JTextField` constructor takes an integer parameter that specifies the size of the field in number of characters based on the current default font.

The text field and various labels are added to the panel to be displayed. Remember that the default layout manager for a panel puts as many components on a line as it can fit, so if you resize the frame, the orientation of the labels and text field may change.

#### LISTING D.5

```
import javax.swing.JFrame;

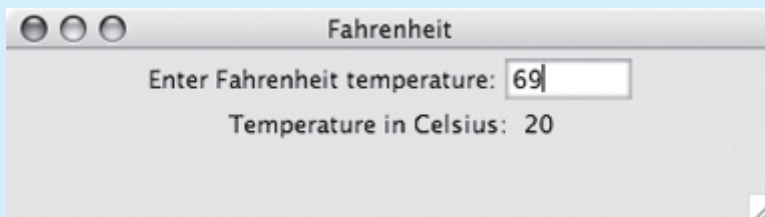
/**
 * Fahrenheit.java
 *
 * Demonstrates the use of text fields.
 */
public class Fahrenheit
```

**LISTING D.5***continued*

```
{
 /**
 * Creates and displays the temperature converter GUI.
 */
 public static void main(String[] args)
 {
 JFrame frame = new JFrame("Fahrenheit");
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

 FahrenheitPanel panel = new FahrenheitPanel();
 frame.getContentPane().add(panel);

 frame.pack();
 frame.setVisible(true);
 }
}
```

**DISPLAY****LISTING D.6**

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

/**
 * FahrenheitPanel.java
 * Demonstrates the use of text fields.
 */
public class FahrenheitPanel extends JPanel
{
 private JLabel inputLabel, outputLabel, resultLabel;
 private JTextField fahrenheit;
```

## LISTING D.6

*continued*

```

/**
 * Constructor: Sets up the main GUI components.
 */
public FahrenheitPanel()
{
 inputLabel = new JLabel("Enter Fahrenheit temperature:");
 outputLabel = new JLabel("Temperature in Celsius: ");
 resultLabel = new JLabel("---");

 fahrenheit = new JTextField(5);
 fahrenheit.addActionListener(new TempListener());

 add(inputLabel);
 add(fahrenheit);
 add(outputLabel);
 add(resultLabel);

 setPreferredSize(new Dimension(300, 75));
 setBackground(Color.yellow);
}

/**
 * Represents an action listener for the temperature input field.
 */
private class TempListener implements ActionListener
{
 /**
 * Performs the conversion when the enter key is pressed in
 * the text field.
 */
 public void actionPerformed(ActionEvent event)
 {
 int fahrenheitTemp, celsiusTemp;

 String text = fahrenheit.getText();

 fahrenheitTemp = Integer.parseInt(text);
 celsiusTemp = (fahrenheitTemp-32) * 5/9;

 resultLabel.setText(Integer.toString(celsiusTemp));
 }
}
}

```

A text field generates an action event when the Enter or Return key is pressed (and the cursor is in the text field). Therefore, we need to set up a listener object to respond to action events, much as in previous examples.

The text field component calls the `actionPerformed` method when the user presses the Enter key. The method first retrieves the text from the text field by calling its `getText` method, which returns a character string. The text is converted into an integer using the `parseInt` method of the `Integer` wrapper class. Then the method performs the calculation to determine the equivalent Celsius temperature and sets the text of the appropriate label with the result.

Note that a push button and a text field generate the same kind of event: an action event. Thus an alternative to the Fahrenheit program design is to add to the GUI a `JButton` object that causes the conversion to occur when the user uses the mouse to press the button. For that matter, the same listener object can be used to listen to multiple components at the same time, so the listener could be added to both the text field and the button, giving the user the option. Pressing either the button or the Enter key will cause the conversion to be performed. These variations are left as programming projects.

## Check Boxes

A *check box* is a button that can be toggled on or off using the mouse, indicating that a particular boolean condition is set or unset. Although you might have a group of check boxes indicating a set of options, each check box operates independently. That is, each can be set to on or off, and the status of one does not influence the others.

The program in Listing D.7 displays two check boxes and a label. The check boxes determine whether the text of the label is displayed in bold, italic, both, or neither. Any combination of bold and italic is valid. For example, both check boxes could be checked (on), in which case the text is displayed in both bold and italic. If neither is checked, the text of the label is displayed in a plain style.

The GUI for the `StyleOptions` program is embodied in the `StyleOptionsPanel` class shown in Listing D.8. A check box is represented by the `JCheckBox` class. When a check box changes state from selected (checked) to deselected (unchecked), or vice versa, it generates an *item event*. The `ItemListener` interface contains a single method called `itemStateChanged`. In this example, we use the same listener object to handle both check boxes.

This program also uses the `Font` class, which represents a particular character font. A `Font` object is defined by the font name, the font style, and the font size. The font name establishes the general visual characteristics of the characters. We are using the Helvetica font in this program. The style of a Java font can be plain,



bold, italic, or bold and italic combined. The listener is set up to change the characteristics of our font style.

The style of a font is represented as an integer, and integer constants defined in the `Font` class are used to represent the various aspects of the style. The constant `PLAIN` is used to represent a plain style. The constants `BOLD` and `ITALIC` are used to represent bold and italic, respectively. The sum of the `BOLD` and `ITALIC` constants indicates a style that is both bold and italic.

### LISTING D.7

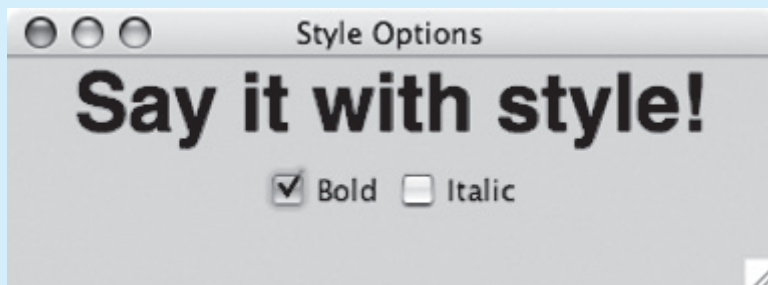
```
import javax.swing.JFrame;

/**
 * StyleOptions.java
 *
 * Demonstrates the use of check boxes.
 */
public class StyleOptions
{
 /**
 * Creates and displays the style options frame.
 */
 public static void main(String[] args)
 {
 JFrame frame = new JFrame("Style Options");
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

 frame.getContentPane().add(new StyleOptionsPanel());

 frame.pack();
 frame.setVisible(true);
 }
}
```

### DISPLAY



## LISTING D.8

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

/**
 * StyleOptionsPanel.java
 *
 * Demonstrates the use of check boxes.
 */
public class StyleOptionsPanel extends JPanel
{
 private JLabel saying;
 private JCheckBox bold, italic;

 /**
 * Sets up a panel with a label and some check boxes that
 * control the style of the label's font.
 */
 public StyleOptionsPanel()
 {
 saying = new JLabel("Say it with style!");
 saying.setFont(new Font("Helvetica", Font.PLAIN, 36));

 bold = new JCheckBox("Bold");
 bold.setBackground(Color.cyan);
 italic = new JCheckBox("Italic");
 italic.setBackground(Color.cyan);

 StyleListener listener = new StyleListener();
 bold.addItemListener(listener);
 italic.addItemListener(listener);

 add(saying);
 add(bold);
 add(italic);

 setBackground(Color.cyan);
 setPreferredSize(new Dimension(300, 100));
 }
}
```

## LISTING D.8

*continued*

```

/**
 * Represents the listener for both check boxes.
 */
private class StyleListener implements ItemListener
{
 /**
 * Updates the style of the label font style.
 */
 public void itemStateChanged(ItemEvent event)
 {
 int style = Font.PLAIN;

 if (bold.isSelected())
 style = Font.BOLD;

 if (italic.isSelected())
 style += Font.ITALIC;

 saying.setFont(new Font("Helvetica", style, 36));
 }
}

```

The `itemStateChanged` method of the listener determines what the revised style should be now that one of the check boxes has changed state. It initially sets the style to be plain. Then each check box is consulted in turn using the `isSelected` method, which returns a boolean value. First, if the Bold check box is selected (checked), then the style is set to bold. Then, if the Italic check box is selected, the `ITALIC` constant is added to the style variable. Finally, the font of the label is set to a new font with its revised style.

Note that, given the way the listener is written in this program, it doesn't matter which check box was clicked to generate the event. The same listener processes both check boxes. It also doesn't matter whether the changed check box was toggled from selected to unselected or vice versa. The state of both check boxes is examined if either is changed.

## Radio Buttons

A *radio button* is used with other radio buttons to provide a set of mutually exclusive options. Unlike a check box, a radio button is not particularly useful by itself. It has meaning only when it is used with one or more other radio buttons. Only one option out of the group is valid. At any point in time, one and only one button of the group of radio buttons is selected (on). When a radio button from the group is pushed, the other button in the group that is currently on is automatically toggled off.

The term *radio buttons* comes from the way the buttons worked on an old-fashioned car radio. At any point, one button was pushed to specify the current choice of station; when another was pushed, the button that was in automatically popped out.

The `QuoteOptions` program, shown in Listing D.9, displays a label and a group of radio buttons. The radio buttons determine which quote is displayed in the label.

### KEY CONCEPT

Radio buttons operate as a group, providing a set of mutually exclusive options.

### LISTING D.9

```
import javax.swing.JFrame;

/**
 * QuoteOptions.java
 *
 * Demonstrates the use of radio buttons.
 */
public class QuoteOptions
{
 /**
 * Creates and presents the program frame.
 */
 public static void main(String[] args)
 {
 JFrame frame = new JFrame("Quote Options");
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

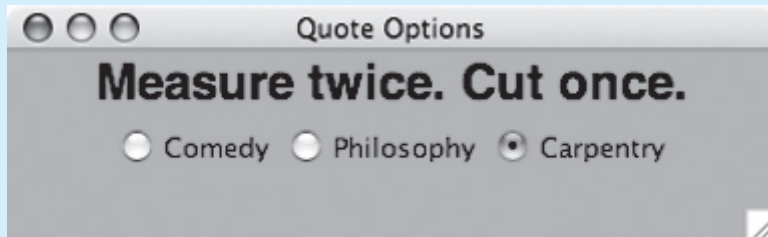
 frame.getContentPane().add(new QuoteOptionsPanel());

 frame.pack();
 frame.setVisible(true);
 }
}
```

## LISTING D.9

*continued*

## DISPLAY



Because only one of the quotes can be displayed at a time, the use of radio buttons is appropriate. For example, if the Comedy radio button is selected, the comedy quote is displayed in the label. If the Philosophy button is then pressed, the Comedy radio button is automatically toggled off, and the comedy quote is replaced by a philosophical one.

The `QuoteOptionsPanel` class, shown in Listing D.10, sets up and displays the GUI components. A radio button is represented by the `JRadioButton` class. Because the radio buttons in a set work together, the `ButtonGroup` class is used to define a set of related radio buttons.

## LISTING D.10

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

/**
 * QuoteOptionsPanel.java
 *
 * Demonstrates the use of radio buttons.
 */
public class QuoteOptionsPanel extends JPanel
{
 private JLabel quote;
 private JRadioButton comedy, philosophy, carpentry;
 private String comedyQuote, philosophyQuote, carpentryQuote;
```

## LISTING D.10

*continued*

```
/**
 * Sets up a panel with a label and a set of radio buttons
 * that control its text.
 */
public QuoteOptionsPanel()
{
 comedyQuote = "Take my wife, please.";
 philosophyQuote = "I think, therefore I am.";
 carpentryQuote = "Measure twice. Cut once.";

 quote = new JLabel(comedyQuote);
 quote.setFont(new Font("Helvetica", Font.BOLD, 24));

 comedy = new JRadioButton("Comedy", true);
 comedy.setBackground(Color.green);
 philosophy = new JRadioButton("Philosophy");
 philosophy.setBackground(Color.green);
 carpentry = new JRadioButton("Carpentry");
 carpentry.setBackground(Color.green);

 ButtonGroup group = new ButtonGroup();
 group.add(comedy);
 group.add(philosophy);
 group.add(carpentry);

 QuoteListener listener = new QuoteListener();
 comedy.addActionListener(listener);
 philosophy.addActionListener(listener);
 carpentry.addActionListener(listener);

 add(quote);
 add(comedy);
 add(philosophy);
 add(carpentry);

 setBackground(Color.green);
 setPreferredSize(new Dimension(300, 100));
}

/**
 * Represents the listener for all radio buttons
 */
```

## LISTING D.10

*continued*

```

private class QuoteListener implements ActionListener
{
 /**
 * Sets the text of the label depending on which radio
 * button was pressed.
 */
 public void actionPerformed(ActionEvent event)
 {
 Object source = event.getSource();

 if (source == comedy)
 quote.setText (comedyQuote);
 else
 if (source == philosophy)
 quote.setText (philosophyQuote);
 else
 quote.setText (carpentryQuote);
 }
}

```

Note that each button is added to the button group, and also that each button is added individually to the panel. A `ButtonGroup` object is not a container to organize and display components; it is simply a way to define the group of radio buttons that work together to form a set of dependent options. The `ButtonGroup` object ensures that the currently selected radio button is turned off when another in the group is selected.

A radio button produces an action event when it is selected. The `actionPerformed` method of the listener first retrieves the source of the event using the `getSource` method and then compares it to each of the three radio buttons in turn. Depending on which button was selected, the text of the label is set to the appropriate quote.

Note that unlike push buttons, both check boxes and radio buttons are *toggle buttons*, which means that at any time, they are either on or off. Independent options (choose any combination) are controlled with check boxes. Dependent options (choose one of a set) are controlled with radio buttons. If there is only one

option to be managed, a check box can be used by itself. As we mentioned earlier, a radio button makes sense only in conjunction with one or more other radio buttons.

Also note that check boxes and radio buttons produce different types of events. A check box produces an item event, and a radio button produces an action event. The use of different event types is related to the differences in button functionality. A check box produces an event when it is selected or deselected, and the listener could make the distinction if desired. A radio button, on the other hand, produces an event only when it is selected (the currently selected button from the group is deselected automatically).

## Sliders

A *slider* is a GUI component that allows the user to specify a numeric value within a bounded range. A slider can be presented either vertically or horizontally and can have optional tick marks and labels indicating the range of values.

A program called `SlideColor` is shown in Listing D.11. This program presents three sliders that control the RGB components of a color. The color specified by the values of the sliders is shown in a square that is displayed to the right of the sliders. Using RGB values to represent color is discussed in Appendix C.

### KEY CONCEPT

A slider lets the user specify a numeric value within a bounded range.

### LISTING D.11

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

/**
 * SlideColor.java
 *
 * Demonstrates the use slider components.
 */
public class SlideColor
{
 /**
 * Presents a frame with a control panel and a panel that
 * changes color as the sliders are adjusted.
 */
}
```



**LISTING D.11** *continued*

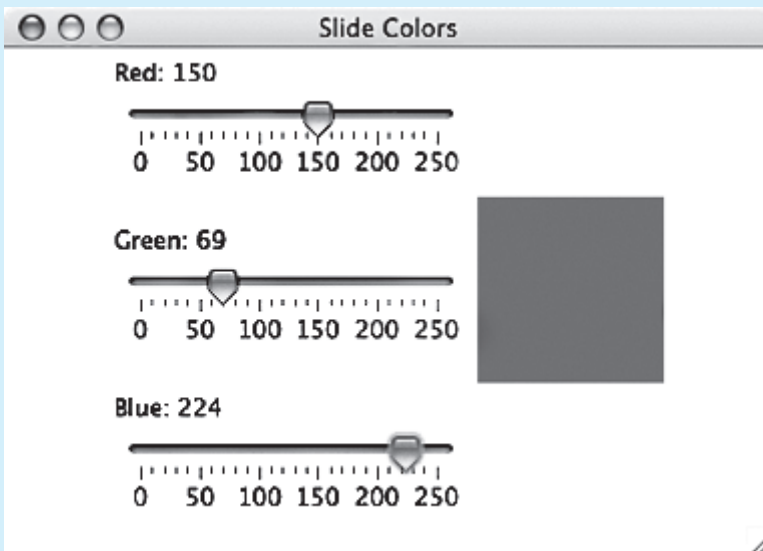
```

public static void main(String[] args)
{
 JFrame frame = new JFrame("Slide Colors");
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

 frame.getContentPane().add(new SlideColorPanel());

 frame.pack();
 frame.setVisible(true);
}
}

```

**DISPLAY**

The `SlideColorPanel` class shown in Listing D.12 is a panel used to display the three sliders and the color panel. Each slider is created from the `JSlider` class, which accepts four parameters. The first determines the orientation of the slider using one of two `JSlider` constants (`HORIZONTAL` or `VERTICAL`). The second and third parameters specify the maximum and minimum values of the slider,

which are set to 0 and 255, respectively, for each of the sliders in the example. The last parameter of the `JSlider` constructor specifies the slider's initial value. In our example, the initial value of each slider is 0, which puts the slider knob to the far left when the program initially executes.

The `JSlider` class has several methods that allow the programmer to tailor the look of a slider. Major tick marks can be set at specific intervals using the `setMajorTickSpacing` method. Intermediate minor tick marks can be set using the `setMinorTickSpacing` method. Neither is displayed, however, unless the `setPaintTicks` method, with a parameter of `true`, is invoked as well.

### LISTING D.12

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

/**
 * SlideColorPanel.java
 *
 * Represents the slider control panel for the SlideColor program.
 */
public class SlideColorPanel extends JPanel
{
 private JPanel controls, colorPanel;
 private JSlider rSlider, gSlider, bSlider;
 private JLabel rLabel, gLabel, bLabel;

 /**
 * Sets up the sliders and their labels, aligning them along
 * their left edge using a box layout.
 */
 public SlideColorPanel()
 {
 rSlider = new JSlider(JSlider.HORIZONTAL, 0, 255, 0);
 rSlider.setMajorTickSpacing(50);
 rSlider.setMinorTickSpacing(10);
 rSlider.setPaintTicks(true);
 rSlider.setPaintLabels(true);
 rSlider.setAlignmentX(Component.LEFT_ALIGNMENT);
 }
}
```

## LISTING D.12

*continued*

```

gSlider = new JSlider(JSlider.HORIZONTAL, 0, 255, 0);
gSlider.setMajorTickSpacing(50);
gSlider.setMinorTickSpacing(10);
gSlider.setPaintTicks(true);
gSlider.setPaintLabels(true);
gSlider.setAlignmentX(Component.LEFT_ALIGNMENT);

bSlider = new JSlider(JSlider.HORIZONTAL, 0, 255, 0);
bSlider.setMajorTickSpacing(50);
bSlider.setMinorTickSpacing(10);
bSlider.setPaintTicks(true);
bSlider.setPaintLabels(true);
bSlider.setAlignmentX(Component.LEFT_ALIGNMENT);

SliderListener listener = new SliderListener();
rSlider.addChangeListener(listener);
gSlider.addChangeListener(listener);
bSlider.addChangeListener(listener);

rLabel = new JLabel("Red: 0");
rLabel.setAlignmentX(Component.LEFT_ALIGNMENT);
gLabel = new JLabel("Green: 0");
gLabel.setAlignmentX(Component.LEFT_ALIGNMENT);
bLabel = new JLabel("Blue: 0");
bLabel.setAlignmentX(Component.LEFT_ALIGNMENT);

controls = new JPanel();
BoxLayout layout = new BoxLayout(controls, BoxLayout.Y_AXIS);
controls.setLayout(layout);
controls.add(rLabel);
controls.add(rSlider);
controls.add(Box.createRigidArea(new Dimension(0, 20)));
controls.add(gLabel);
controls.add(gSlider);
controls.add(Box.createRigidArea(new Dimension(0, 20)));
controls.add(bLabel);
controls.add(bSlider);

colorPanel = new JPanel();
colorPanel.setPreferredSize(new Dimension(100, 100));
colorPanel.setBackground(new Color(0, 0, 0));

add(controls);
add(colorPanel);
}

```

## LISTING D.12

*continued*

```
/**
 * Represents the listener for all three sliders.
 */
private class SliderListener implements ChangeListener
{
 private int red, green, blue;

 /**
 * Gets the value of each slider, then updates the labels and
 * the color panel.
 */
 public void stateChanged(ChangeEvent event)
 {
 red = rSlider.getValue();
 green = gSlider.getValue();
 blue = bSlider.getValue();

 rLabel.setText("Red: " + red);
 gLabel.setText("Green: " + green);
 bLabel.setText("Blue: " + blue);

 colorPanel.setBackground(new Color(red, green, blue));
 }
}
```

Labels indicating the value of the major tick marks are displayed if indicated by a call to the `setPaintLabels` method.

Note that in this example, the major tick spacing is set to 50. Starting at 0, each increment of 50 is labeled. The last label is therefore 250, even though the slider value can reach 255.

A slider produces a *change event*, indicating that the position of the slider and the value it represents have changed. The `ChangeListener` interface contains a single method called `stateChanged`. In the `SlideColor` program, the same listener object is used for all three sliders. In the `stateChanged` method, which is called whenever any of the sliders is adjusted, the value of each slider is obtained, the

labels of all three are updated, and the background color of the color panel is revised. It is actually only necessary to update one of the labels (the one whose corresponding slider changed). However, the effort to determine which slider was adjusted is not warranted. It's easier—and probably more efficient—to update all three labels each time. Another alternative is to have a unique listener for each slider, although that extra coding effort is not needed, either.

A slider is often a good choice when a large range of values is possible but strictly bounded on both ends. Compared to alternatives such as a text field, sliders convey more information to the user and eliminate input errors.

## Combo Boxes

A *combo box* allows the user to select one of several options from a “drop-down” menu. When the user presses a combo box using the mouse, a list of options is displayed from which the user can choose. The current choice is displayed in the combo box. A combo box is defined by the `JComboBox` class.

### KEY CONCEPT

A combo box provides a drop-down menu of options.

A combo box can be either editable or uneditable. By default, a combo box is uneditable. Changing the value of an uneditable combo box can be accomplished only by selecting an item from the list. If the combo box is editable, however, the user can change the value by either selecting an item from the list or typing a particular value into the combo box area.

The options in a combo box list can be established in one of two ways. We can create an array of strings and pass it into the constructor of the `JComboBox` class. Alternatively, we can use the `addItem` method to add an item to the combo box after it has been created. An item in a `JComboBox` can also display an `ImageIcon` object, in addition to text or appearing by itself.

The `JukeBox` program shown in Listing D.13 demonstrates the use of a combo box. The user chooses a song to play, using the combo box, and then presses the Play button to begin playing the song. The Stop button can be pressed at any time to stop the song. Selecting a new song while one is playing also stops the current song.

### LISTING D.13

```
import javax.swing.*;

/**
 * JukeBox.java
 *
 * Demonstrates the use of a combo box.
 */
```

## LISTING D.13

*continued*

```
public class JukeBox
{
 /**
 * Creates and displays the controls for a juke box.
 */
 public static void main(String[] args)
 {
 JFrame frame = new JFrame("Java Juke Box");
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

 frame.getContentPane().add(new JukeBoxControls());

 frame.pack();
 frame.setVisible(true);
 }
}
```

## DISPLAY



The `JukeBoxControls` class shown in Listing D.14 is a panel that contains the components that make up the jukebox GUI. The constructor of the class also loads the audio clips that will be played. An audio clip is obtained first by creating a `URL` object that corresponds to the wav or au file that defines the clip. The first two parameters to the `URL` constructor should be "file" and "localhost", respectively, if the audio clip is stored on the same machine on which the program is

executing. Creating URL objects can potentially throw a checked exception; therefore, they are created in a `try` block. However, this program assumes that the audio clips will be loaded successfully and therefore does nothing if an exception is thrown.

Once created, the URL objects are used to create `AudioClip` objects using the static `newAudioClip` method of the `JApplet` class. The audio clips are stored in an array. The first entry in the array, at index 0, is set to `null`. This entry corresponds to the initial combo box option, which simply encourages the user to make a selection.

### LISTING D.14

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.applet.AudioClip;
import java.net.URL;

/**
 * JukeBoxControls.java
 *
 * Represents the control panel for the juke box.
 */
public class JukeBoxControls extends JPanel
{
 private JComboBox musicCombo;
 private JButton stopButton, playButton;
 private AudioClip[] music;
 private AudioClip current;

 /**
 * Sets up the GUI for the juke box.
 */
 public JukeBoxControls()
 {
 URL url1, url2, url3, url4, url5, url6;
 url1 = url2 = url3 = url4 = url5 = url6 = null;

 // Obtain and store the audio clips to play
 }
}
```

## LISTING D.14

*continued*

```
try
{
 url1 = new URL("file", "localhost", "westernBeat.wav");
 url2 = new URL("file", "localhost", "classical.wav");
 url3 = new URL("file", "localhost", "jeopardy.au");
 url4 = new URL("file", "localhost", "newAgeRhythm.wav");
 url5 = new URL("file", "localhost", "eightiesJam.wav");
 url6 = new URL("file", "localhost", "hitchcock.wav");
}
catch(Exception exception) {}

music = new AudioClip[7];
music[0] = null; // Corresponds to "Make a Selection..."
music[1] = JApplet.newAudioClip(url1);
music[2] = JApplet.newAudioClip(url2);
music[3] = JApplet.newAudioClip(url3);
music[4] = JApplet.newAudioClip(url4);
music[5] = JApplet.newAudioClip(url5);
music[6] = JApplet.newAudioClip(url6);

// Create the list of strings for the combo box options
String[] musicNames = {"Make A Selection...", "Western Beat",
 "Classical Melody", "Jeopardy Theme", "New Age Rhythm",
 "Eighties Jam", "Alfred Hitchcock's Theme"};

musicCombo = new JComboBox(musicNames);
musicCombo.setBackground(Color.cyan);

// Set up the buttons
playButton = new JButton("Play", new ImageIcon("play.gif"));
playButton.setBackground(Color.cyan);
stopButton = new JButton("Stop", new ImageIcon("stop.gif"));
stopButton.setBackground(Color.cyan);

// Set up this panel
setPreferredSize(new Dimension(250, 100));
setBackground(Color.cyan);
add(musicCombo);
add(playButton);
add(stopButton);
musicCombo.addActionListener(new ComboListener());
stopButton.addActionListener(new ButtonListener());
playButton.addActionListener(new ButtonListener());
current = null;
}
```



## LISTING D.14

*continued*

```

/**
 * Represents the action listener for the combo box.
 */
private class ComboListener implements ActionListener
{
 /**
 * Stops playing the current selection (if any) and resets
 * the current selection to the one chosen.
 */
 public void actionPerformed(ActionEvent event)
 {
 if (current != null)
 current.stop();

 current = music[musicCombo.getSelectedIndex()];
 }
}

/**
 * Represents the action listener for both control buttons.
 */
private class ButtonListener implements ActionListener
{
 /**
 * Stops the current selection (if any) in either case. If
 * the play button was pressed, start playing it again.
 */
 public void actionPerformed(ActionEvent event)
 {
 if (current != null)
 current.stop();

 if (event.getSource() == playButton)
 if (current != null)
 current.play();
 }
}
}

```

The list of songs that is displayed in the combo box is defined in an array of strings. The first entry of the array will appear in the combo box by default and is often used to direct the user. We must take care that the rest of the program does not try to use that option as a valid song.

This program also shows the ability of a push button to display an image. In this example, the Play and Stop buttons are displayed with both a text label and an image icon.

A combo box generates an action event whenever the user makes a selection from it. The `JukeBox` program uses one action listener class for the combo box and another for both of the push buttons. They could have been combined, using code to distinguish which component fired the event.

The `actionPerformed` method of the `ComboListener` class is executed when a selection is made from the combo box. The current audio selection that is playing, if any, is stopped. The current clip is then updated to reflect the new selection. Note that the audio clip is not immediately played at that point. The way this program is designed, the user must press the Play button to hear the new selection.

The `actionPerformed` method of the `ButtonListener` class is executed when either of the buttons is pushed. The current audio selection that is playing, if any, is stopped. If the Stop button was pressed, the task is complete. If the Play button was pressed, the current audio selection is played again from the beginning.

## Timers

A *timer*, created from the `Timer` class of the `javax.swing` package, can be thought of as a GUI component. However, unlike other components, it does not have a visual representation that appears on the screen. Instead, as the name implies, it helps us manage an activity over time.

A timer object generates an action event at regular intervals. To perform an animation, we can set up a timer to generate an action event periodically, and then update the animation graphics in the action listener. The methods of the `Timer` class are shown in Figure D.1.

### KEY CONCEPT

A timer generates action events at regular intervals and can be used to control an animation.

```

Timer (int delay, ActionListener listener)
 Constructor: Creates a timer that generates an action event at
 regular intervals, specified by the delay. The event will be handled
 by the specified listener.

void addActionListener (ActionListener listener)
 Adds an action listener to the timer.

boolean isRunning ()
 Returns true if the timer is running.

void setDelay (int delay)
 Sets the delay of the timer.

void start ()
 Starts the timer, causing it to generate action events.

void stop ()
 Stops the timer, causing it to stop generating action events.

```

**FIGURE D.1** Some methods of the `Timer` class

The program shown in Listing D.15 displays the image of a smiling face that seems to glide across the program window at an angle, bouncing off the window edges (even though that's hard to appreciate from a screen shot).

### LISTING D.15

```

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

/**
 * Rebound.java
 *
 * Demonstrates an animation and the use of the Timer class.
 */

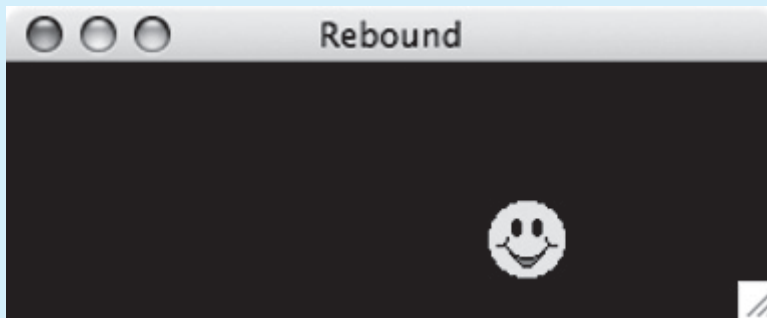
```

**LISTING D.15***continued*

```
public class Rebound
{
 /**
 * Displays the main frame of the program.
 */
 public static void main(String[] args)
 {
 JFrame frame = new JFrame("Rebound");
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

 frame.getContentPane().add(new ReboundPanel());

 frame.pack();
 frame.setVisible(true);
 }
}
```

**DISPLAY**

The constructor of the `ReboundPanel` class, shown in Listing D.16, creates a `Timer` object. The first parameter to the `Timer` constructor is the delay in milliseconds. The second parameter to the constructor is the listener that handles the action events of the timer. The `ReboundPanel` constructor also sets up the initial position for the image and the number of pixels it will move, in both the vertical and horizontal directions, each time the image is redrawn.

## LISTING D.16

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

/**
 * ReboundPanel.java
 *
 * Represents the primary panel for the Rebound program.
 */
public class ReboundPanel extends JPanel
{
 private final int WIDTH = 300, HEIGHT = 100;
 private final int DELAY = 20, IMAGE_SIZE = 35;

 private ImageIcon image;
 private Timer timer;
 private int x, y, moveX, moveY;

 /**
 * Sets up the panel, including the timer for the animation.
 */
 public ReboundPanel()
 {
 timer = new Timer(DELAY, new ReboundListener());

 image = new ImageIcon("happyFace.gif");

 x = 0;
 y = 40;
 moveX = moveY = 3;

 setPreferredSize(new Dimension(WIDTH, HEIGHT));
 setBackground(Color.black);
 timer.start();
 }

 /**
 * Draws the image in the current location.
 */
 public void paintComponent(Graphics page)
```

## LISTING D.16

*continued*

```
{
 super.paintComponent(page);
 image.paintIcon(this, page, x, y);
}

/**
 * Represents the action listener for the timer.
 */
private class ReboundListener implements ActionListener
{
 /**
 * Updates the position of the image and possibly the direction
 * of movement whenever the timer fires an action event.
 */
 public void actionPerformed(ActionEvent event)
 {
 x += moveX;
 y += moveY;

 if (x = WIDTH-IMAGE_SIZE)
 moveX = moveX * -1;

 if (y = HEIGHT-IMAGE_SIZE)
 moveY = moveY * -1;

 repaint();
 }
}
}
```

The `actionPerformed` method of the listener updates the current `x` and `y` coordinate values and then checks to see whether those values cause the image to “run into” the edge of the panel. If so, the movement is adjusted so that the image will make future moves in the opposite direction horizontally, vertically, or both. Note that this calculation takes the image size into account.

After updating the coordinate values, the `actionPerformed` method calls `repaint` to force the component (in this case, the panel) to repaint itself. The call to `repaint` eventually causes the `paintComponent` method to be called, which repaints the image in the new location.

The speed of the animation in this program is a function of two factors: the pause between the action events, and the distance the image is shifted each time. In this example, the timer is set to generate an action event every 20 milliseconds, and the image is shifted 3 pixels each time it is updated. You can experiment with these values to change the speed of the animation. The goal should be to create the illusion of movement that is pleasing to the eye.

## D.3 Layout Managers

### KEY CONCEPT

Every container is managed by a layout manager, which determines how components are visually presented.

### KEY CONCEPT

When changes occur, the components in a container reorganize themselves according to the layout manager's policy.

As we mentioned earlier in this chapter, every container is managed by an object called a *layout manager* that determines how the components in the container are arranged visually. The layout manager is consulted when needed, such as when the container is resized or when a component is added to the container.

A layout manager determines the size and position of each component and may take many factors into account to do so. Every container has a default layout manager, although we can replace it if we prefer another one.

The table in Figure D.2 describes several of the predefined layout managers provided by the Java standard class library.

| Layout Manager | Description                                                                                |
|----------------|--------------------------------------------------------------------------------------------|
| Border Layout  | Organizes components into five areas (North, South, East, West, and Center).               |
| Box Layout     | Organizes components into a single row or column.                                          |
| Card Layout    | Organizes components into one area such that only one is visible at any time.              |
| Flow Layout    | Organizes components from left to right, starting new rows as necessary.                   |
| Grid Layout    | Organizes components into a grid of rows and columns.                                      |
| GridBag Layout | Organizes components into a grid of cells, allowing components to span more than one cell. |

FIGURE D.2 Some predefined Java layout managers

Every layout manager has its own particular properties and rules governing the layout of components. For some layout managers, the order in which you add the components affects their positioning, whereas others provide more specific control. Some layout managers take a component's preferred size or alignment into account, whereas others don't. To develop good GUIs in Java, it is important to become familiar with features and characteristics of various layout managers.

We can use the `setLayout` method of a container to change its layout manager. For example, the following code sets the layout manager of a `JPanel`, which has a flow layout by default, so that it uses a border layout instead.

```
JPanel panel = new JPanel();
panel.setLayout(new BorderLayout());
```

Let's explore some of these layout managers in more detail. We'll focus on the most popular layout managers at this point: flow, border, box, and grid. The class presented in Listing D.17 contains the `main` method of an application that demonstrates the use and effects of these layout managers.

### KEY CONCEPT

The layout manager for each container can be explicitly set.

### LISTING D.17

```
import javax.swing.*;

/**
 * LayoutDemo.java
 *
 * Demonstrates the use of flow, border, grid, and box layouts.
 */
public class LayoutDemo
{
 /**
 * Sets up a frame containing a tabbed pane. The panel on each
 * tab demonstrates a different layout manager.
 */
 public static void main(String[] args)
 {
 JFrame frame = new JFrame("Layout Manager Demo");
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```



## LISTING D.17

*continued*

```

JTabbedPane tp = new JTabbedPane();
tp.addTab("Intro", new IntroPanel());
tp.addTab("Flow", new FlowPanel());
tp.addTab("Border", new BorderPanel());
tp.addTab("Grid", new GridPanel());
tp.addTab("Box", new BoxPanel());

frame.getContentPane().add(tp);

frame.pack();
frame.setVisible(true);
}
}

```

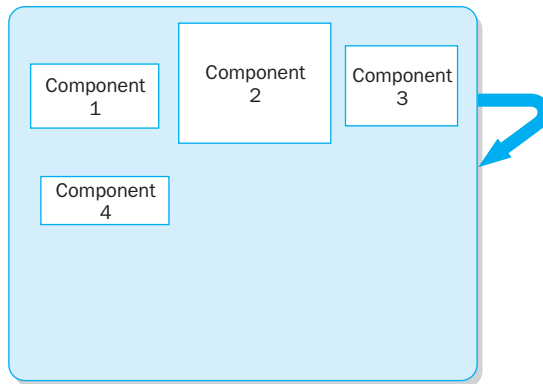
The `LayoutDemo` program introduces the use of a *tabbed pane*, a container that allows the user to select (by clicking on a tab) which of several panes is currently visible. A tabbed pane is defined by the `JTabbedPane` class. The `addTab` method creates a tab, specifying the name that appears on the tab and the component to be displayed on that pane when it achieves focus by being “brought to the front” and made visible to the user.

Interestingly, there is an overlap in the functionality provided by tabbed panes and the card layout manager. Similar to the tabbed pane, a card layout allows several layers to be defined, and only one of those layers is displayed at any given point. However, a container managed by a card layout can be adjusted only under program control, whereas tabbed panes allow the user to indicate directly which tab should be displayed.

In this example, each tab of the tabbed pane contains a panel that is controlled by a different layout manager. The first tab simply contains a panel with an introductory message, as shown in Listing D.18. As we explore each layout manager in more detail, we examine the class that defines the corresponding panel of this program and discuss its visual effect.

## Flow Layout

*Flow layout* is one of the easiest layout managers to use. As we’ve mentioned, the `JPanel` class uses flow layout by default. Flow layout puts as many components as possible on a row, at their preferred size. When a component cannot fit on a



**FIGURE D.3** Flow layout puts as many components as possible on a row

row, it is put on the next row. As many rows as needed are added to fit all components that have been added to the container. Figure D.3 depicts a container governed by a flow layout manager.

### LISTING D.18

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

/**
 * IntroPanel.java
 *
 * Represents the introduction panel for the LayoutDemo program.
 */
public class IntroPanel extends JPanel
{
 /**
 * Sets up this panel with two labels.
 */
 public IntroPanel()
 {
 setBackground(Color.green);
 }
}
```

## LISTING D.18

*continued*

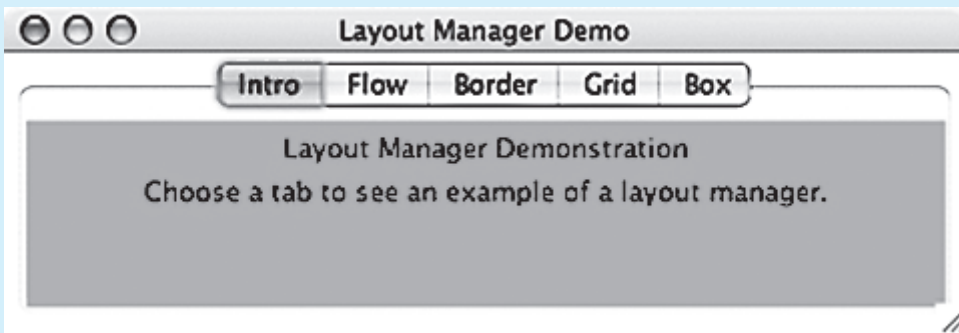
```

JLabel l1 = new JLabel("Layout Manager Demonstration");
JLabel l2 = new JLabel("Choose a tab to see an example of " +
 "a layout manager.");

add(l1);
add(l2);
}
}

```

## DISPLAY



The class in Listing D.19 represents the panel that demonstrates a flow layout in the `LayoutDemo` program. It explicitly sets the layout to be a flow layout (although in this case that is unnecessary, because `JPanel` defaults to flow layout). The buttons are then created and added to the panel.

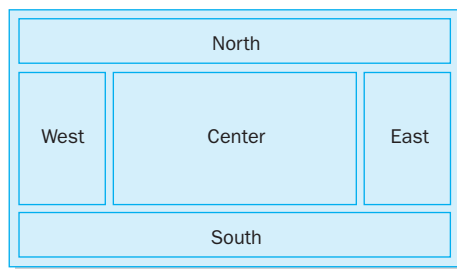
The size of each button is made large enough to accommodate the size of the label that is put on it. Flow layout puts as many of these buttons as possible on one row within the panel and then starts putting components on another row. When the size of the frame is widened (by dragging the lower-right corner with the mouse, for example), the panel grows as well, and more buttons can fit on a row. When the frame is resized, the layout manager is consulted and the components are reorganized automatically. The display in Listing D.19 shows two screen shots of the window with different sizes.

The constructor of the `FlowLayout` class is overloaded to allow the programmer to tailor the characteristics of the layout manager. Within each row, components are either centered, left aligned, or right aligned. The alignment defaults to

centered. The horizontal and vertical gap size between components can be specified when the layout manager is created. The `FlowLayout` class also has methods to set the alignment and gap sizes after the layout manager is created.

## Border Layout

A *border layout* has five areas to which components can be added: North, South, East, West, and Center. The areas have a particular positional relationship to each other, as shown in Figure D.4.



**FIGURE D.4** Border layout organizes components in five areas

### LISTING D.19

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

/**
 * FlowPanel.java
 *
 * Represents the panel in the LayoutDemo program that demonstrates
 * the flow layout manager.
 */
public class FlowPanel extends JPanel
{
 /**
 * Sets up this panel with some buttons to show how flow layout
 * affects their position.
 */
}
```

## LISTING D.19

*continued*

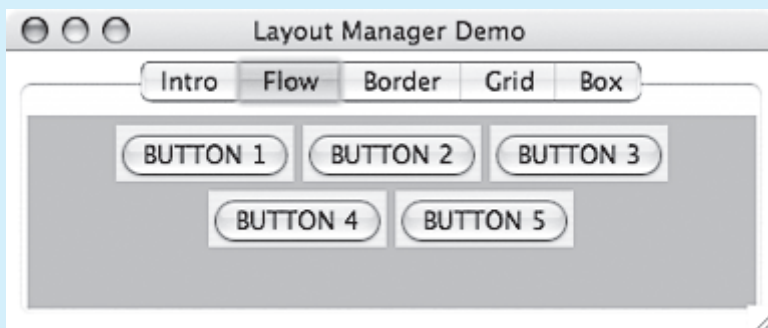
```
public FlowPanel()
{
 setLayout(new FlowLayout());

 setBackground(Color.green);

 JButton b1 = new JButton("BUTTON 1");
 JButton b2 = new JButton("BUTTON 2");
 JButton b3 = new JButton("BUTTON 3");
 JButton b4 = new JButton("BUTTON 4");
 JButton b5 = new JButton("BUTTON 5");

 add(b1);
 add(b2);
 add(b3);
 add(b4);
 add(b5);
}
}
```

## DISPLAY



The four outer areas become as big as needed in order to accommodate the component they contain. If no components are added to the North, South, East, or West areas, these areas do not take up any room in the overall layout. The Center area expands to fill any available space.

A particular container might use only a few areas, depending on the functionality of the system. For example, a program might use only the Center, South, and West areas. This versatility makes border layout a very useful layout manager.

The `add` method for a container governed by a border layout takes as its first parameter the component to be added. The second parameter indicates the area to which it is added. The area is specified using constants defined in the `BorderLayout` class. Listing D.20 shows the panel used by the `LayoutDemo` program to demonstrate the border layout.

### LISTING D.20

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

/**
 * BorderLayout.java
 *
 * Represents the panel in the LayoutDemo program that demonstrates
 * the border layout manager.
 */
public class BorderLayout extends JPanel
{
 /**
 * Sets up this panel with a button in each area of a border
 * layout to show how it affects their position, shape, and size.
 */
 public BorderLayout()
 {
 setLayout(new BorderLayout());

 setBackground(Color.green);

 JButton b1 = new JButton("BUTTON 1");
 JButton b2 = new JButton("BUTTON 2");
 JButton b3 = new JButton("BUTTON 3");
 }
}
```

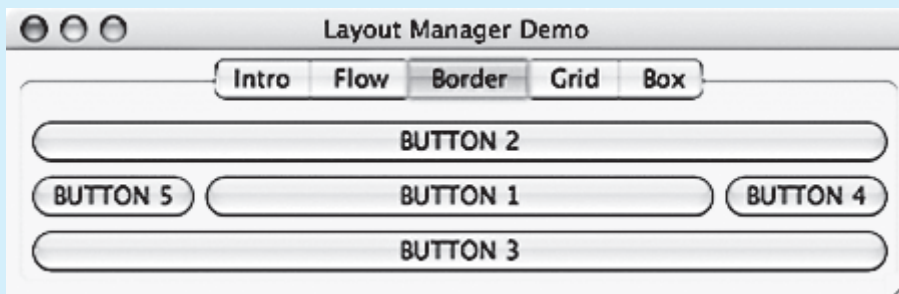
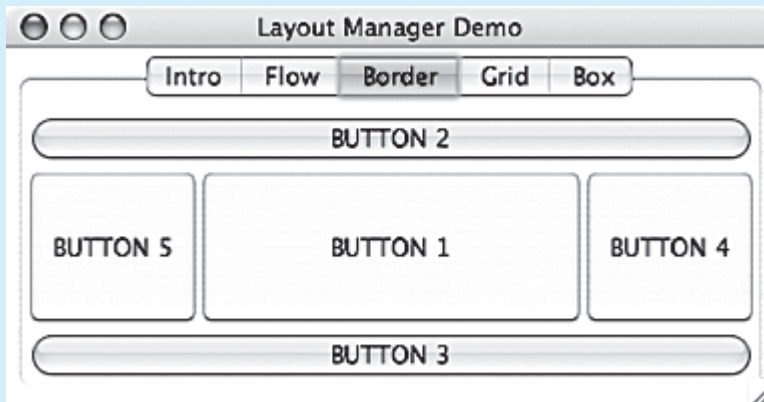
**LISTING D.20** *continued*

```

 JButton b4 = new JButton("BUTTON 4");
 JButton b5 = new JButton("BUTTON 5");

 add(b1, BorderLayout.CENTER);
 add(b2, BorderLayout.NORTH);
 add(b3, BorderLayout.SOUTH);
 add(b4, BorderLayout.EAST);
 add(b5, BorderLayout.WEST);
 }
}

```

**DISPLAY**

In the `BorderPanel` class constructor, the layout manager of the panel is explicitly set to be border layout. The buttons are then created and added to specific panel areas. By default, each button is made wide enough to accommodate its label and tall enough to fill the area to which it has been assigned. As the frame (and the panel) is resized, the size of each button adjusts as needed, with the button in the Center area filling any unused space.

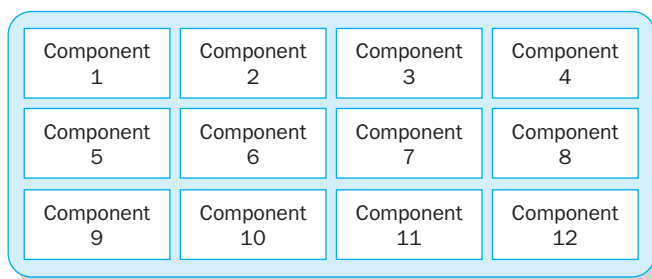
Each area in a border layout displays only one component. That is, only one component is added to each area of a given border layout. A common error is to add two components to a particular area of a border layout, in which case the first component added is replaced by the second, and only the second is seen when the container is displayed. To add multiple components to an area within a border layout, you must first add the components to another container, such as a `JPanel`, and then add the panel to the area.

Note that although the panel used to display the buttons has a green background, no green is visible in the display for Listing D.20. By default there are no horizontal or vertical gaps between the areas of a border layout. These gaps can be set with an overloaded constructor or with explicit methods of the `BorderLayout` class. If the gaps are increased, the underlying panel will show through.

## Grid Layout

A *grid layout* presents a container's components in a rectangular grid of rows and columns. One component is placed in each grid cell, and all cells are the same size. Figure D.5 shows the general organization of a grid layout.

The number of rows and columns in a grid layout is established using parameters to the constructor when the layout manager is created. The class in Listing D.21 shows the panel used by the `LayoutDemo` program to demonstrate a grid



**FIGURE D.5** Grid layout creates a rectangular grid of equal-size cells



layout. It specifies that the panel should be managed using a grid of two rows and three columns.

As buttons are added to the container, they fill the grid (by default) from left to right and from top to bottom. There is no way to explicitly assign a component to a particular location in the grid, other than the order in which they are added to the container.

The size of each cell is determined by the container's overall size. When the container is resized, all of the cells change size proportionally to fill the container.

## LISTING D.21

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

/**
 * GridPanel.java
 *
 * Represents the panel in the LayoutDemo program that demonstrates
 * the grid layout manager.
 */
public class GridPanel extends JPanel
{
 /**
 * Sets up this panel with some buttons to show how grid
 * layout affects their position, shape, and size.
 */
 public GridPanel()
 {
 setLayout(new GridLayout(2, 3));

 setBackground(Color.green);

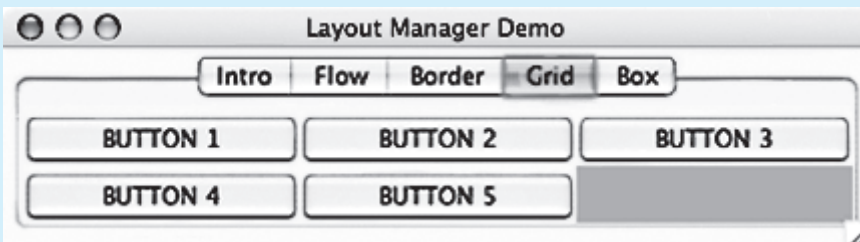
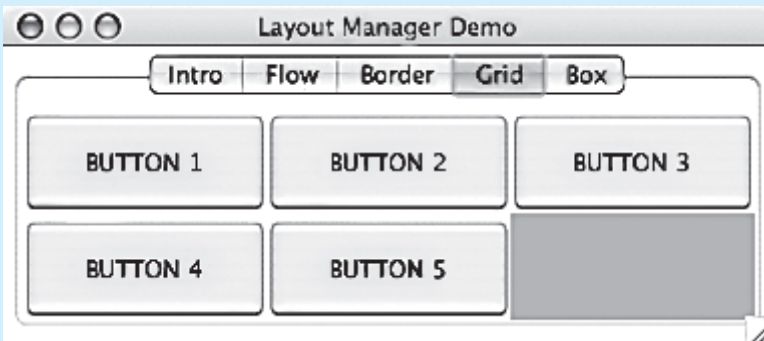
 JButton b1 = new JButton("BUTTON 1");
 JButton b2 = new JButton("BUTTON 2");
 JButton b3 = new JButton("BUTTON 3");
 JButton b4 = new JButton("BUTTON 4");
 JButton b5 = new JButton("BUTTON 5");
 }
}
```

## LISTING D.21

*continued*

```
 add(b1);
 add(b2);
 add(b3);
 add(b4);
 add(b5);
 }
}
```

## DISPLAY

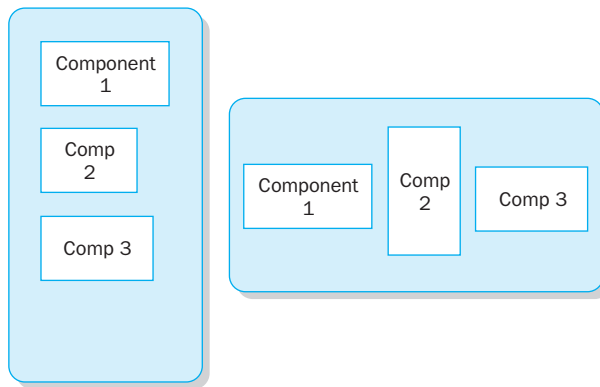


If the value used to specify either the number of rows or the number of columns is zero, the grid expands as needed in that dimension to accommodate the number of components added to the container. The values for the number of rows and the number of columns cannot both be zero.

By default, there are no horizontal and vertical gaps between the grid cells. The gap sizes can be specified using an overloaded constructor or with the appropriate `GridLayout` methods.

## Box Layout

A *box layout* organizes components either vertically or horizontally, in one row or one column, as shown in Figure D.6. It is easy to use, yet when combined with other box layouts, it can produce complex GUI designs similar to those that can be accomplished with a *grid bag layout*, which in general is far more difficult to master.



**FIGURE D.6** Box layout organizes components either vertically or horizontally

When a `BoxLayout` object is created, we specify that it will follow either the x axis (horizontal) or the y axis (vertical), using constants defined in the `BoxLayout` class. Unlike other layout managers, the constructor of the `BoxLayout` class takes as its first parameter the component that it will govern. Therefore, a new `BoxLayout` object must be created for each component. Listing D.22 shows the panel used by the `LayoutDemo` program to demonstrate the box layout.

Components in containers governed by a box layout are organized (top to bottom or left to right) in the order in which they are added to the container.

There are no gaps between the components in a box layout. Unlike previous layout managers we've explored, a box layout does not have a specific vertical or horizontal gap that can be specified for the entire container. Instead, we can add invisible components to the container that take up space between other components. The `Box` class, which is also part of the Java standard class library, contains static methods that can be used to create these invisible components.

The two types of invisible components used in the `BoxPanel` class are *rigid areas*, which have a fixed size, and *glue*, which specifies where excess space in a container should go. A rigid area is created using the `createRigidArea` method of the `Box` class and takes a `Dimension` object as a parameter to define the size of the invisible area. Glue is created using the `createHorizontalGlue` method or the `createVerticalGlue` method, as appropriate.

Note that in our example, the space between buttons separated by a rigid area remains constant even when the container is resized. Glue, on the other hand, expands or contracts as needed to fill the space.

### LISTING D.22

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

/**
 * BoxPanel.java
 *
 * Represents the panel in the LayoutDemo program that demonstrates
 * the box layout manager.
 */
public class BoxPanel extends JPanel
{
 /**
 * Sets up this panel with some buttons to show how a vertical
 * box layout (and invisible components) affects their position.
 */
 public BoxPanel()
 {
 setLayout(new BorderLayout(this, BorderLayout.Y_AXIS));

 setBackground(Color.green);

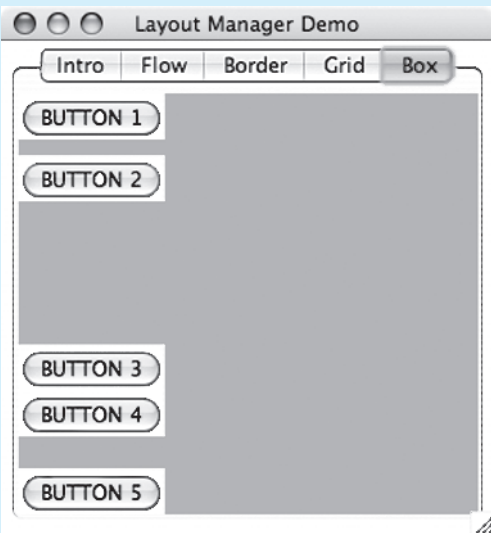
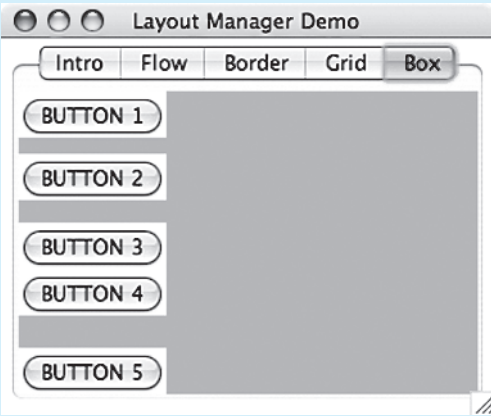
 JButton b1 = new JButton("BUTTON 1");
 JButton b2 = new JButton("BUTTON 2");
 JButton b3 = new JButton("BUTTON 3");
 JButton b4 = new JButton("BUTTON 4");
 JButton b5 = new JButton("BUTTON 5");

 add(b1);
 add(Box.createRigidArea(new Dimension(0, 10)));
 add(b2);
 add(Box.createVerticalGlue());
 add(b3);
 add(b4);
 add(Box.createRigidArea(new Dimension(0, 20)));
 add(b5);
 }
}
```

LISTING D.22

*continued*

DISPLAY



A box layout—more than most of the other layout managers—respects the alignments and the minimum, maximum, and preferred sizes of the components it governs. Therefore, setting the characteristics of the components that go into the container is another way to tailor the visual effect.

## Containment Hierarchies

The way components are grouped into containers, and the way those containers are nested within each other, establishes the *containment hierarchy* for a GUI. The interplay between the containment hierarchy and the layout managers of the containers involved dictates the overall visual effect of the GUI.

For any Java program, there is generally one primary container, called a top-level container, such as a frame or applet. The top-level container of a program often contains one or more other containers, such as panels. These panels may contain other panels to organize the other components as desired.

Keep in mind that each container can have its own tailored layout manager. The final appearance of a GUI is a function of the layout managers chosen for each of the containers and the design of the containment hierarchy. Many combinations are possible, and there is rarely a single best option. We should be guided by the desired system goals and general GUI design guidelines.

When changes are made that might affect the visual layout of the components in a program, the layout managers of each container are consulted in turn. The changes in one may affect another. These changes ripple through the containment hierarchy as needed.

### KEY CONCEPT

A GUI's appearance is a function of the containment hierarchy and the layout managers of each container.

## D.4 Mouse and Key Events

---

In addition to events that are generated when the user interacts with a component, there are events that are fired when the user interacts with the computer's mouse and keyboard. We can design a program to capture and respond to these as well.

### Mouse Events

Java divides the events generated by the user interacting with the mouse into two categories: *mouse events* and *mouse motion events*. The tables in Figure D.7 define these events.

| Mouse Event    | Description                                                                        |
|----------------|------------------------------------------------------------------------------------|
| mouse pressed  | The mouse button is pressed down.                                                  |
| mouse released | The mouse button is released.                                                      |
| mouse clicked  | The mouse button is pressed down and released without moving the mouse in between. |
| mouse entered  | The mouse pointer is moved onto (over) a component.                                |
| mouse exited   | The mouse pointer is moved off of a component.                                     |

| Mouse Motion Event | Description                                                |
|--------------------|------------------------------------------------------------|
| mouse moved        | The mouse is moved.                                        |
| mouse dragged      | The mouse is moved while the mouse button is pressed down. |

FIGURE D.7 Mouse events and mouse motion events

When you click the mouse button over a Java GUI component, three events are generated: one when the mouse button is pushed down (*mouse pressed*) and two when it is let up (*mouse released* and *mouse clicked*). A mouse click is defined as pressing and releasing the mouse button in the same location. If you press the mouse button down, move the mouse, and then release the mouse button, a mouse clicked event is not generated.

A component will generate a *mouse entered* event when the mouse pointer passes into its graphical space. Likewise, it generates a *mouse exited* event when the mouse pointer leaves.

#### KEY CONCEPT

Moving the mouse and clicking the mouse button generate events to which a program can respond.

Mouse motion events, as the name implies, occur while the mouse is in motion. The *mouse moved* event indicates simply that the mouse is in motion. The *mouse dragged* event is generated when the user has pressed the mouse button down and moved the mouse without releasing the button. Mouse motion events are generated many times, very quickly, while the mouse is in motion.

In a specific situation, we may care about only one or two mouse events. What we listen for depends on what we are trying to accomplish.

The `Coordinates` program shown in Listing D.23 responds to one mouse event. Specifically, it draws a green dot at the location of the mouse pointer whenever the mouse button is pressed, and displays those coordinates. Keep in mind

that (as discussed in Appendix C) the coordinate system in Java has the origin in the upper-left corner of a component (such as a panel), with x coordinates increasing to the right and y coordinates increasing downward.

The `CoordinatesPanel` class, shown in Listing D.24, keeps track of the (x, y) coordinates at which the user has pressed the mouse button most recently. The `getX` and `getY` methods of the `MouseEvent` object return the x and y coordinates of the location where the mouse event occurred.

### LISTING D.23

```
import javax.swing.JFrame;

/**
 * Coordinates.java
 *
 * Demonstrates mouse events.
 */
public class Coordinates
{
 /**
 * Creates and displays the application frame.
 */
 public static void main(String[] args)
 {
 JFrame frame = new JFrame("Coordinates");
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

 frame.getContentPane().add(new CoordinatesPanel());

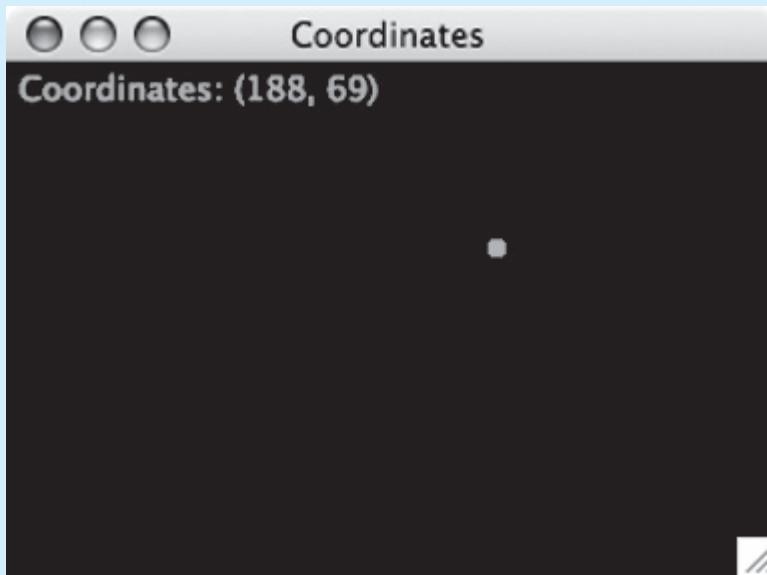
 frame.pack();
 frame.setVisible(true);
 }
}
```



## LISTING D.23

*continued*

## DISPLAY



## LISTING D.24

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

/**
 * CoordinatesPanel.java
 *
 * Represents the primary panel for the Coordinates program.
 */
public class CoordinatesPanel extends JPanel
{
 private final int SIZE = 6; // diameter of dot
```

## LISTING D.24

*continued*

```
private int x = 50, y = 50; // coordinates of mouse press

/**
 * Constructor: Sets up this panel to listen for mouse events.
 */
public CoordinatesPanel()
{
 addMouseListener(new CoordinatesListener());

 setBackground(Color.black);
 setPreferredSize(new Dimension(300, 200));
}

/**
 * Draws all of the dots stored in the list.
 */
public void paintComponent(Graphics page)
{
 super.paintComponent(page);

 page.setColor(Color.green);

 page.fillOval(x, y, SIZE, SIZE);

 page.drawString("Coordinates:(" + x + ", " + y + ")", 5, 15);
}

/**
 * Represents the listener for mouse events.
 */
private class CoordinatesListener implements MouseListener
{
 /**
 * Adds the current point to the list of points and redraws
 * the panel whenever the mouse button is pressed.
 */
 public void mousePressed(MouseEvent event)
```

## LISTING D.24

*continued*

```

 {
 x = event.getX();
 y = event.getY();
 repaint();
 }

/**
 * Provide empty definitions for unused event methods.
 */
public void mouseClicked(MouseEvent event) {}
public void mouseReleased(MouseEvent event) {}
public void mouseEntered(MouseEvent event) {}
public void mouseExited(MouseEvent event) {}
}
}

```

The listener for the mouse pressed event implements the `MouseListener` interface. The panel invokes the `mousePressed` method each time the user presses down on the mouse button while it is over the panel.

Note that, unlike the listener interfaces that we've used in previous examples that contain one method each, the `MouseListener` interface contains five methods. For this program, the only event in which we are interested is the mouse pressed event. Therefore, the only method in which we have any interest is the `mousePressed` method. However, implementing an interface means we must provide definitions for all methods in the interface. Therefore, we provide empty methods corresponding to the other events. When those events are generated, the empty methods are called, but no code is executed. At the end of this section, we discuss a technique for creating listeners that enables us to avoid creating such empty methods.

**KEY CONCEPT**

A listener may have to provide empty method definitions for unheeded events to satisfy the interface.

Let's look at an example that responds to two mouse-oriented events. The `RubberLines` program shown in Listing D.25 draws a line between two points.

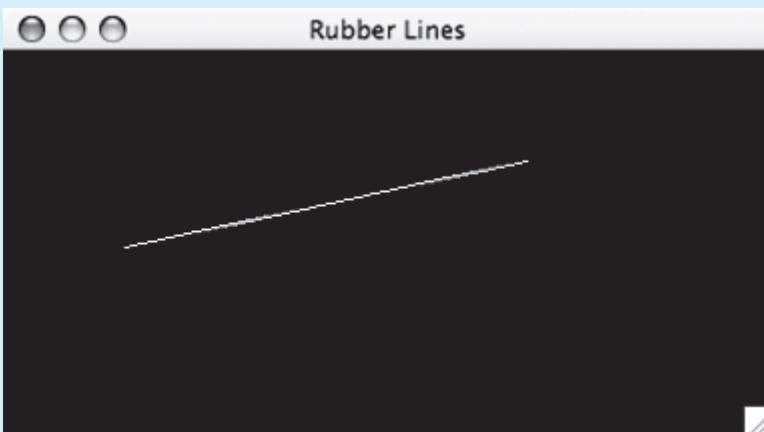
**LISTING D.25**

```
import javax.swing.JFrame;

/**
 * RubberLines.java
 *
 * Demonstrates mouse events and rubberbanding.
 */
public class RubberLines
{
 /**
 * Creates and displays the application frame.
 */
 public static void main(String[] args)
 {
 JFrame frame = new JFrame("Rubber Lines");
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

 frame.getContentPane().add(new RubberLinesPanel());

 frame.pack();
 frame.setVisible(true);
 }
}
```

**DISPLAY**

The first point is determined by the location at which the mouse is first pressed down. The second point changes as the mouse is dragged while the mouse button is held down. When the button is released, the line remains fixed between the first and second points. When the mouse button is pressed again, a new line is started.

The `RubberLinesPanel` class is shown in Listing D.26. Because we need to listen for both a mouse pressed event and a mouse dragged event, we need a listener that responds both to mouse events and to mouse motion events.

## LISTING D.26

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

/**
 * RubberLinesPanel.java
 *
 * Represents the primary drawing panel for the RubberLines program.
 */
public class RubberLinesPanel extends JPanel
{
 private Point point1 = null, point2 = null;

 /**
 * Constructor: Sets up this panel to listen for mouse events.
 */
 public RubberLinesPanel()
 {
 LineListener listener = new LineListener();
 addMouseListener(listener);
 addMouseMotionListener(listener);

 setBackground(Color.black);
 setPreferredSize(new Dimension(400, 200));
 }

 /**
 * Draws the current line from the initial mouse-pressed point to
 * the current position of the mouse.
 */
}
```

## LISTING D.26

*continued*

```
public void paintComponent(Graphics page)
{
 super.paintComponent(page);

 page.setColor(Color.yellow);
 if (point1 != null && point2 != null)
 page.drawLine(point1.x, point1.y, point2.x, point2.y);
}

/**
 * Represents the listener for all mouse events.
 */
private class LineListener implements MouseListener,
 MouseMotionListener
{
 /**
 * Captures the initial position at which the mouse button is
 * pressed.
 */
 public void mousePressed(MouseEvent event)
 {
 point1 = event.getPoint();
 }

 /**
 * Gets the current position of the mouse as it is dragged and
 * redraws the line to create the rubberband effect.
 */
 public void mouseDragged(MouseEvent event)
 {
 point2 = event.getPoint();
 repaint();
 }

 /**
 * Provide empty definitions for unused event methods.
 */
}
```

## LISTING D.26

*continued*

```

 public void mouseClicked(MouseEvent event) {}
 public void mouseReleased(MouseEvent event) {}
 public void mouseEntered(MouseEvent event) {}
 public void mouseExited(MouseEvent event) {}
 public void mouseMoved(MouseEvent event) {}
}
}

```

**KEY CONCEPT**

Rubberbanding is the graphical effect caused when a shape seems to expand as the mouse is dragged.

Note that the listener class in this example implements both the `MouseListener` interface and the `MouseMotionListener` interface. It must therefore implement all methods of both interfaces. The two methods of interest, `mousePressed` and `mouseDragged`, are implemented to accomplish our goals, and the other methods are given empty definitions to satisfy the interface contract.

When the `mousePressed` method is called, the variable `point1` is set. Then, as the mouse is dragged, the variable `point2` is continually reset and the panel repainted. Therefore, the line is constantly being redrawn as the mouse is dragged, giving the appearance that one line is being stretched between a fixed point and a moving point. This effect is called *rubberbanding* and is common in graphical programs.

The starting and ending points of the line are stored as `Point` objects. The `Point` class is defined in the `java.awt` package and encapsulates the `x` and `y` values of a two-dimensional coordinate.

Note that in the `RubberLinesPanel` constructor, the listener object is added to the panel twice: once as a mouse listener and once as a mouse motion listener. The method called to add the listener must correspond to the object passed as the parameter. In this case, we had one object that served as a listener for both categories of events. We could have had two listener classes if desired: one listening for mouse events and one listening for mouse motion events. A component can have multiple listeners for various event categories.

**Key Events**

A *key event* is generated when a keyboard key is pressed. Key events allow a program to respond immediately to the user while he or she is typing or pressing other keyboard keys, such as the arrow keys. If key events are being processed, the program can respond as soon as the key is pressed; there is no need to wait for the Enter key to be pressed or for some other component (such as a button) to be activated.

The `Direction` program shown in Listing D.27 responds to key events. An image of an arrow is displayed, and the image moves across the screen as the arrow keys are pressed. Actually, four different images are used, one for the arrow pointing in each of the primary directions (up, down, right, and left).

The `DirectionPanel` class, shown in Listing D.28, represents the panel on which the arrow image is displayed. The constructor loads the four arrow images, one of which is always considered to be the current image (the one displayed). The current image is set based on the arrow key that was most recently pressed. For example, if the up arrow is pressed, the image with the arrow pointing up is displayed. If an arrow key is continually pressed, the appropriate image “moves” in the appropriate direction.

### KEY CONCEPT

Key events allow a program to respond immediately to the user pressing keyboard keys.

### LISTING D.27

```
import javax.swing.JFrame;

/**
 * Direction.java
 *
 * Demonstrates key events.
 */
public class Direction
{
 /**
 * Creates and displays the application frame.
 */
 public static void main(String[] args)
 {
 JFrame frame = new JFrame("Direction");
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

 frame.getContentPane().add(new DirectionPanel());

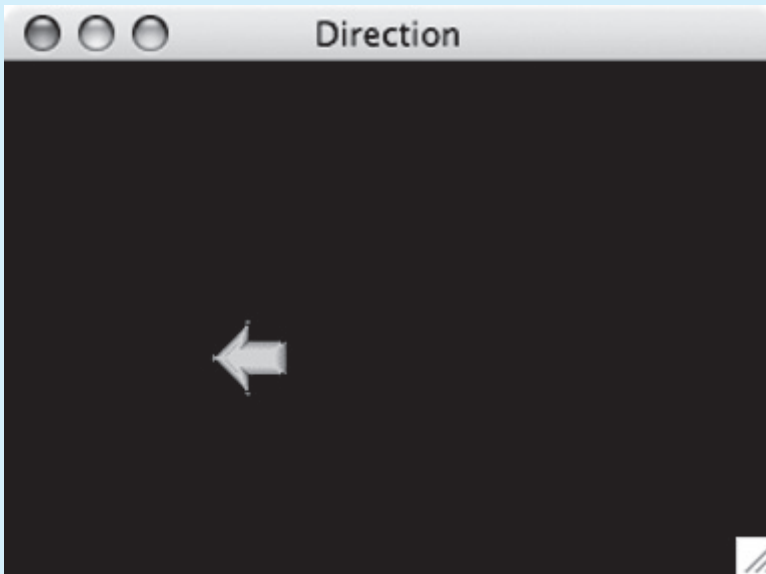
 frame.pack();
 frame.setVisible(true);
 }
}
```



## LISTING D.27

*continued*

## DISPLAY



## LISTING D.28

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

/**
 * DirectionPanel.java
 *
 * Represents the primary display panel for the Direction program.
 */
public class DirectionPanel extends JPanel
```

## LISTING D.28

*continued*

```
{
 private final int WIDTH = 300, HEIGHT = 200;
 private final int JUMP = 10; // increment for image movement

 private final int IMAGE_SIZE = 31;

 private ImageIcon up, down, right, left, currentImage;
 private int x, y;

 /**
 * Constructor: Sets up this panel and loads the images.
 */
 public DirectionPanel()
 {
 addKeyListener(new DirectionListener());

 x = WIDTH / 2;
 y = HEIGHT / 2;

 up = new ImageIcon("arrowUp.gif");
 down = new ImageIcon("arrowDown.gif");
 left = new ImageIcon("arrowLeft.gif");
 right = new ImageIcon("arrowRight.gif");

 currentImage = right;

 setBackground(Color.black);
 setPreferredSize(new Dimension(WIDTH, HEIGHT));
 setFocusable(true);
 }

 /**
 * Draws the image in the current location.
 */
 public void paintComponent(Graphics page)
 {
 super.paintComponent(page);
 currentImage.paintIcon(this, page, x, y);
 }
}
```

## LISTING D.28

*continued*

```

/**
 * Represents the listener for keyboard activity.
 */
private class DirectionListener implements KeyListener
{
 /**
 * Responds to the user pressing arrow keys by adjusting the
 * image and image location accordingly.
 */
 public void keyPressed(KeyEvent event)
 {
 switch(event.getKeyCode())
 {
 case KeyEvent.VK_UP:
 currentImage = up;
 y -= JUMP;
 break;
 case KeyEvent.VK_DOWN:
 currentImage = down;
 y += JUMP;
 break;
 case KeyEvent.VK_LEFT:
 currentImage = left;
 x -= JUMP;
 break;
 case KeyEvent.VK_RIGHT:
 currentImage = right;
 x += JUMP;
 break;
 }

 repaint();
 }

 /**
 * Provide empty definitions for unused event methods.
 */
 public void keyTyped(KeyEvent event) {}
 public void keyReleased(KeyEvent event) {}
}
}

```

The arrow images are managed as `ImageIcon` objects. In this example, the image is drawn using the `paintIcon` method each time the panel is repainted. The `paintIcon` method takes four parameters: a component to serve as an image observer, the graphics context on which the image will be drawn, and the (x, y) coordinates where the image is drawn. An *image observer* is a component that serves to manage image loading; in this case we use the panel as the image observer.

The private inner class called `KeyListener` is set up to respond to key events. It implements the `KeyListener` interface, which defines three methods that we can use to respond to keyboard activity. Figure D.8 lists these methods.

```
void keyPressed (KeyEvent event)
 Called when a key is pressed.

void keyReleased (KeyEvent event)
 Called when a key is released.

void keyTyped (KeyEvent event)
 Called when a pressed key or key combination produces
 a key character.
```

**FIGURE D.8** The methods of the `KeyListener` interface

Specifically, the `Direction` program responds to key pressed events. Because the listener class must implement all methods defined in the interface, we provide empty methods for the other events.

The `KeyEvent` object passed to the `keyPressed` method of the listener can be used to determine which key was pressed. In the example, we call the `getKeyCode` method of the event object to get a numeric code that represents the key that was pressed. We use a `switch` statement to determine which key was pressed and to respond accordingly. The `KeyEvent` class contains constants that correspond to the numeric code that is returned from the `getKeyCode` method. If any key other than an arrow key is pressed, it is ignored.

Key events fire whenever a key is pressed, but most systems enable the concept of *key repetition*. That is, when a key is pressed and held down, it's as if that key is being pressed repeatedly and quickly. Key events are generated in the same way. In the `Direction` program, the user can hold down an arrow key and watch the image move across the screen quickly.

The component that generates key events is the one that currently has the keyboard focus. Usually the keyboard focus is held by the primary “active” component. A component usually gets the keyboard focus when the user clicks on it with the mouse. The call to the `setFocusable` method in the panel constructor sets the keyboard focus to the panel.

The `Direction` program sets no boundaries for the arrow image, so it can be moved out of the visible window and then moved back in if desired. You could add code to the listener to stop the image when it reaches one of the window boundaries. This modification is left as a programming project.

## Extending Adapter Classes

In previous event-based examples, we’ve created the listener classes by implementing a particular listener interface. For instance, to create a class that listens for mouse events, we created a listener class that implements the `MouseListener` interface. As we saw in the previous examples in this section, a listener interface often contains event methods that are not important to a particular program, in which case we provided empty definitions to satisfy the interface requirement.

An alternative technique for creating a listener class is to use inheritance and extend an *event adapter class*. Each listener interface that contains more than one method has a corresponding adapter class that already contains empty definitions for all of the methods in the interface. To create a listener, we can derive a new listener class from the appropriate adapter class and override any event methods in which we are interested. Using this technique, we no longer need to provide empty definitions for unused methods.

### KEY CONCEPT

A listener class can be created by deriving it from an event adapter class.

The `MouseAdapter` class, for instance, implements the `MouseListener` interface and provides empty method definitions for the five mouse event methods (`mousePressed`, `mouseClicked`, and so on). Therefore, you can create a mouse listener class by extending the `MouseAdapter` class instead of implementing the `MouseListener` interface directly. The new listener class inherits the empty definitions and therefore doesn’t need to define them.

Because of inheritance, we now have a choice when it comes to creating event listeners. We can implement an event listener interface, or we can extend an event adapter class. This is a design decision that should be considered carefully. The best technique depends on the situation. Inheritance is discussed further in Appendix B.

```
static String showInputDialog (Object msg)
 Displays a dialog box containing the specified message and an input text field.
 The contents of the text field are returned.

static int showConfirmDialog (Component parent, Object msg)
 Displays a dialog box containing the specified message and Yes/No button
 options. If the parent component is null, the box is centered on the screen.

static void showMessageDialog (Component parent, Object msg)
 Displays a dialog box containing the specified message. If the parent
 component is null, the box is centered on the screen.
```

FIGURE D.9 Some methods of the `JOptionPane` class

## D.5 Dialog Boxes

A component called a *dialog box* can be helpful to assist in GUI processing. A dialog box is a graphical window that pops up on top of any currently active window so that the user can interact with it. A dialog box can serve a variety of purposes, such as conveying some information, confirming an action, or allowing the user to enter some information. Usually a dialog box has a solitary purpose, and the user's interaction with it is brief.

The Swing package of the Java class library contains a class called `JOptionPane` that simplifies the creation and use of basic dialog boxes. Figure D.9 lists some of the methods of `JOptionPane`.

The basic formats for a `JOptionPane` dialog box fall into three categories. A *message dialog box* simply displays an output string. An *input dialog box* presents a prompt and a single input text field into which the user can enter one string of data. A *confirm dialog box* presents the user with a simple yes-or-no question.

Let's look at a program that uses each of these types of dialog boxes. Listing D.29 shows a program that first presents the user with an input dialog box that requests the user to enter an integer. After the user presses the OK button on the input dialog box, a second dialog box (this time a message dialog box) appears, informing the user whether the number entered was even or odd. After the user dismisses that box, a third dialog box appears, to determine whether the user would like to test another number. If the user presses the button labeled Yes, the series of dialog boxes repeats. Otherwise, the program terminates.

## LISTING D.29

```
import javax.swing.JOptionPane;

/**
 * EvenOdd.java
 *
 * Demonstrates the use of the JOptionPane class.
 */
public class EvenOdd
{
 /**
 * Determines if the value input by the user is even or odd.
 * Uses multiple dialog boxes for user interaction.
 */
 public static void main(String[] args)
 {
 String numStr, result;
 int num, again;

 do
 {
 numStr = JOptionPane.showInputDialog("Enter an integer: ");

 num = Integer.parseInt(numStr);

 result = "That number is " + ((num%2 == 0) ? "even" : "odd");

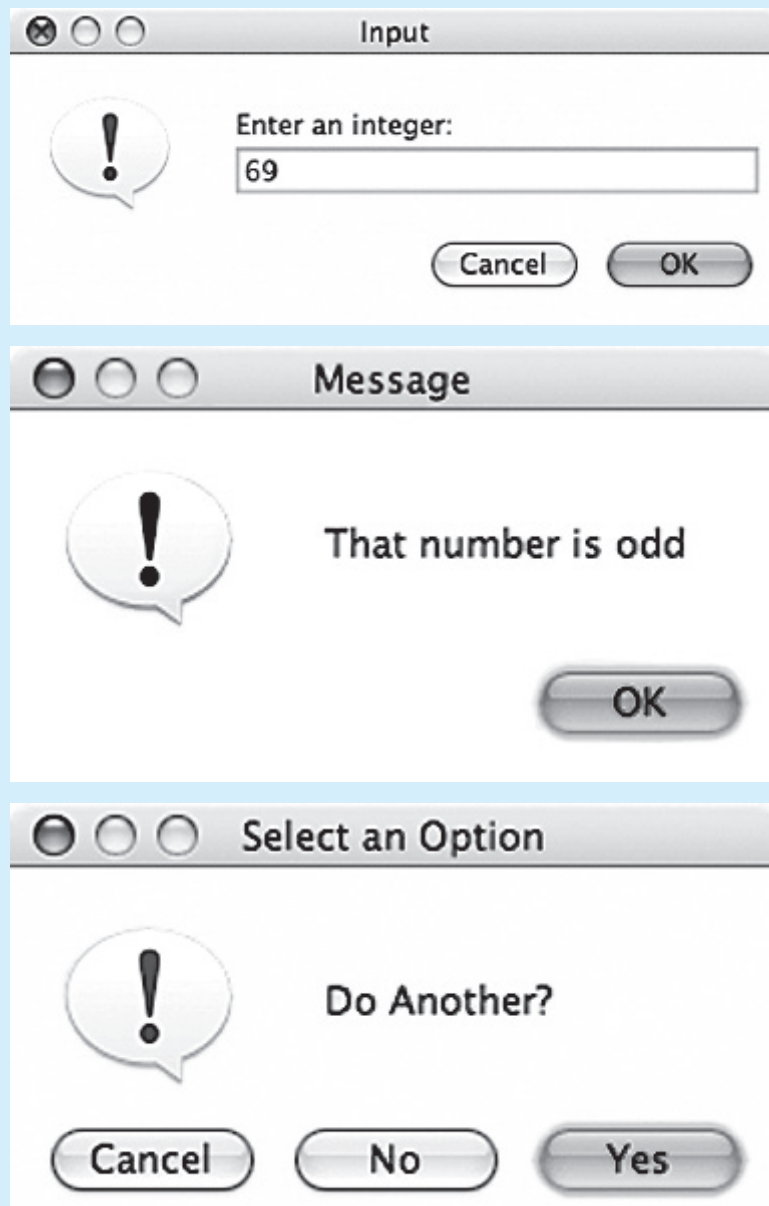
 JOptionPane.showMessageDialog(null, result);

 again = JOptionPane.showConfirmDialog(null, "Do Another?");
 }
 while (again == JOptionPane.YES_OPTION);
 }
}
```

## LISTING D.29

*continued*

## DISPLAY





The first parameter to the `showMessageDialog` and the `showConfirmDialog` methods specifies the governing parent component for the dialog box. Using a null reference as this parameter causes the dialog box to appear centered on the screen.

Many of the `JOptionPane` methods allow the programmer to tailor the contents of the dialog box. Furthermore, the `showOptionDialog` method can be used to create dialog boxes that combine characteristics of the three basic formats for more elaborate interactions.

Dialog boxes should be used only when the immediate attention of the user is necessary. A program that constantly has new windows popping up for different interactions is annoying to the user.

## File Choosers

A *file chooser* is a specialized dialog box that allows the user to select a file from a disk or other storage medium. You have probably run many programs that allow you to open a file, such as when you are specifying which file to open in a word processing program. The need to specify a file occurs so often that the `JFileChooser` class was made part of the Java standard class library for just that purpose.

### KEY CONCEPT

A file chooser allows the user to browse a disk and select a file to be processed.

The program shown in Listing D.30 uses a `JFileChooser` dialog box to select a file. This program also demonstrates the use of another GUI component, a *text area*, which is similar to a text field but can display multiple lines of text at one time. In this example, after the user selects a file using the file chooser dialog box, the text contained in that file is displayed in a text area.

The file chooser dialog box is displayed when the `showOpenDialog` method is invoked. It automatically presents the list of files contained in a particular directory. The user can use the controls on the dialog box to navigate to other directories, change the way the files are viewed, and specify which types of files are displayed.

The `showOpenDialog` method returns an integer representing the status of the operation, which can be checked against constants defined in the `JFileChooser` class. In this program, if a file was not selected (perhaps by pressing the Cancel button), a default message is displayed in the text area. If the user chose a file, it is opened and its contents are read using the `Scanner` class. Note that this program assumes that the selected file contains text. It does not catch any exceptions, so if the user selects an inappropriate file, the program will terminate when the exception is thrown.

**LISTING D.30**

```
import java.util.Scanner;
import java.io.*;
import javax.swing.*;

/**
 * DisplayFile.java
 *
 * Demonstrates the use of a file chooser and a text area.
 */
public class DisplayFile
{
 /**
 * Opens a file chooser dialog, reads the selected file and
 * loads it into a text area.
 */
 public static void main(String[] args) throws IOException
 {
 JFrame frame = new JFrame("Display File");
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

 JTextArea ta = new JTextArea(20, 30);
 JFileChooser chooser = new JFileChooser();

 int status = chooser.showOpenDialog(null);

 if (status != JFileChooser.APPROVE_OPTION)
 ta.setText("No File Chosen");
 else
 {
 File file = chooser.getSelectedFile();
 Scanner scan = new Scanner(file);

 String info = "";
 while (scan.hasNext())
 info += scan.nextLine() + "\n";

 ta.setText(info);
 }
 }
}
```

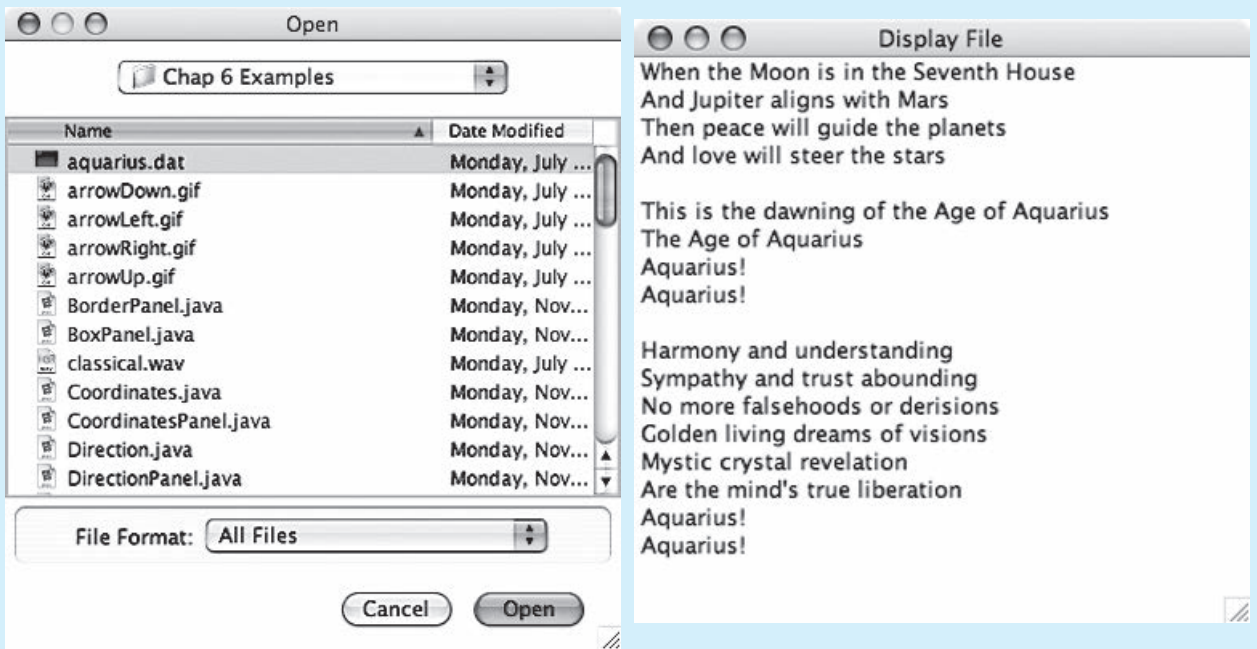
LISTING D.30 *continued*

```

 frame.getContentPane().add(ta);
 frame.pack();
 frame.setVisible(true);
 }
}

```

## DISPLAY



A text area component is defined by the `JTextArea` class. In this program, we pass two parameters to its constructor, specifying the size of the text area in terms of the number of characters (rows and columns) it should display. The text to display is set using the `setText` method.

A text area component, like a text field, can be set so that it is either editable or noneditable. The user can change the contents of an editable text area by clicking on the text area and typing with the mouse. If the text area is noneditable, it is used to display text only. By default, a `JTextArea` component is editable.

A `JFileChooser` component makes it easy to allow users to specify a specific file to use. Another specialized dialog box—one that allows the user to choose a color—is discussed in the next section.

## Color Choosers

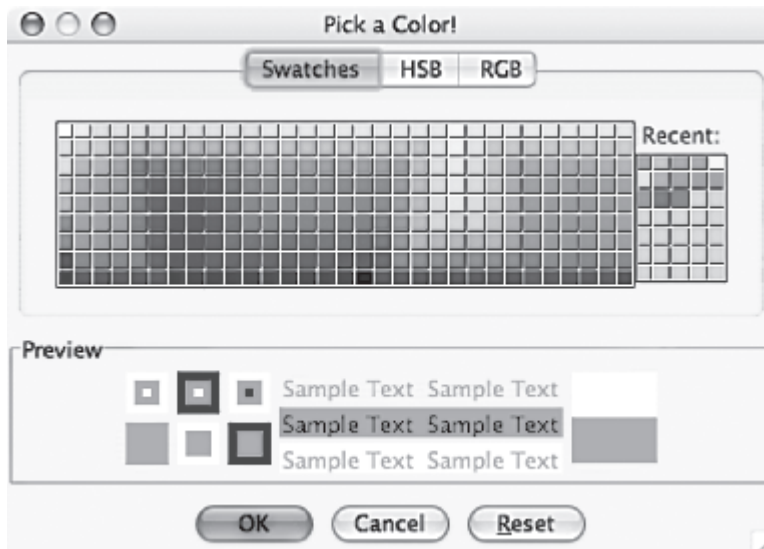
In many situations we may want to give the user of a program the ability to choose a color. We could accomplish this in various ways. For instance, we could provide a list of colors using a set of radio buttons. However, with the wide variety of colors available, it's nice to have an easier and more flexible technique to accomplish this common task. A specialized dialog box, often referred to as a *color chooser*, is a graphical component that serves this purpose.

The `JColorChooser` class represents a color chooser. It can be used to display a dialog box that lets the user click on a color of choice from a palette presented for that purpose. The user can also specify a color by using RGB values or other color representation techniques. Invoking the static `showDialog` method of the `JColorChooser` class causes the color chooser dialog box to appear. The parameters to that method specify the parent component for the dialog box, the title that appears in the dialog box frame, and the initial color showing in the color chooser.

Figure D.10 shows a color chooser dialog box.

### KEY CONCEPT

A color chooser allows the user to select a color from a palette or by using RGB values.



**FIGURE D.10** A color chooser dialog box

## D.6 Some Important Details

There are a variety of small but important details that can add considerable value to the interface of a program. Some enhance the visual effect, and others provide shortcuts to make the user more productive. Let's examine some of them now.

### Borders

Java provides the ability to put a border around any Swing component. A border is not itself a component; rather, it defines how the edge of any component should be drawn and has an important effect on the design of a GUI. A border provides visual cues to how GUI components are organized and can be used to give titles to components. Figure D.11 lists the predefined borders in the Java standard class library.

#### KEY CONCEPT

Borders can be applied to components to group objects and focus attention.

The `BorderFactory` class is useful for creating borders for components. It has many methods for creating specific types of borders. A border is applied to a component by using the component's `setBorder` method.

The program in Listing D.31 demonstrates several types of borders. It simply creates several panels, sets a different border for each, and then displays them in a larger panel using a grid layout.

| Border          | Description                                                                              |
|-----------------|------------------------------------------------------------------------------------------|
| Empty Border    | Puts buffering space around the edge of a component, but otherwise has no visual effect. |
| Line Border     | A simple line surrounding the component.                                                 |
| Etched Border   | Creates the effect of an etched groove around a component.                               |
| Bevel Border    | Creates the effect of a component raised above the surface or sunken below it.           |
| Titled Border   | Includes a text title on or around the border.                                           |
| Matte Border    | Allows the size of each edge to be specified. Uses either a solid color or an image.     |
| Compound Border | A combination of two borders.                                                            |

FIGURE D.11 Component borders

**LISTING D.31**

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

/**
 * BorderDemo.java
 *
 * Demonstrates the use of various types of borders.
 */
public class BorderDemo
{
 /**
 * Creates several bordered panels and displays them.
 */
 public static void main(String[] args)
 {
 JFrame frame = new JFrame("Border Demo");
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

 JPanel panel = new JPanel();
 panel.setLayout(new GridLayout(0, 2, 5, 10));
 panel.setBorder(BorderFactory.createEmptyBorder(8, 8, 8, 8));

 JPanel p1 = new JPanel();
 p1.setBorder(BorderFactory.createLineBorder(Color.red, 3));
 p1.add(new JLabel("Line Border"));
 panel.add(p1);

 JPanel p2 = new JPanel();
 p2.setBorder(BorderFactory.createEtchedBorder());
 p2.add(new JLabel("Etched Border"));
 panel.add(p2);

 JPanel p3 = new JPanel();
 p3.setBorder(BorderFactory.createRaisedBevelBorder());
 p3.add(new JLabel("Raised Bevel Border"));
 panel.add(p3);
 }
}
```

## LISTING D.31

*continued*

```
JPanel p4 = new JPanel();
p4.setBorder(BorderFactory.createLoweredBevelBorder());
p4.add(new JLabel("Lowered Bevel Border"));
panel.add(p4);

JPanel p5 = new JPanel();
p5.setBorder(BorderFactory.createTitledBorder("Title"));
p5.add(new JLabel("Titled Border"));
panel.add(p5);

JPanel p6 = new JPanel();
TitledBorder tb = BorderFactory.createTitledBorder("Title");
tb.setTitleJustification(TitledBorder.RIGHT);
p6.setBorder(tb);
p6.add(new JLabel("Titled Border(right)"));
panel.add(p6);

JPanel p7 = new JPanel();
Border b1 = BorderFactory.createLineBorder(Color.blue, 2);
Border b2 = BorderFactory.createEtchedBorder();
p7.setBorder(BorderFactory.createCompoundBorder(b1, b2));
p7.add(new JLabel("Compound Border"));
panel.add(p7);

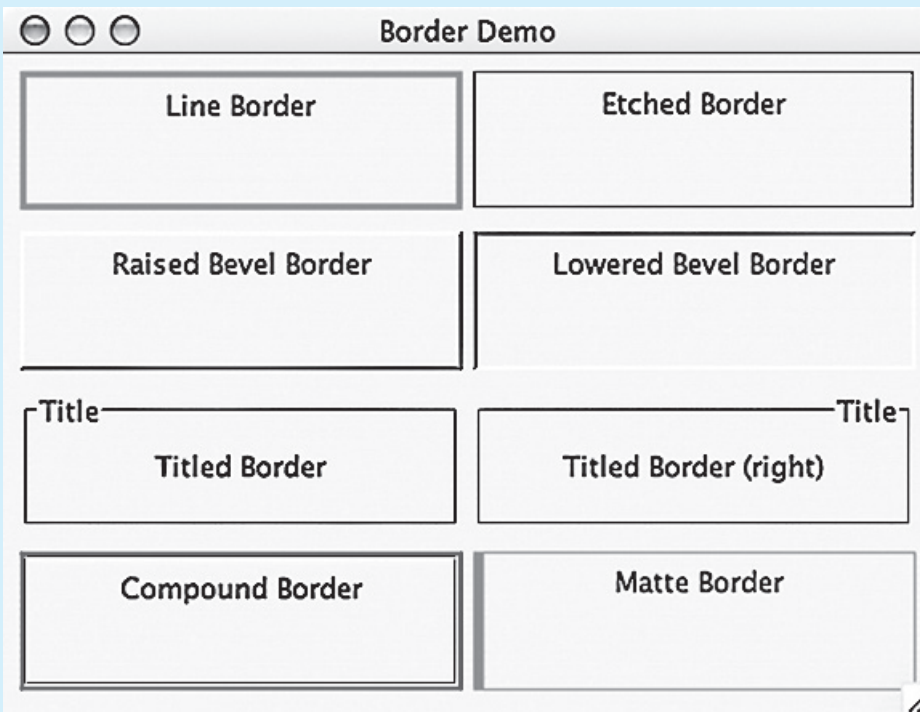
JPanel p8 = new JPanel();
Border mb = BorderFactory.createMatteBorder(1, 5, 1, 1, Color.red);
p8.setBorder(mb);
p8.add(new JLabel("Matte Border"));
panel.add(p8);

frame.getContentPane().add(panel);
frame.pack();
frame.setVisible(true);
}
}
```

## LISTING D.31

continued

## DISPLAY



Let's look at each type of border created in this program. An *empty border* is applied to the larger panel that holds all the others, to create a buffer of space around the outer edge of the frame. The sizes of the top, left, bottom, and right edges of the empty border are specified in pixels. The *line border* is created using a particular color and specifies the line thickness in pixels (3 in this case). The line thickness defaults to 1 pixel if left unspecified. The *etched border* created in this program uses default colors for the highlight and shadow of the etching, but both could be explicitly set if desired.

A *bevel border* can be either raised or lowered. The default coloring is used in this program, although the coloring of each aspect of the bevel can be tailored as



desired, including the outer highlight, inner highlight, outer shadow, and inner shadow. Each of these aspects could be a different color if desired.

A *titled border* places a title on or around the border. The default position for the title is on the border at the top-left edge. Using the `setTitleJustification` method of the `TitledBorder` class, one can set this position to many other places above, below, on, or to the left, right, or center of the border.

A *compound border* is a combination of two or more borders. The example in this program creates a compound border using a line border and an etched border. The `createCompoundBorder` method accepts two borders as parameters and makes the first parameter the outer border and the second parameter the inner border. Combinations of three or more borders are created by first creating a compound border using two borders and then making another compound border using that compound border and yet another border.

A *matte border* specifies the sizes, in pixels, of the top, left, bottom, and right edges of the border. Those edges can be composed of a single color, as they are in this example, or an image icon can be used.

Borders should be used carefully. They can be helpful in drawing attention to appropriate parts of your GUI and can conceptually group related items together. If used inappropriately, however, they can also detract from the elegance of the presentation. Borders should enhance the interface, not complicate or compete with it.

## Tool Tips and Mnemonics

Any Swing component can be assigned a *tool tip*, which is a short line of text that will appear when the cursor is rested momentarily on top of the component. Tool tips are generally used to give the user some information about the component, such as the purpose of a button.

A tool tip can be assigned using the `setToolTipText` method of a component. Here in an example:

```
JButton button = new JButton("Compute");
button.setToolTipText("Calculates the area under the curve.");
```

When the button is added to a container and displayed, it appears normally. When the user rolls the mouse pointer over the button, hovering there momentarily, the tool tip text pops up. When the user moves the mouse pointer off the button, the tool tip text disappears.

A *mnemonic* is a character that allows the user to push a button or make a menu choice using the keyboard in addition to the mouse. For example, when a mnemonic has been defined for a button, the user can hold down the Alt key and press the mnemonic character to activate the button. Using a mnemonic to activate

the button causes the system to behave just as it would if the user had used the mouse to press the button.

A mnemonic character should be chosen from the label on a button or menu item. Once the mnemonic has been established using the `setMnemonic` method, the character in the label will be underlined to indicate that it can be used as a shortcut. If a letter is chosen that is not in the label, nothing will be underlined, and the user won't know how to use the shortcut. You can set a mnemonic as follows:

```
JButton button = new JButton("Calculate");
button.setMnemonic('C');
```

When the button is displayed, the letter *C* in Calculate is underlined on the button label. When the user presses Alt-C, the button is activated just as if the user had pressed it with the mouse.

Some components can be *disabled* if they should not be used. A disabled component will appear “grayed out,” and nothing will happen if the user attempts to interact with it. To disable and enable components, we invoke the `setEnabled` method of the component, passing it a boolean value to indicate whether the component should be disabled (`false`) or enabled (`true`). For example:

```
JButton button = new JButton("Do It");
button.setEnabled(false);
```

Disabling components is a good idea when users should not be allowed to use the functionality of a component. The grayed appearance of the disabled component is an indication that using the component is inappropriate (and, in fact, impossible) at the current time. Disabled components not only convey to the user which actions are appropriate and which aren't but also prevent erroneous situations from occurring.

Let's look at an example that uses tool tips, mnemonics, and disabled components. The program in Listing D.32 presents the image of a light bulb and provides a button to turn the light bulb on and a button to turn the light bulb off.

There are actually two images of the light bulb: one showing it turned on and one showing it turned off. These images are brought in as `ImageIcon` objects. The `setIcon` method of the label that displays the image is used to set the appropriate image, depending on the current status. This processing is controlled in the `LightBulbPanel` class shown in Listing D.33.

The `LightBulbControls` class shown in Listing D.34 is a panel that contains the On and Off buttons. Both of these buttons have tool tips assigned to them, and both use mnemonics. Also, when one of the buttons is enabled, the other is disabled, and vice versa. When the light bulb is on, there is no reason for the On button to be enabled. Likewise, when the light bulb is off, there is no reason for the Off button to be enabled.

#### KEY CONCEPT

Components should be disabled when their use is inappropriate.

## LISTING D.32

```

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

/**
 * LightBulb.java
 *
 * Demonstrates mnemonics and tool tips.
 */
public class LightBulb
{
 /**
 * Sets up a frame that displays a light bulb image that can be
 * turned on and off.
 */
 public static void main(String[] args)
 {
 JFrame frame = new JFrame("Light Bulb");
 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

 LightBulbPanel bulb = new LightBulbPanel();
 LightBulbControls controls = new LightBulbControls(bulb);

 JPanel panel = new JPanel();
 panel.setBackground(Color.black);
 panel.setLayout(new BorderLayout(panel, BorderLayout.Y_AXIS));
 panel.add(Box.createRigidArea(new Dimension(0, 20)));
 panel.add(bulb);
 panel.add(Box.createRigidArea(new Dimension(0, 10)));
 panel.add(controls);
 panel.add(Box.createRigidArea(new Dimension(0, 10)));

 frame.getContentPane().add(panel);

 frame.pack();
 frame.setVisible(true);
 }
}

```

## LISTING D.32

*continued*

## DISPLAY



## LISTING D.33

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

/**
 * LightBulbPanel.java
 *
 * Represents the image for the LightBulb program.
 */
public class LightBulbPanel extends JPanel
```

## LISTING D.33

*continued*

```
{
 private boolean on;
 private ImageIcon lightOn, lightOff;
 private JLabel imageLabel;

 /**
 * Constructor: Sets up the images and the initial state.
 */
 public LightBulbPanel()
 {
 lightOn = new ImageIcon("lightBulbOn.gif");
 lightOff = new ImageIcon("lightBulbOff.gif");

 setBackground(Color.black);

 on = true;
 imageLabel = new JLabel(lightOff);
 add(imageLabel);
 }

 /**
 * Paints the panel using the appropriate image.
 */
 public void paintComponent(Graphics page)
 {
 super.paintComponent(page);

 if (on)
 imageLabel.setIcon(lightOn);
 else
 imageLabel.setIcon(lightOff);
 }

 /**
 * Sets the status of the light bulb.
 */
 public void setOn(boolean lightBulbOn)
 {
 on = lightBulbOn;
 }
}
```

Each button has its own listener class. The `actionPerformed` method of each sets the bulb's status, toggles the enabled state of both buttons, and causes the panel with the image to repaint itself.

### LISTING D.34

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

/**
 * LightBulbControls.java
 *
 * Represents the control panel for the LightBulb program.
 */
public class LightBulbControls extends JPanel
{
 private LightBulbPanel bulb;
 private JButton onButton, offButton;

 /**
 * Sets up the lightbulb control panel.
 */
 public LightBulbControls(LightBulbPanel bulbPanel)
 {
 bulb = bulbPanel;

 onButton = new JButton("On");
 onButton.setEnabled(false);
 onButton.setMnemonic('n');
 onButton.setToolTipText("Turn it on!");
 onButton.addActionListener(new OnListener());

 offButton = new JButton("Off");
 offButton.setEnabled(true);
 offButton.setMnemonic('f');
 offButton.setToolTipText("Turn it off!");
 offButton.addActionListener(new OffListener());

 setBackground(Color.black);
 add(onButton);
 add(offButton);
 }
}
```

## LISTING D.33

*continued*

```
/**
 * Represents the listener for the On button.
 */
private class OnListener implements ActionListener
{
 /**
 * Turns the bulb on and repaints the bulb panel.
 */
 public void actionPerformed(ActionEvent event)
 {
 bulb.setOn(true);
 onButton.setEnabled(false);
 offButton.setEnabled(true);
 bulb.repaint();
 }
}

/**
 * Represents the listener for the Off button.
 */
private class OffListener implements ActionListener
{
 /**
 * Turns the bulb off and repaints the bulb panel.
 */
 public void actionPerformed(ActionEvent event)
 {
 bulb.setOn(false);
 onButton.setEnabled(true);
 offButton.setEnabled(false);
 bulb.repaint();
 }
}
}
```

Note that the mnemonic characters used for each button are underlined in the display. When you run the program, you'll see that the tool tips automatically include an indication of the mnemonic that can be used for the button.

## D.7 GUI Design

---

As we focus on the details that allow us to create GUIs, we may sometimes lose sight of the big picture. We should always keep in mind that our goal is to solve a problem—to create software that is truly useful. Knowing the details of components, events, and other language elements gives us the tools we need to put GUIs together, but we must guide that knowledge with the following fundamental ideas of good GUI design:

- Know the user.
- Prevent user errors.
- Optimize user abilities.
- Be consistent.

The software designer must understand the user's needs and potential activities in order to develop an interface that will serve that user well. Keep in mind that to the user, the interface is the software. It is the only way the user interacts with the system. As such, the interface must satisfy the user's needs.

Whenever possible, we should design interfaces so that the user can make as few mistakes as possible. In many situations, we have the flexibility to choose one of several components to accomplish a specific task. We should always try to choose components that will prevent inappropriate actions and avoid invalid input. For example, if an input value must be one of a set of particular values, we should use components that allow the user to make only a valid choice. That is, constraining the user to a few valid choices with, for instance, a set of radio buttons is better than allowing the user to type arbitrary and possibly invalid data into a text field. We covered additional components appropriate for specific situations in this chapter.

Not all users are alike. Some are more adept than others at using a particular GUI or GUI components in general. We shouldn't design with only the lowest common denominator in mind. For example, we should provide shortcuts whenever reasonable. That is, in addition to a normal series of actions that will allow a user to accomplish a task, we should provide redundant ways to accomplish the same task. Using keyboard shortcuts (mnemonics) is a good example. Sometimes these additional mechanisms are less intuitive, but they may be faster for the experienced user.

Finally, consistency is important when dealing with large systems or multiple systems in a common environment. Users become familiar with a particular organization or color scheme; these should not be changed arbitrarily.

### KEY CONCEPT

The design of any GUI should adhere to basic guidelines on consistency and usability.



## Summary of Key Concepts

- A GUI is made up of components, events that represent user actions, and listeners that respond to those events.
- A frame is displayed as a separate window, but a panel can be displayed only as part of another container.
- Listeners are often defined as inner classes because of the intimate relationship between the listener and the GUI components.
- Radio buttons operate as a group, providing a set of mutually exclusive options.
- A slider lets the user specify a numeric value within a bounded range.
- A combo box provides a drop-down menu of options.
- A timer generates action events at regular intervals and can be used to control an animation.
- Every container is managed by a layout manager, which determines how components are visually presented.
- When changes occur, the components in a container reorganize themselves according to the layout manager's policy.
- The layout manager for each container can be explicitly set.
- A GUI's appearance is a function of the containment hierarchy and the layout managers of each container.
- Moving the mouse and clicking the mouse button generate events to which a program can respond.
- A listener may have to provide empty method definitions for unheeded events to satisfy the interface.
- Rubberbanding is the graphical effect caused when a shape seems to expand as the mouse is dragged.
- Key events allow a program to respond immediately to the user pressing keyboard keys.
- A listener class can be created by deriving it from an event adapter class.
- A file chooser allows the user to browse a disk and select a file to be processed.
- A color chooser allows the user to select a color from a palette or by using RGB values.
- Borders can be applied to components to group objects and focus attention.
- Components should be disabled when their use is inappropriate.
- The design of any GUI should adhere to basic guidelines on consistency and usability.

## Self-Review Questions

- SR D.1 What three elements are needed in any Java GUI?
- SR D.2 What is the difference between a frame and a panel?
- SR D.3 What is the relationship between an event and a listener?
- SR D.4 Can we add any kind of listener to any component? Explain.
- SR D.5 What type of event does a push button generate? A text field? A check box?
- SR D.6 Compare and contrast check boxes and radio buttons.
- SR D.7 When would you use a slider?
- SR D.8 What does a `Timer` object do?
- SR D.9 When is a layout manager consulted?
- SR D.10 How does the flow layout manager behave?
- SR D.11 Describe the areas of a border layout.
- SR D.12 What effect does a glue component in a box layout have?
- SR D.13 What is the containment hierarchy for a GUI?
- SR D.14 What is a mouse event?
- SR D.15 What is a key event?
- SR D.16 What is an event adapter class?
- SR D.17 What is a dialog box?
- SR D.18 What is a file chooser? A color chooser?
- SR D.19 What is the role of the `BorderFactory` class?
- SR D.20 What is a tool tip?
- SR D.21 When should a component be disabled?

## Exercises

- EX D.1 Explain how two components can be set up to share the same listener. How can the listener tell which component generated the event?
- EX D.2 Explain how one component can use two separate listeners at the same time. Give an example.
- EX D.3 Explain what would happen if the radio buttons used in the `QuoteOptions` program were not organized into a `ButtonGroup` object. Modify the program to test your answer.

- EX D.4 Why, in the `SlideColor` program, is the value of a slider able to reach 255 but the largest labeled tick mark is 250?
- EX D.5 What are the two main factors that affect how smooth the animation is in the `Rebound` program? Explain how changing either would affect it.
- EX D.6 What visual effect would result from changing the horizontal and vertical gaps on the border layout used in the `LayoutDemo` program? Make the change to test your answer.
- EX D.7 What would happen if, in the `Coordinates` program, we did not provide empty definitions for one or more of the unused mouse events?
- EX D.8 The `Coordinates` program listens for a mouse pressed event to draw a dot. How would the program behave differently if it listened for a mouse released event instead? A mouse clicked event?
- EX D.9 What would happen if the call to `super.paintComponent` were removed from the `paintComponent` method of the `CoordinatesPanel` class? Remove it and run the program to test your answer.
- EX D.10 What would happen if the call to `super.paintComponent` were removed from the `paintComponent` method of the `RubberLinesPanel` class? Remove it and run the program to test your answer. In what ways is the answer different from the answer to Exercise D.9.
- EX D.11 Write the lines of code that will define a compound border using three borders. Use a line border on the inner edge, an etched border on the outer edge, and a raised bevel border in between.
- EX D.12 Draw a UML class diagram that shows the relationships among the classes used in the `PushCounter` program.
- EX D.13 Draw a UML class diagram that shows the relationships among the classes used in the `Fahrenheit` program.
- EX D.14 Draw a UML class diagram that shows the relationships among the classes used in the `LayoutDemo` program.
- EX D.15 Create a UML class diagram for the `Direction` program.

## Programming Projects

- PP D.1 Design and implement an application that displays a button and a label. Every time the button is pushed, the label should display a random number between 1 and 100, inclusive.

- PP D.2 Design and implement an application that presents two buttons and a label to the user. Label the buttons Increment and Decrement, respectively. Display a numeric value (initially 50) using the label. Each time the Increment button is pushed, increment the value displayed. Likewise, each time the Decrement button is pressed, decrement the value displayed. Create two separate listener classes for the two buttons.
- PP D.3 Modify your solution to Programming Project D.2 so that it uses only one listener for both buttons.
- PP D.4 Modify the `Fahrenheit` program so that it displays a button that, when pressed, causes the conversion calculation to take place. That is, your modification will give the user the option of pressing Enter in the text field or pressing the button. Have the listener that is already defined for the text field also listen for the button push.
- PP D.5 Modify the `Direction` program so that the image is not allowed to move out of the visible area of the panel. Ignore any key event that would cause that to happen.
- PP D.6 Modify the `Direction` program so that, in addition to responding to the arrow keys, it also responds to four other keys that cause the image to move in diagonal directions. When the T key is pressed, move the image up and to the left. Likewise, use U to move up and right, G to move down and left, and J to move down and right. Do not move the image if it has reached a window boundary.
- PP D.7 Design and implement an application that draws a traffic light and uses a push button to change the state of the light. Derive the drawing surface from the `JPanel` class, and use another panel to organize the drawing surface and the button.
- PP D.8 Develop an application that implements a prototype user interface for composing an email message. The application should have text fields for the To, CC, and Bcc address lists and subject line, and one for the message body. Include a button labeled Send. When the Send button is pushed, the program should print the contents of all fields to standard output using `println` statements.
- PP D.9 Design and implement an application that uses dialog boxes to obtain two integer values (one dialog box for each value) and display the sum and product of the values. Use another dialog box to see whether the user wants to process another pair of values.

- PP D.10 Design and implement a program whose background changes color depending on where the mouse pointer is located. If the mouse pointer is on the left half of the program window, display red; if it is on the right half, display green.
- PP D.11 Design and implement an application that serves as a mouse odometer, continually displaying how far, in pixels, the mouse has moved (while over the program window). Display the current odometer value using a label. *Hint:* Compare the current position of the mouse to the last position, and use the distance formula to determine how far the mouse has traveled.
- PP D.12 Design and implement an application that draws a circle using a rubberbanding technique. The circle size is determined by a mouse drag. Use the original mouse click location as a fixed center point. *Hint:* Compute the distance between the mouse pointer and the center point to determine the current radius of the circle.
- PP D.13 Modify the `StyleOptions` program to allow the user to specify the size of the font. Use a text field to obtain the size.
- PP D.14 Modify your solution to Programming Project D.13 such that it uses a slider to obtain the font size.
- PP D.15 Develop a simple tool for calculating basic statistics for a segment of text. The application should have a single window with a scrolling text box (a `JTextArea`) and a stats box. The stats box should be a panel with a titled border, containing labeled fields that display the number of words in the text box and the average word length, as well as any other statistics that you would like to add. The stats box should also contain a button that, when pressed, recomputes the statistics for the current contents of the text field.
- PP D.16 Modify the `Rebound` program from this chapter such that when the mouse button is clicked on the program window, the animation stops, and when it is clicked again, the animation resumes.
- PP D.17 Design and implement a program that uses a `JColorChooser` object to obtain a color from the user and display that color as the background of the primary program window. Use a dialog box to determine whether the user wants to display another color and, if so, to redisplay the color chooser.
- PP D.18 Modify the `JukeBox` program such that the Play and Stop button functionality can also be controlled using keyboard mnemonics.
- PP D.19 Modify the `Coordinates` program such that it creates its listener by extending an adapter class instead of implementing an interface.

- PP D.20 Design and implement an application that displays an animation of a car (side view) moving across the screen from left to right. Create a `Car` class that represents the car.
- PP D.21 Design and implement an application that plays a game called Catch-the-Creature. Use an image to represent the creature. Have the creature appear at a random location for a random duration and then disappear and reappear somewhere else. The goal is to “catch” the creature by pressing the mouse button while the mouse pointer is on the creature image. Create a separate class to represent the creature, and include in it a method that determines whether the location of the mouse click corresponds to the current location of the creature. Display a count of the number of times the creature is caught.
- PP D.22 Design and implement an application that works as a stopwatch. Include a display that shows the time (in seconds) as it increments. Include buttons that allow the user to start and stop the time and to reset the display to zero. Arrange the components to present a nice interface.

## Answers to Self-Review Questions

- SR D.1 A GUI in a Java program is made up of on-screen components, events that those components generate, and listeners that respond to events when they occur.
- SR D.2 Both a frame and a panel are containers that can hold GUI elements. However, a frame is displayed as a separate window with a title bar, whereas a panel cannot be displayed on its own. A panel is often displayed inside a frame.
- SR D.3 Events usually represent user actions. A listener object is set up to listen for a certain event to be generated from a particular component. The relationship between a particular component that generates an event and the listener that responds to that event is set up explicitly.
- SR D.4 No, we cannot add any listener to any component. Each component generates a certain set of events, and only listeners of those types can be added to the component.
- SR D.5 Both push buttons and text fields generate action events. A check box generates an item state changed event.

- SR D.6 Both check boxes and radio buttons show a toggled state: either on or off. However, radio buttons work as a group in which only one can be toggled on at any point in time. Check boxes, on the other hand, represent independent options. They can be used alone or in a set in which any combination of toggled states is valid.
- SR D.7 A slider is useful when the user needs to specify a numeric value within specific bounds. Using a slider to get this input, rather than a text field or some other component, minimizes user error.
- SR D.8 An object created from the `TIMER` class produces an action event at regular intervals. It can be used to control the speed of an animation.
- SR D.9 A layout manager is consulted whenever the visual appearance of its components might be affected, such as when the container is resized or when a new component is added to the container.
- SR D.10 Flow layout attempts to put as many components on a row as possible. Multiple rows are created as needed.
- SR D.11 Border layout is divided into five areas: North, South, East, West, and Center. The North and South areas are at the top and bottom of the container, respectively, and span the entire width of the container. Sandwiched between them, from left to right, are the West, Center, and East areas. Any unused area takes up no space, and the others fill in as needed.
- SR D.12 A glue component in a box layout dictates where any extra space in the layout should go. It expands as necessary but takes up no space if there is no extra space to distribute.
- SR D.13 The containment hierarchy of a GUI is created by nested containers. The way the containers are nested, and the layout managers that those containers employ, dictate the details of the visual presentation of the GUI.
- SR D.14 A mouse event is an event generated when the user manipulates the mouse in various ways. There are several types of mouse events that may be of interest in a particular situation, including the mouse being moved, a mouse button being pressed, the mouse entering a particular component, and the mouse being dragged.
- SR D.15 A key event is generated when a keyboard key is pressed, which allows a listening program to respond immediately to the user input. The object representing the event holds a code that specifies which key was pressed.

- SR D.16 An event adapter class is a class that implements a listener interface, providing empty definitions for all of its methods. A listener class can be created by extending the appropriate adapter class and defining only the methods of interest.
- SR D.17 A dialog box is a small window that appears for the purpose of conveying information, confirming an action, or accepting input. Generally, dialog boxes are used in specific situations for brief user interactions.
- SR D.18 A file chooser and a color chooser are specialized dialog boxes that allow the user to select a file from disk and a color, respectively.
- SR D.19 The `BorderFactory` class contains several methods used to create borders that can be applied to components.
- SR D.20 A tool tip is a small amount of text that appears when the mouse cursor is allowed to rest over a specific component. Tool tips are used to explain, briefly, the purpose of a component.
- SR D.21 GUI components should be disabled when their use is inappropriate. This helps guide the user to proper actions and minimizes error handling and special cases.



*This page is intentionally left blank.*



# Hashing

# E

In Chapter 11, we discussed the idea that a binary search tree is, in effect, an efficient implementation of a set or a map. In this appendix, we examine hashing, an approach to implementing a set or map collection that can be even more efficient than binary search trees.

## Appendix

## E.1 Hashing

In all of our discussions of the implementations of collections, we have proceeded with one of two assumptions about the order of elements in a collection:

- Order is determined by the order in which elements are added to and/or removed from our collection, as in the case of stacks, queues, unordered lists, and indexed lists.
- Order is determined by comparing the values of the elements (or some key component of the elements) to be stored in the collection, as in the case of ordered lists and binary search trees.

In this appendix, we will explore the concept of *hashing*, which means that the order—and, more specifically, the location of an item within the collection—is determined by some function of the value of the element to be stored, or some function of a key value of the element to be stored. In hashing, elements are stored

in a *hash table*, and the location of each element in the table is determined by a *hashing function*. Each location in the table may be referred to as a *cell* or a *bucket*. We will discuss hashing functions further in Section E.2. We will discuss implementation strategies and algorithms, and we will leave the implementations as programming projects.

### KEY CONCEPT

In hashing, elements are stored in a hash table, and the location of each element in the table is determined by a hashing function.

Consider a simple example where we create an array that will hold 26 elements. Wishing to store names in our array, we create a hashing function that equates each name to the position in the array associated with the first letter of the name. (For example, a first letter of A would be mapped to position 0 of the array, a first letter of D would be mapped to position 3 of the array, and so on.) Figure E.1 illustrates this scenario after several names have been added.

Notice that, unlike our earlier implementations of collections, using a hashing approach results in the access time to a particular element being independent of the number of elements in the table. This means that all of the operations on an element of a hash table should be  $O(1)$ . This is the result of no longer having to do comparisons to find a particular element or to locate the appropriate position for a given element. Using hashing, we simply calculate where a particular element should be.

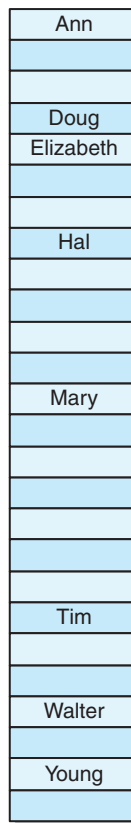
### KEY CONCEPT

The situation where two elements or keys map to the same location in the table is called a collision.

### KEY CONCEPT

A hashing function that maps each element to a unique position in the table is called a perfect hashing function.

However, this efficiency is fully realized only if each element maps to a unique position in the table. Consider our example from Figure E.1. What will happen if we attempt to store the name “Ann” and the name “Andrew”? This situation, where two elements or keys map to the same location in the table, is called a *collision*. We will discuss how to resolve collisions in Section E.3.



|           |
|-----------|
| Ann       |
|           |
|           |
| Doug      |
| Elizabeth |
|           |
|           |
| Hal       |
|           |
|           |
|           |
|           |
| Mary      |
|           |
|           |
|           |
|           |
|           |
| Tim       |
|           |
|           |
| Walter    |
|           |
| Young     |
|           |

**FIGURE E.1** A simple hashing example

A hashing function that maps each element to a unique position in the table is said to be a *perfect hashing function*. Although it is possible in some situations to develop a perfect hashing function, a hashing function that does a good job of distributing the elements among the table positions will still result in constant time ( $O(1)$ ) access to elements in the table and an improvement over our earlier algorithms that were either  $O(n)$ , in the case of our linear approaches, or  $O(\log n)$ , in the case of search trees.

Another issue surrounding hashing is the question of how large the table should be. If the data set is of known size and a perfect hashing function can be used, then we simply make the table the same size as the data set. If a perfect hashing function is not available or practical but the size of the data set is known, a good rule of thumb is to make the table 150 percent of the size of the data set.

The third case is very common and far more interesting. What if we do not know the size of the data set? In this case, we depend on *dynamic resizing*. Dynamic resizing of a hash table involves creating a new hash table that is larger than—perhaps even twice as large as—the original, inserting all of the elements of the original table into the new table, and then discarding the original table. When to resize is also an interesting question. One possibility is to use the same method we used with our earlier array implementations and simply expand the table when it is full. However, it is the nature of hash tables that their performance seriously degrades as they become full. A better approach is to use a *load factor*. The load factor of a hash table is the percentage occupancy of the table at which the table will be resized. For example, if the load factor were set to 0.50, then the table would be resized each time it reached 50 percent capacity.

## E.2 Hashing Functions

Although perfect hashing functions are possible if the data set is known, we do not need the hashing function to be perfect to get good performance from the hash table. Our goal is simply to develop a function that does a reasonably good job of distributing our elements in the table such that we avoid collisions. A reasonably good hashing function will still result in constant time access ( $O(1)$ ) to our data set.

### KEY CONCEPT

Extraction involves using only a part of the element's value or key to compute the location at which to store the element.

There are a variety of approaches to developing a hashing function for a particular data set. The method that we used in our example in the previous section is called *extraction*. Extraction involves using only a part of the element's value or key to compute the location at which to store the element. In our previous example, we simply extracted the first letter of a string and computed its value relative to the letter A.

Other examples of extraction include storing phone numbers according to the last four digits and storing information about cars according to the first three characters of the license plate.

### The Division Method

Creating a hashing function by *division* simply means using the remainder of the key divided by some positive integer  $p$  as the index for the given element. This function could be defined as follows:

$$\text{Hashcode}(\text{key}) = \text{Math.abs}(\text{key}) \% p$$

This function will yield a result in the range of 0 to  $p-1$ . If we use our table size as  $p$ , we then have an index that maps directly to a location in the table.

Using a prime number  $p$  as the table size and the divisor helps provide a better distribution of keys to locations in the table.

For example, if our key value is 79 and our table size is 43, then the division method will result in an index value of 36. The division method is very effective when one is dealing with an unknown set of key values.

## The Folding Method

In the *folding method*, the key is divided into parts that are then combined or folded together to create an index into the table. This is done by first dividing the key into parts, where each of the parts of the key will be the same length as the desired index, except possibly the last one. In the *shift folding method*, these parts are then added together to create the index. For example, if our key were the Social Security number 987-65-4321, we might divide this into three parts: 987, 654, and 321. Adding these together would yield 1962. Assuming that we are looking for a three-digit key, at this point we could use either division or extraction to get our index.

A second possibility is *boundary folding*. There are a number of variations on this approach. However, generally, they involve reversing some of the parts of the key before adding. One variation on this approach is to imagine that the parts of the key are written side by side on a piece of paper and that the piece of paper is folded along the boundaries of the parts of the key. In this way, if we begin with the same key (987-65-4321), we first divide it into parts: 987, 654, and 321. We then reverse every other part of the key, which yields 987, 456, and 321. Adding these together yields 1764, and once again we can proceed with either extraction or division to get our index. Other variations on folding use different algorithms to determine which parts of the key to reverse.

Folding may also be a useful method for building a hashing function for a key that is a string. One approach to this is to divide the string into substrings the same length (in bytes) as the desired index and then combine these strings using an *exclusive-or* function. This is also a useful way to convert a string into a number so that other methods, such as division, may be applied to strings.

## The Mid-Square Method

In the *mid-square method*, the key is multiplied by itself, and then the extraction method is used to extract the appropriate number of digits from the middle of the squared result to serve as an index. The same “middle” digits must be chosen each time, to provide consistency. For example, if our key were 4321, we would multiply

### KEY CONCEPT

In the shift folding method, the parts of the key are added together to create the index.

the key by itself, which would yield 18671041. Assuming that we need a three-digit key, we might extract 671 or 710, depending on how we construct our algorithm. It is also possible to extract bits instead of digits and then construct the index from the extracted bits.

The mid-square method may also be effectively used with strings by manipulating the binary representations of the characters in the string.

## The Radix Transformation Method

In the *radix transformation method*, the key is transformed into another numeric base. For example, if our key were 23 in base 10, we might convert it into 32 in base 7. We would then use the division method and divide the converted key by the table size and use the remainder as our index. Continuing our previous example, if our table size were 17, we would compute the function

$$\begin{aligned}\text{Hashcode}(23) &= \text{Math.abs}(32) \% 17 \\ &= 15\end{aligned}$$

## The Digit Analysis Method

In the *digit analysis method*, the index is formed by extracting, and then manipulating, specific digits from the key. For example, if our key were 1234567, we might select the digits in positions 2 through 4, obtaining 234, and then manipulate them to form our index. This manipulation can take many forms, including simply reversing the digits (which yields 432), performing a circular shift to the right (which yields 423), performing a circular shift to the left (which yields 342), swapping each pair of digits (which yields 324), or any number of other possibilities, including the methods we have already discussed. The goal is simply to provide a function that does a reasonable job of distributing keys to locations in the table.

## The Length-Dependent Method

In the *length-dependent method*, the key and the length of the key are combined in some way to form either the index itself or an intermediate value that is then used with one of our other methods to form the index. For example, if our key were 8765, we might multiply the first two digits by the length and then divide by the last digit, which would yield 69. If our table size were 43, we would then use the division method, which would result in an index of 26.

### KEY CONCEPT

The length-dependent method and the mid-square method may also be effectively used with strings by manipulating the binary representations of the characters in the string.

The length-dependent method may also be effectively used with strings by manipulating the binary representations of the characters in the string.

## Hashing Functions in the Java Language

The `java.lang.Object` class defines a method called `hashCode` that returns an integer based on the memory location of the object. This is generally not very useful. Classes that are derived from `Object` often override the inherited definition of `hashCode` to provide their own version. For example, the `String` and `Integer` classes define their own `hashCode` methods. These more specific `hashCode` functions can be very effective for hashing. Having the `hashCode` method defined in the `Object` class means that all Java objects can be hashed. However, it is also possible—and often preferable—to define your own `hashCode` method for any class that you intend to store in a hash table.

### KEY CONCEPT

Although Java provides a `hashCode` method for all objects, it is often preferable to define a specific hashing function for any particular class.

## E.3 Resolving Collisions

If we are able to develop a perfect hashing function for a particular data set, then we do not need to concern ourselves with collisions, the situation where more than one element or key map to the same location in the table. However, when a perfect hashing function is not possible or practical, there are a number of ways to handle collisions. Similarly, if we are able to develop a perfect hashing function for a particular data set, then we do not need to concern ourselves with the size of the table. In this case, we will simply make the table the exact size of the data set. Otherwise, if the size of the data set is known, it is generally a good idea to set the initial size of the table to about 150 percent of the expected element count. If the size of the data set is not known, then dynamic resizing of the table becomes an issue.

### Chaining

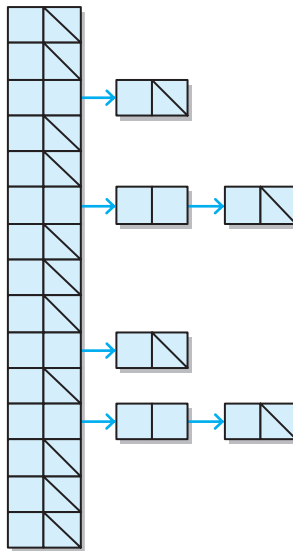
The *chaining method* for handling collisions simply treats the hash table conceptually as a table of collections, rather than as a table of individual cells. Thus each cell is a pointer to the collection associated with that location in the table. Usually this internal collection is either an unordered list or an ordered list. Figure E.2 illustrates this conceptual approach.

Chaining can be implemented in a variety of ways. One approach is to make the array holding the table larger than the number of cells in the table and use the extra space as an overflow area to store the linked lists associated with each table location. In this method, each position in the array can store both an element (or a key) and the array index of the next element in its list. The first element mapped to a particular location in the table would actually be stored in

### KEY CONCEPT

The chaining method for handling collisions simply treats the hash table conceptually as a table of collections, rather than as a table of individual cells.



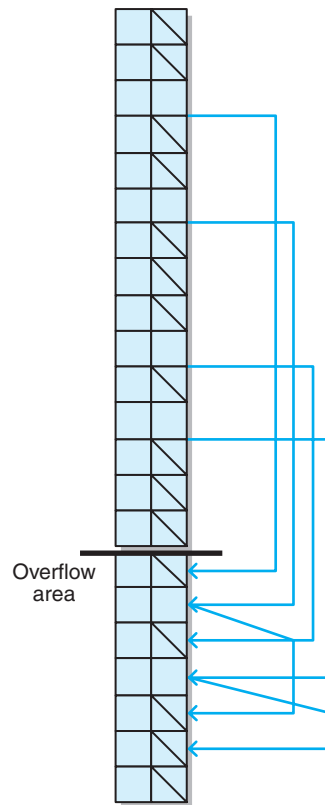


**FIGURE E.2** The chaining method of handling a collision

that location. The next element mapped to that location would be stored in a free location in this overflow area, and the array index of this second element would be stored with the first element in the table. If a third element were mapped to the same location, the third element would also be stored in this overflow area, and the index of the third element would be stored with the second element. Figure E.3 illustrates this strategy.

Note that when this method is used, the table itself can never be full. However, if the table is implemented as an array, the array can become full, requiring a decision on whether to throw an exception or simply expand capacity. In our earlier collections, we chose to expand the capacity of the array. In this case, expanding the capacity of the array but leaving the embedded table the original size would have disastrous effects on efficiency. A more satisfactory solution is to expand the array and expand the embedded table within the array. This will, however, require that all of the elements in the table be rehashed using the new table size. We will discuss the dynamic resizing of hash tables further in Section E.5.

With this method, the worst case is that our hashing function will not do a good job of distributing elements to locations in the table, and consequently we end up with one linked list of  $n$  elements, or a small number of linked lists with roughly  $n/k$  elements each, where  $k$  is some relatively small constant. In this case, hash tables become  $O(n)$  for both insertions and searches. Thus you can see how important it is to develop a good hashing function.



**FIGURE E.3** Chaining using an overflow area

A second method for implementing chaining is using links. In this method, each cell or bucket in the hash table would be something like the `LinearNode` class used earlier in this text to construct linked lists. In this way, as a second element is mapped to a particular bucket, we simply create a new `LinearNode`, set the `next` reference of the existing node to point to the new node, set the `element` reference of the new node to the element being inserted, and set the `next` reference of the new node to null. The result is an implementation model that looks exactly like the conceptual model shown in Figure E.2.

A third method for implementing chaining is to literally make each position in the table a pointer to a collection. In this way, we could represent each position in the table with a list or perhaps even a more efficient collection (such as a balanced binary search tree), and this would improve our worst case. Keep in mind, however, that if our hashing function is doing a good job of distributing elements to

locations in the table, this approach may incur a great deal of overhead while achieving very little improvement.

## Open Addressing

The *open addressing method* for handling collisions looks for an open position in the table other than the one to which the element is originally hashed. There are a variety of methods for finding another available location in the table. We will examine three of these methods: linear probing, quadratic probing, and double hashing.

The simplest of these methods is *linear probing*. In linear probing, if an element hashes to position  $p$ , and position  $p$  is already occupied, we simply try position  $(p+1)\%s$ , where  $s$  is the size of the table. If position  $(p+1)\%s$  is already occupied, we try position  $(p+2)\%s$ , and so on until either we find an open position or we find ourselves back at the original position. If we find an open position, we insert the new element. What to do if we do not find an open position is a design decision made when creating a hash table. As we have discussed before, one possibility is to throw an exception if the table is full. Another possibility is to expand the capacity of the table and rehash the existing entries.

### KEY CONCEPT

The open addressing method for handling collisions looks for an open position in the table other than the one to which the element is originally hashed.

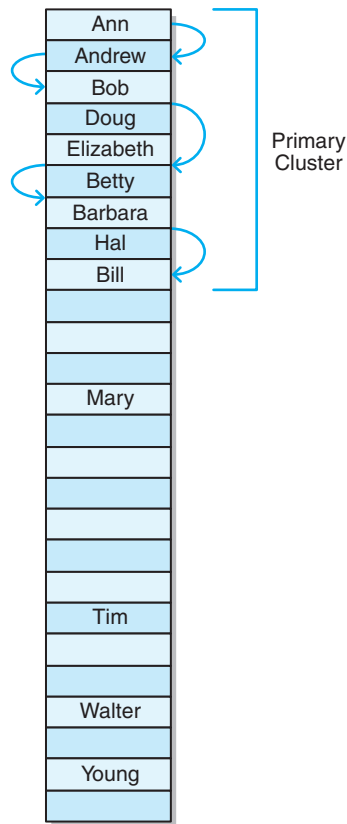
The problem with linear probing is that it tends to create clusters of filled positions within the table, and these clusters then affect the performance of insertions and searches. Figure E.4 illustrates the linear probing method and the creation of a cluster using our earlier hashing function of extracting the first character of the string.

In this example, Ann was entered, followed by Andrew. Because Ann already occupied position 0 of the array, Andrew was placed in position 1. Later, Bob was entered. Because Andrew already occupied position 1, Bob was placed in the next open position, which was position 2. Doug and Elizabeth were already in the table by the time Betty arrived, so Betty could not be placed in position 1, 2, 3, or 4 and was placed in the next open position, position 5. After Barbara, Hal, and Bill were added, we find that there is now a nine-location cluster at the front of the table, which will continue to grow as more names are added. Thus we see that linear probing may not be the best approach.

A second form of the open addressing method is *quadratic probing*. If we use quadratic probing, instead of a linear approach, then once we have a collision, we follow a formula such as

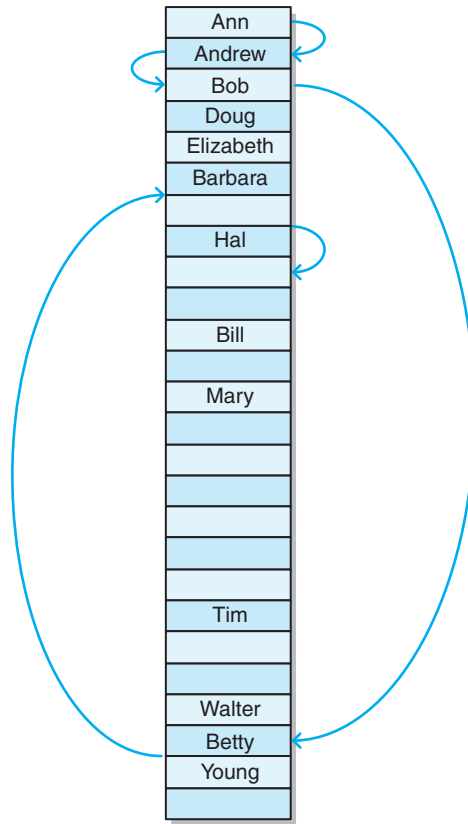
$$\text{newhashcode}(x) = \text{hashcode}(x) + (-1)^{i-1} ((i+1)/2)^2$$

for  $i$  in the range of 1 to  $s-1$ , where  $s$  is the table size.



**FIGURE E.4** Open addressing using linear probing

The result of this formula is the search sequence  $p, p+1, p-1, p+4, p-4, p+9, p-9, \dots$ . Of course, this new hash code is then put through the division method to keep it within the table range. As with linear probing, the same possibility exists that we will eventually get back to the original hash code without having found an open position in which to insert. This “full” condition can be handled in all of the same ways that we described for chaining and linear probing. The benefit of the quadratic probing method is that it does not have as strong a tendency toward clustering as linear probing. Figure E.5 illustrates quadratic probing for the same key set and hashing function that we used in Figure E.4. Notice that after the same data have been entered, we still have a cluster at the front of the table. However, this cluster occupies only six buckets instead of the nine-bucket cluster created by linear probing.



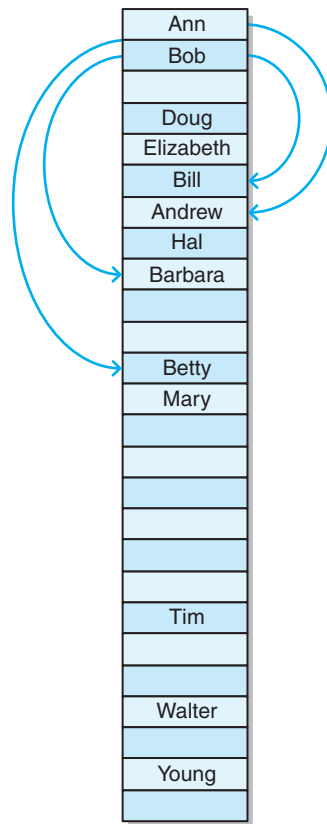
**FIGURE E.5** Open addressing using quadratic probing

A third form of the open addressing method is *double hashing*. Using the double hashing method, we resolve collisions by providing a secondary hashing function to be used when the primary hashing function results in a collision. For example, if a key  $x$  hashes to a position  $p$  that is already occupied, then the next position  $p'$  that we try is

$$p' = p + \text{secondaryhashcode}(x)$$

If this new position is also occupied, then we look to position

$$p'' = p + 2 * \text{secondaryhashcode}(x)$$



**FIGURE E.6** Open addressing using double hashing

We continue searching in this way, of course using the division method to maintain our index within the bounds of the table, until an open position is found. This method, although it is somewhat more costly because of the introduction of an additional function, tends to further reduce clustering beyond the improvement gained by quadratic probing. Figure E.6 illustrates this approach, again using the same key set and hashing function as our previous examples. For this example, the secondary hashing function is the length of the string. Notice that with the same data, we no longer have a cluster at the front of the table. However, we have developed a six-bucket cluster from Doug through Barbara. The advantage of double hashing is that even after a cluster has been created, it will tend to grow more slowly than it would if we were using linear probing or even quadratic probing.

## E.4 Deleting Elements from a Hash Table

---

Thus far, our discussion has centered on the efficiency of insertion of and searching for elements in a hash table. What happens if we remove an element from a hash table? The answer to this question depends on which implementation we have chosen.

### Deleting from a Chained Implementation

If we have chosen to implement our hash table using a chained implementation and an array with an overflow area, then removing an element falls into one of five cases:

**Case 1** The element we are attempting to remove is the only one mapped to the particular location in the table. In this case, we simply remove the element by setting the table position to null.

**Case 2** The element we are attempting to remove is stored in the table (not in the overflow area) but has an index into the overflow area for the next element at the same position. In this case, we replace the element and the next index value in the table with the element and the next index value of the array position pointed to by the element to be removed. We then also must set the position in the overflow area to null and add it back to whatever mechanism we are using to maintain a list of free positions.

**Case 3** The element we are attempting to remove is at the end of the list of elements stored at that location in the table. In this case, we set its position in the overflow area to null, and we set the next index value of the previous element in the list to null as well. We then also must set the position in the overflow area to null and add it back to whatever mechanism we are using to maintain a list of free positions.

**Case 4** The element we are attempting to remove is in the middle of the list of elements stored at that location in the table. In this case, we set its position in the overflow area to null, and we set the next index value of the previous element in the list to the next index value of the element being removed. We then also must add it back to whatever mechanism we are using to maintain a list of free positions.

**Case 5** The element we are attempting to remove is not in the list. In this case, we throw an `ElementNotFoundException`.

If we have chosen to implement our hash table using a chained implementation where each element in the table is a collection, then we simply remove the target element from the collection.

## Deleting from an Open Addressing Implementation

If we have chosen to implement our hash table using an open addressing implementation, then deletion creates more of a challenge. Consider the example in Figure E.7. Note that the elements “Ann,” “Andrew,” and “Amy” all mapped to the same location in the table, and the collision was resolved using linear probing. What happens if we now remove “Andrew”? If we then search for “Amy” we will not find that element because the search will find “Ann” and then follow the linear probing rule to look in the next position, find it null, and return an exception.

The solution to this problem is to mark items as deleted but not actually remove them from the table until some future point when the deleted element is overwritten by a new inserted element or the entire table is rehashed, either because it is being expanded or because we have reached some predetermined threshold for the percentage of deleted records in the table. This means that we

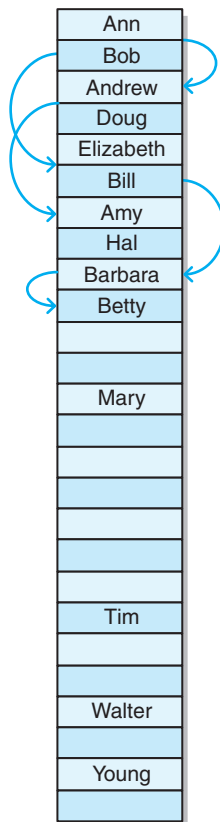


FIGURE E.7 Open addressing and deletion



will need to add a `boolean` flag to each node in the table and modify all of our algorithms to test and/or manipulate that flag.

## E.5 Hash Tables in the Java Collections API

The Java Collections API provides seven implementations of hashing: `Hashtable`, `HashMap`, `HashSet`, `IdentityHashMap`, `LinkedHashSet`, `LinkedHashMap`, and `WeakHashMap`. To understand these different solutions, we must first remind ourselves of the distinction between a *set* and a *map* in the Java Collections API, as well as some of our other pertinent definitions.

### KEY CONCEPT

The load factor is the maximum percentage occupancy allowed in the hash table before it is resized.

A *set* is a collection of objects where, in order to find an object, we must have an exact copy of the object we are looking for. A *map*, on the other hand, is a collection that stores key-value pairs so that, given the key, we can find the associated value.

Another definition that will be useful to us as we explore the Java Collections API implementations of hashing is that of a *load factor*. The load factor, as stated earlier, is the maximum percentage occupancy allowed in the hash table before it is resized. For the implementations that we are going to discuss here, the default is 0.75. Thus, when one of these implementations becomes 75 percent full, a new hash table is created that is twice the size of the current one, and then all of the elements from the current table are inserted into the new table. The load factor of these implementations can be altered when the table is created.

All of these implementations rely on the `hashCode` method of the object being stored to return an integer. This integer is then processed using the division method (using the table size) to produce an index within the bounds of the table. As stated earlier, the best practice is to define your own `hashCode` method for any class that you intend to store in a hash table.

Let's look at each of these implementations.

### The `Hashtable` Class

The `Hashtable` implementation of hashing is the oldest of the implementations in the Java Collections API. In fact, it predates the Collections API and was modified in version 1.2 to implement the `Map` interface so that it would become a part of the Collections API. Unlike the newer Java Collections implementations, `Hashtable` is synchronized. Figure E.8 shows the operations for the `Hashtable` class.

| Return Value                | Method                                                        | Description                                                                                                                                                                                                            |
|-----------------------------|---------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|                             | <code>Hashtable()</code>                                      | Constructs a new, empty hash table with a default initial capacity (11) and load factor, which is 0.75.                                                                                                                |
|                             | <code>Hashtable(int initialCapacity)</code>                   | Constructs a new, empty hash table with the specified initial capacity and default load factor, which is 0.75.                                                                                                         |
|                             | <code>Hashtable(int initialCapacity, float loadFactor)</code> | Constructs a new, empty hash table with the specified initial capacity and the specified load factor.                                                                                                                  |
|                             | <code>Hashtable (Map t)</code>                                | Constructs a new hash table with the same mappings as the given <code>Map</code> .                                                                                                                                     |
| <code>void</code>           | <code>clear()</code>                                          | Clears this hash table so that it contains no keys.                                                                                                                                                                    |
| <code>Object</code>         | <code>clone()</code>                                          | Creates a shallow copy of this hash table.                                                                                                                                                                             |
| <code>boolean</code>        | <code>contains(Object value)</code>                           | Tests if some key maps into the specified value in this hash table.                                                                                                                                                    |
| <code>boolean</code>        | <code>containsKey(Object key)</code>                          | Tests if the specified object is a key in this hash table.                                                                                                                                                             |
| <code>boolean</code>        | <code>containsValue(Object value)</code>                      | Returns true if this hash table maps one or more keys to this value.                                                                                                                                                   |
| <code>Enumeration</code>    | <code>elements()</code>                                       | Returns an enumeration of the values in this hash table.                                                                                                                                                               |
| <code>Set</code>            | <code>entrySet()</code>                                       | Returns a <code>Set</code> view of the entries contained in this hash table.                                                                                                                                           |
| <code>boolean</code>        | <code>equals(Object o)</code>                                 | Compares the specified <code>Object</code> with this <code>Map</code> for equality, as per the definition in the <code>Map</code> interface.                                                                           |
| <code>Object</code>         | <code>get(Object key)</code>                                  | Returns the value to which the specified key is mapped in this hash table.                                                                                                                                             |
| <code>int</code>            | <code>hashCode()</code>                                       | Returns the hash code value for this <code>Map</code> as per the definition in the <code>Map</code> interface.                                                                                                         |
| <code>boolean</code>        | <code>isEmpty()</code>                                        | Tests if this hash table maps no keys to values.                                                                                                                                                                       |
| <code>Enumeration</code>    | <code>keys()</code>                                           | Returns an enumeration of the keys in this hash table.                                                                                                                                                                 |
| <code>Set</code>            | <code>keySet()</code>                                         | Returns a <code>Set</code> view of the keys contained in this hash table.                                                                                                                                              |
| <code>Object</code>         | <code>put(Object key Object value)</code>                     | Maps the specified key to the specified value in this hash table.                                                                                                                                                      |
| <code>void</code>           | <code>putAll(Map t)</code>                                    | Copies all of the mappings from the specified <code>Map</code> to this hash table. These mappings will replace any mappings that this hash table had for any of the keys currently in the specified <code>Map</code> . |
| <code>protected void</code> | <code>rehash()</code>                                         | Increases the capacity of and internally reorganizes this hash table, in order to accommodate and access its entries more efficiently.                                                                                 |
| <code>Object</code>         | <code>remove(Object key)</code>                               | Removes the key (and its corresponding value) from this hash table.                                                                                                                                                    |
| <code>int</code>            | <code>size()</code>                                           | Returns the number of keys in this hash table.                                                                                                                                                                         |
| <code>String</code>         | <code>toString()</code>                                       | Returns a string representation of this hash table object in the form of a set of entries, enclosed in braces and separated by the ASCII characters comma and space.                                                   |
| <code>Collection</code>     | <code>values()</code>                                         | Returns a <code>Collection</code> view of the values contained in this hash table.                                                                                                                                     |

FIGURE E.8 Operations on the `Hashtable` class

Creation of a `Hashtable` requires two parameters: initial capacity (with a default of 11) and load factor (with a default of 0.75). The capacity is the number of cells or locations in the initial table. As we noted earlier, the load factor is the maximum percentage occupancy allowed in the hash table before it is resized. `Hashtable` uses the chaining method for resolving collisions.

The `Hashtable` class is a legacy class that will be most useful if you are connecting to legacy code or require synchronization. Otherwise, it is preferable to use the `HashMap` class.

## The `HashSet` Class

The `HashSet` class implements the `Set` interface using a hash table. The `HashSet` class, like most of the Java Collections API implementations of hashing, uses chaining to resolve collisions (each table position effectively being a linked list). The `HashSet` implementation does not guarantee the order of the set on iteration and does not guarantee that the order will remain constant over time. This is because the iterator simply steps through the table in order. Because the hashing function will somewhat randomly distribute the elements to table positions, order cannot be guaranteed. Further, if the table is expanded, all of the elements are rehashed relative to the new table size, and the order may change.

Like the `Hashtable` class, the `HashSet` class requires two parameters: initial capacity and load factor. The default for the load factor is the same as it is for `Hashtable` (0.75). The default for initial capacity is currently unspecified (originally it was 101). Figure E.9 shows the operations for the `HashSet` class. The `HashSet` class is not synchronized and permits null values.

## The `HashMap` Class

The `HashMap` class implements the `Map` interface using a hash table. The `HashMap` class also uses a chaining method to resolve collisions. Like the `HashSet` class, the `HashMap` class is not synchronized and allows null values. Also like the previous implementations, the default load factor is 0.75. Like the `HashSet` class, the current default initial capacity is unspecified, although it was also originally 101.

| Return Value | Method                                                      | Description                                                                                                                                       |
|--------------|-------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------|
|              | <code>HashSet()</code>                                      | Constructs a new, empty set; the backing <code>HashMap</code> instance has the default capacity and load factor, which is 0.75.                   |
|              | <code>HashSet(Collection c)</code>                          | Constructs a new set containing the elements in the specified collection.                                                                         |
|              | <code>HashSet(int initialCapacity)</code>                   | Constructs a new, empty set; the backing <code>HashMap</code> instance has the specified initial capacity and default load factor, which is 0.75. |
|              | <code>HashSet(int initialCapacity, float loadFactor)</code> | Constructs a new, empty set; the backing <code>HashMap</code> instance has the specified initial capacity and the specified load factor.          |
| boolean      | <code>add(Object o)</code>                                  | Adds the specified element to this set if it is not already present.                                                                              |
| void         | <code>clear()</code>                                        | Removes all of the elements from this set.                                                                                                        |
| Object       | <code>clone()</code>                                        | Returns a shallow copy of this <code>HashSet</code> instance: the elements themselves are not cloned.                                             |
| boolean      | <code>contains(Object o)</code>                             | Returns true if this set contains the specified element.                                                                                          |
| boolean      | <code>isEmpty()</code>                                      | Returns true if this set contains no elements.                                                                                                    |
| iterator()   | <code>iterator()</code>                                     | Returns an iterator over the elements in this set.                                                                                                |
| boolean      | <code>remove(Object o)</code>                               | Removes the given element from this set if it is present.                                                                                         |
| int          | <code>size()</code>                                         | Returns the number of elements in this set (its cardinality).                                                                                     |

**FIGURE E.9** Operations on the `HashSet` class

Figure E.10 shows the operations on the `HashMap` class.

## The IdentityHashMap Class

The `IdentityHashMap` class implements the `Map` interface using a hash table. The difference between this and the `HashMap` class is that the `IdentityHashMap` class uses reference-equality instead of object-equality when comparing both keys and values. This is the difference between using `key1==key2` and using `key1.equals(key2)`.

This class has one parameter: expected maximum size. This is the maximum number of key-value pairs that the table is expected to hold. If the table exceeds this maximum, then the table size will be increased and the table entries rehashed.

| Return Value | Method                                                       | Description                                                                                                  |
|--------------|--------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------|
|              | <code>HashMap()</code>                                       | Constructs a new, empty map with a default capacity and load factor, which is 0.75.                          |
|              | <code>HashMap(int initial Capacity)</code>                   | Constructs a new, empty map with the specified initial capacity and default load factor, which is 0.75.      |
|              | <code>HashMap(int initial Capacity, float loadFactor)</code> | Constructs a new, empty map with the specified initial capacity and the specified load factor.               |
|              | <code>HashMap(Map t)</code>                                  | Constructs a new map with the same mappings as the given map.                                                |
| void         | <code>clear()</code>                                         | Removes all mappings from this map.                                                                          |
| Object       | <code>clone()</code>                                         | Returns a shallow copy of this <code>HashMap</code> instance: the keys and values themselves are not cloned. |
| boolean      | <code>containsKey(Object key)</code>                         | Returns true if this map contains a mapping for the specified key.                                           |
| boolean      | <code>containsValue(Object value)</code>                     | Returns true if this map maps one or more keys to the specified value.                                       |
| set          | <code>entrySet()</code>                                      | Returns a collection view of the mappings contained in this map.                                             |
| Object       | <code>get(Object key)</code>                                 | Returns the value to which this map maps the specified key.                                                  |
| boolean      | <code>isEmpty()</code>                                       | Returns true if this map contains no key-value mappings.                                                     |
| Set          | <code>keySet()</code>                                        | Returns a set view of the keys contained in this map.                                                        |
| Object       | <code>put(Object key, Object value)</code>                   | Associates the specified value with the specified key in this map.                                           |
| void         | <code>putAll(Map t)</code>                                   | Copies all of the mappings from the specified map to this one.                                               |
| Object       | <code>remove(Object key)</code>                              | Removes the mapping for this key from this map if present.                                                   |
| int          | <code>size()</code>                                          | Returns the number of key-value mappings in this map.                                                        |
| Collection   | <code>values()</code>                                        | Returns a collection view of the values contained in this map.                                               |

**FIGURE E.10** Operations on the `HashMap` class

Figure E.11 shows the operations on the `IdentityHashMap` class.

## The `WeakHashMap` Class

The `WeakHashMap` class implements the `Map` interface using a hash table. This class is specifically designed with weak keys so that an entry in a `WeakHashMap` will automatically be removed when its key is no longer in use. In other words, if the use of the key in a mapping in the `WeakHashMap` is the only remaining use of the key, the garbage collector will collect it anyway.

| Return Value | Method                                            | Description                                                                                                                                                                   |
|--------------|---------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|              | <code>IdentityHashMap()</code>                    | Constructs a new, empty identity hash map with a default expected maximum size (21).                                                                                          |
|              | <code>IdentityHashMap(int expectedMaxSize)</code> | Constructs a new, empty map with the specified expected maximum size.                                                                                                         |
|              | <code>IdentityHashMap(Map m)</code>               | Constructs a new identity hash map containing the key-value mappings in the specified map.                                                                                    |
| void         | <code>clear()</code>                              | Removes all mappings from this map.                                                                                                                                           |
| Object       | <code>clone()</code>                              | Returns a shallow copy of this identity hash map: the keys and values themselves are not cloned.                                                                              |
| boolean      | <code>containsKey(Object key)</code>              | Tests whether the specified object reference is a key in this identity hash map.                                                                                              |
| boolean      | <code>containsValue(Object value)</code>          | Tests whether the specified object reference is a value in this identity hash map.                                                                                            |
| Set          | <code>entrySet()</code>                           | Returns a set view of the mappings contained in this map.                                                                                                                     |
| boolean      | <code>equals(Object o)</code>                     | Compares the specified object with this map for equality.                                                                                                                     |
| Object       | <code>get(Object key)</code>                      | Returns the value to which the specified key is mapped in this identity hash map, or null if the map contains no mapping for this key.                                        |
| int          | <code>hashCode()</code>                           | Returns the hash code value for this map.                                                                                                                                     |
| boolean      | <code>isEmpty()</code>                            | Returns true if this identity hash map contains no key-value mappings.                                                                                                        |
| Set          | <code>keySet()</code>                             | Returns an identity-based set view of the keys contained in this map.                                                                                                         |
| Object       | <code>put(Object key, Object value)</code>        | Associates the specified value with the specified key in this identity hash map.                                                                                              |
| void         | <code>putAll(Map t)</code>                        | Copies all of the mappings from the specified map to this map. These mappings will replace any mappings that this map had for any of the keys currently in the specified map. |
| Object       | <code>remove(Object key)</code>                   | Removes the mapping for this key from this map if present.                                                                                                                    |
| int          | <code>size()</code>                               | Returns the number of key-value mappings in this identity hash map.                                                                                                           |
| Collection   | <code>values()</code>                             | Returns a collection view of the values contained in this map.                                                                                                                |

**FIGURE E.11** Operations on the `IdentityHashMap` class

| Return Value | Method                                                          | Description                                                                                                                                                                   |
|--------------|-----------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|              | <code>WeakHashMap()</code>                                      | Constructs a new, empty <code>WeakHashMap</code> with the default initial capacity and the default load factor, which is 0.75.                                                |
|              | <code>WeakHashMap(int initialCapacity)</code>                   | Constructs a new, empty <code>WeakHashMap</code> with the given initial capacity and the default load factor, which is 0.75.                                                  |
|              | <code>WeakHashMap(int initialCapacity, float loadFactor)</code> | Constructs a new, empty <code>WeakHashMap</code> with the given initial capacity and the given load factor.                                                                   |
|              | <code>WeakHashMap(Map t)</code>                                 | Constructs a new <code>WeakHashMap</code> with the same mappings as the specified map.                                                                                        |
| void         | <code>clear()</code>                                            | Removes all mappings from this map.                                                                                                                                           |
| boolean      | <code>containsKey(Object key)</code>                            | Returns true if this map contains a mapping for the specified key.                                                                                                            |
| Set          | <code>entrySet()</code>                                         | Returns a set view of the mappings in this map.                                                                                                                               |
| Object       | <code>get(Object key)</code>                                    | Returns the value to which this map maps the specified key.                                                                                                                   |
| boolean      | <code>isEmpty()</code>                                          | Returns true if this map contains no key-value mappings.                                                                                                                      |
| Set          | <code>keySet()</code>                                           | Returns a set view of the keys contained in this map.                                                                                                                         |
| Object       | <code>put(Object key, Object value)</code>                      | Associates the specified value with the specified key in this map.                                                                                                            |
| void         | <code>putAll(Map t)</code>                                      | Copies all of the mappings from the specified map to this map. These mappings will replace any mappings that this map had for any of the keys currently in the specified map. |
| Object       | <code>remove(Object key)</code>                                 | Removes the mapping for the given key from this map, if present.                                                                                                              |
| int          | <code>size()</code>                                             | Returns the number of key-value mappings in this map.                                                                                                                         |
| Collection   | <code>values()</code>                                           | Returns a collection view of the values contained in this map.                                                                                                                |

**FIGURE E.12** Operations on the `WeakHashMap` class

The `WeakHashMap` class allows both null values and null keys, and it has the same tuning parameters as the `HashMap` class: initial capacity and load factor.

Figure E.12 shows the operations on the `WeakHashMap` class.

## LinkedHashSet and LinkedHashMap

The two remaining hashing implementations are extensions of previous classes. The `LinkedHashSet` class extends the `HashSet` class, and the `LinkedHashMap` class extends the `HashMap` class. Both of them are designed to solve the problem of iterator order. These implementations maintain a doubly linked list running through the

| Return Value | Method                                                            | Description                                                                                                     |
|--------------|-------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------|
|              | <code>LinkedHashSet()</code>                                      | Constructs a new, empty linked hash set with the default initial capacity (16) and load factor (0.75).          |
|              | <code>LinkedHashSet(Collection c)</code>                          | Constructs a new linked hash set with the same elements as the specified collection.                            |
|              | <code>LinkedHashSet(int initialCapacity)</code>                   | Constructs a new, empty linked hash set with the specified initial capacity and the default load factor (0.75). |
|              | <code>LinkedHashSet(int initialCapacity, float loadFactor)</code> | Constructs a new, empty linked hash set with the specified initial capacity and load factor.                    |

**FIGURE E.13** Additional operations on the `LinkedHashSet` class

entries to maintain the insertion order of the elements. Thus the iterator order for these implementations is the order in which the elements were inserted.

Figure E.13 shows the additional operations on the `LinkedHashSet` class. Figure E.14 shows the additional operations on the `LinkedHashMap` class.

| Return Value      | Method                                                                                 | Description                                                                                                                                     |
|-------------------|----------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
|                   | <code>LinkedHashMap()</code>                                                           | Constructs an empty insertion-ordered <code>LinkedHashMap</code> instance with a default capacity (16) and load factor (0.75).                  |
|                   | <code>LinkedHashMap(int initialCapacity)</code>                                        | Constructs an empty insertion-ordered <code>LinkedHashMap</code> instance with the specified initial capacity and a default load factor (0.75). |
|                   | <code>LinkedHashMap(int initialCapacity, float loadFactor)</code>                      | Constructs an empty insertion-ordered <code>LinkedHashMap</code> instance with the specified initial capacity and load factor.                  |
|                   | <code>LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)</code> | Constructs an empty <code>LinkedHashMap</code> instance with the specified initial capacity, load factor, and ordering mode.                    |
|                   | <code>LinkedHashMap(Map m)</code>                                                      | Constructs an insertion-ordered <code>LinkedHashMap</code> instance with the same mappings as the specified map.                                |
| void              | <code>clear()</code>                                                                   | Removes all mappings from this map.                                                                                                             |
| boolean           | <code>containsValue(Object value)</code>                                               | Returns true if this map maps one or more keys to the specified value.                                                                          |
| Object            | <code>get(Object key)</code>                                                           | Returns the value to which this map maps the specified key.                                                                                     |
| protected boolean | <code>removeEldestEntry(Map.Entry eldest)</code>                                       | Returns true if this map should remove its eldest entry.                                                                                        |

**FIGURE E.14** Additional operations on the `LinkedHashMap` class



## Summary of Key Concepts

- In hashing, elements are stored in a hash table, and their location in the table is determined by a hashing function.
- The situation where two elements or keys map to the same location in the table is called a collision.
- A hashing function that maps each element to a unique position in the table is said to be a perfect hashing function.
- Extraction involves using only a part of the element's value or key to compute the location at which to store the element.
- The division method is very effective when one is dealing with an unknown set of key values.
- In the shift folding method, the parts of the key are added together to create the index.
- The length-dependent method and the mid-square method may be effectively used with strings by manipulating the binary representations of the characters in the string.
- Although Java provides a `hashCode` method for all objects, it is often preferable to define a specific hashing function for any particular class.
- The chaining method for handling collisions simply treats the hash table conceptually as a table of collections, rather than as a table of individual cells.
- The open addressing method for handling collisions looks for an open position in the table other than the one to which the element is originally hashed.
- The load factor is the maximum percentage occupancy allowed in the hash table before it is resized.

### Self-Review Questions

- SR E.1 What is the difference between a hash table and the other collections we have discussed?
- SR E.2 What is a collision in a hash table?
- SR E.3 What is a perfect hashing function?
- SR E.4 What is our goal for a hashing function?
- SR E.5 What is the consequence of not having a good hashing function?
- SR E.6 What is the extraction method?
- SR E.7 What is the division method?

- SR E.8 What is the shift folding method?
- SR E.9 What is the boundary folding method?
- SR E.10 What is the mid-square method?
- SR E.11 What is the radix transformation method?
- SR E.12 What is the digit analysis method?
- SR E.13 What is the length-dependent method?
- SR E.14 What is chaining?
- SR E.15 What is open addressing?
- SR E.16 What are linear probing, quadratic probing, and double hashing?
- SR E.17 Why is deletion from an open addressing implementation a problem?
- SR E.18 What is the load factor, and how does it affect table size?

## Exercises

- EX E.1 Draw the hash table that results from adding the following integers (34 45 3 87 65 32 1 12 17) to a hash table of size 11 using the division method and linked chaining.
- EX E.2 Draw the hash table from Exercise E.1 using a hash table of size 11 and array chaining with a total array size of 20.
- EX E.3 Draw the hash table from Exercise E.1 using a table size of 17 and open addressing with linear probing.
- EX E.4 Draw the hash table from Exercise E.1 using a table size of 17 and open addressing with quadratic probing.
- EX E.5 Draw the hash table from Exercise E.1 using a table size of 17 and double hashing using extraction of the first digit as the secondary hashing function.
- EX E.6 Draw the hash table that results from adding the following integers (1983, 2312, 6543, 2134, 3498, 7654, 1234, 5678, 6789) to a hash table using shift folding of the first two digits with the last two digits. Use a table size of 13.
- EX E.7 Draw the hash table from Exercise E.6 using boundary folding.
- EX E.8 Draw a UML diagram that shows how all of the various implementations of hashing within the Java Collections API are constructed.

## Programming Projects

- PP E.1 Implement the hash table illustrated in Figure E.1 using the array version of chaining.
- PP E.2 Implement the hash table illustrated in Figure E.1 using the linked version of chaining.
- PP E.3 Implement the hash table illustrated in Figure E.1 using open addressing with linear probing.
- PP E.4 Implement a dynamically resizable hash table to store people's names and Social Security numbers. Use the extraction method with division using the last four digits of the Social Security number. Use an initial table size of 31 and a load factor of 0.80. Use open addressing with double hashing using an extraction method on the first three digits of the Social Security number.
- PP E.5 Implement the problem from Programming Project E.4 using linked chaining.
- PP E.6 Implement the problem from Programming Project E.4 using the `HashMap` class of the Java Collections API.
- PP E.7 Create a new implementation of the bag collection called `HashtableBag` using a hash table.
- PP E.8 Implement the problem from Programming Project E.4 using shift folding with the Social Security number divided into three equal three-digit parts.
- PP E.9 Create a graphical system that will allow a user to add and remove employees where each employee has an employee id (a six-digit number), an employee name, and years of service. Use the `hashCode` method of the `Integer` class as your hashing function, and use one of the Java Collections API implementations of hashing.
- PP E.10 Complete Programming Project E.9 using your own `hashCode` function. Use extraction of the first three digits of the employee id as the hashing function, and use one of the Java Collections API implementations of hashing.
- PP E.11 Complete Programming Project E.9 using your own `hashCode` function and your own implementation of a hash table.
- PP E.12 Create a system that will allow a user to add and remove vehicles from an inventory system. Vehicles will be represented by license number (an eight-character string), make, model, and color. Use your own array-based implementation of a hash table using chaining.

PP E.13 Complete Programming Project E.12 using a linked implementation with open addressing and double hashing.

## Answers to Self-Review Questions

- SRA E.1 Elements are placed into a hash table at an index produced by a function of the value of the element or a key of the element. This is different from other collections, where the position/location of an element in the collection is determined either by comparison with the other values in the collection or by the order in which the elements were added or removed from the collection.
- SRA E.2 The situation where two elements or keys map to the same location in the table is called a collision.
- SRA E.3 A hashing function that maps each element to a unique position in the table is said to be a perfect hashing function.
- SRA E.4 We need a hashing function that will do a good job of distributing elements into positions in the table.
- SRA E.5 If we do not have a good hashing function, the result will be too many elements mapped to the same location in the table. This will result in poor performance.
- SRA E.6 Extraction involves using only a part of the element's value or key to compute the location at which to store the element.
- SRA E.7 The division method involves dividing the key by some positive integer  $p$  (usually the table size and usually prime) and then using the remainder as the index.
- SRA E.8 Shift folding involves dividing the key into parts (usually the same length as the desired index) and then adding the parts. Extraction or division is then used to get an index within the bounds of the table.
- SRA E.9 Like shift folding, boundary folding involves dividing the key into parts (usually the same length as the desired index). However, some of the parts are then reversed before adding. One example is to imagine that the parts are written side by side on a piece of paper, which is then folded on the boundaries between parts. In this way, every other part is reversed.
- SRA E.10 The mid-square method involves multiplying the key by itself and then extracting some number of digits or bytes from the middle of the result. Division can then be used to guarantee an index within the bounds of the table.

- SRA E.11 The radix transformation method is a variation on the division method where the key is first converted to another numeric base and then divided by the table size with the remainder used as the index.
- SRA E.12 In the digit analysis method, the index is formed by extracting, and then manipulating, specific digits from the key.
- SRA E.13 In the length-dependent method, the key and the length of the key are combined in some way to form either the index itself or an intermediate value that is then used with one of our other methods to form the index.
- SRA E.14 The chaining method for handling collisions simply treats the hash table conceptually as a table of collections, rather than as a table of individual cells. Thus each cell is a pointer to the collection associated with that location in the table. This internal collection usually is either an unordered list or an ordered list.
- SRA E.15 The open addressing method for handling collisions looks for an open position in the table other than the one to which the element is originally hashed.
- SRA E.16 Linear probing, quadratic probing, and double hashing are methods for determining the next table position to try if the original hash causes a collision.
- SRA E.17 Because of the way a path is formed in open addressing, deleting an element from the middle of that path can cause elements beyond that point on the path to be unreachable.
- SRA E.18 The load factor is the maximum percentage occupancy allowed in the hash table before it is resized. Once the load factor has been reached, a new table is created that is twice the size of the current table, and then all of the elements in the current table are inserted into the new table.



# Regular Expressions

F

Appendix

Throughout this text, we've used the `Scanner` class to read interactive input from the user and parse strings into individual tokens such as words. Usually we used the default whitespace delimiters for tokens in the scanner input.

The `Scanner` class can also be used to parse its input according to a *regular expression*, which is a character string that represents a pattern. A regular expression can be used to set the delimiters used when extracting tokens, or it can be used in methods such as `findInLine` to match a particular string.

Here are some of the general rules for constructing regular expressions:

- The dot (.) character matches any single character.
- The asterisk (\*) character, which is called the Kleene star, matches zero or more characters.
- A string of characters in brackets ([]) matches any single character in the string.
- The \ character followed by a special character (such as one of those in this list) matches the character itself.
- The \ character followed by a character matches the pattern specified by that character (see Figure F.1).

| Regular Expression  | Matches                                       |
|---------------------|-----------------------------------------------|
| x                   | The character x                               |
| .                   | Any character                                 |
| [ abc ]             | a, b, or c                                    |
| [ ^abc ]            | Any character except a, b, or c (negation)    |
| [ a-z ] [ A-Z ]     | a through z or A through Z, inclusive (range) |
| [ a-o [ m-p ] ]     | a through d or m through p (union)            |
| [ a-z && [ def ] ]  | d, e, or f (intersection)                     |
| [ a-z && [ ^bc ] ]  | a through z, except for b and c (subtraction) |
| [ a-z && [ ^m-p ] ] | a through z but not m through p (subtraction) |
| \d                  | A digit: [0-9]                                |
| \D                  | A non-digit: [^0-9]                           |
| \s                  | A whitespace character                        |
| \S                  | A non-whitespace character                    |
| ^                   | The beginning of a line                       |
| \$                  | The end of a line                             |

**FIGURE F.1** Some patterns that can be specified in a Java regular expression

For example, the regular expression `B.b*` matches `Bob`, `Bubba`, and `Baby`. The regular expression `T[aei]*ing` matches `Taking`, `Tickling`, and `Telling`.

Figure F.1 shows some of the patterns that can be matched in a Java regular expression. This list is not complete—see the online documentation for the `Pattern` class for a complete list.



*This page is intentionally left blank.*

# Index

## Symbols

#, protected class visibility, 457  
\$, end of a line expression, 662  
\*, Kleene star expression, 662  
\*, multiplication operator, 65  
\*/, close comment indicator, 71–72  
-, private class visibility, 457  
., dot expression matching any character, 662  
., dot operator, 467  
/\*\*, open comment indicator, 71–72  
@, Javadoc tags, 72  
\d, digit (0-9) expression, 662  
\D, non-digit (0-9) expression, 662  
\s, non-whitespace character expression, 662  
\s, whitespace character expression, 662  
^, beginning of a line expression, 662  
{ }, nonpublic class visibility, 457  
+, addition operator, 65  
+, public class visibility, 457  
< >, stereotypes, 457  
==, object comparison operator, 481–482

## A

Abstract class, 494–495  
Abstract data types (ADT), 57–61, 74–76, 87, 135–137, 172–178, 285–289, 320–322, 358–362

binary search trees, 320–322  
binary trees, 285–289  
collections, 57–61  
data structures compared  
    to, 58–59  
heaps, 358–362  
lists, 172–178  
operations, 59–60  
queues, 135–137  
stacks, 59–61, 74–76  
Abstract Windowing Toolkit (AWT), 539  
Abstraction, 57–58, 87, 465–466  
Action events, 543–548  
Activation record, 102–103, 118  
Acyclic graph, 429, 451  
Adapter classes, events, 604–605  
add() operations, 126, 158, 173, 183–184, 333, 336, 387  
    array-based implementation, 183–184  
    balance and, 336  
    binary search trees, 333, 336  
    linked structure implementation, 333, 336  
    lists, 158, 173, 183–184, 333, 336  
    sets, 387  
addAfter operation, 173, 185  
addAll operation, 387  
addEdge method, 448  
addEdge method, 448–449

- addElement operation, 320, 323–325, 358, 360–361, 368–370, 375–376
- array-based implementation using, 375–376
- binary search trees, 320, 323–325
- heaps, 358, 360–361, 368–370, 375–376
- linked implementation using, 323–325, 368–370
- Addition (+) operator, 65
- addToFront operation, 173, 185
- addToRear operation, 173, 185
- addVertex method, 448–449
- Adjacency lists, 442
- Adjacency matrices, 442–450
- Adjacent vertices, 428, 451
- Aggregation, UML relationship of, 459–460
- Algorithms, 41–54, 233–234, 248–249, 432–441
  - analysis of, 41–54, 233–234
  - asymptotic complexity of, 43–45, 52
  - Big-Oh notation, 43–45
  - complexity of, 43–45, 48–52
  - connectivity, testing for, 436–437
  - efficiency of, 42–43
  - exponential complexity, 47
  - graphs, 432–441
  - growth functions, 43–47
  - logarithmic, 249
  - loops, 48–49
  - method calls, 49–51
  - minimum spanning trees, 438–440
  - processor speed and, 45–47
  - recursive, 233–234
  - search, comparison of, 248–249
  - shortest path determination, 441
  - speed increase and, 45–47
  - time complexity of, 48–51
  - traversals, 432–436
- Aliases, 481–482
- Ancestors, 276–277, 314
- Application programming interface (API), 59
- Arc angle, 519
- ArrayList class, 158, 178–186, 205–207
- Arrays, 77–86, 143–150, 178–186, 205–207, 279–281, 373–378
  - add() operation, 183–184
  - addAfter operation, 185
  - addElement operation, 375–376
  - addToFront operation, 185
  - addToRear operation, 185
  - assumptions for implementation, 79
  - capacity, 78
  - circular implementation strategy, 144–150
  - computational strategy for, 279–280
  - constructors, 80–81
  - contains() operation, 182–183
  - dequeue operation, 143, 149–150
  - enqueue operation, 143, 147–149
  - exceptions for, 84–86, 205–207
  - expandCapacity method, 82–83, 149
  - find() method, 181–182
  - fixed implementation strategy, 77–86, 143–144
  - freelist, 280, 315
  - generic types, errors from, 81
  - heap implementation using, 373–378
  - iterator implementation using, 205–207
  - linked strategy for, 279–281
  - list implementation using, 178–186
  - modification count for, 205
  - object references, 77
  - peek operation, 85
  - pop operation, 83–85
  - push operation, 82–83

- queue implementation using, 143–150
- `remove()` operations, 180–182
- `removeMin` operation, 376–378
- stack implementation using, 77–86
- tree implementation using, 279–281, 305–313

`ArrayStack` class, 79–85

Association, UML relationship of, 459–460

Asymptotic complexity, 43–45, 52.  
*See also* Algorithms

Attributes, 456, 469

AVL tree, 340–343, 350

## B

- B-trees, 418–421, 422
- Background colors, 519
- Balance factor, 340–343, 350
- Base case, recursion, 215–216, 235
- Base class, 489
- Behavior of objects, 469–470
- Bevel border, 615–616
- Big-Oh notation, 43–45, 48–49, 52, 143
  - asymptotic complexity and, 43–45
  - growth functions and, 43–45
  - loops, 48–49
  - queue operation complexity, 143
- Binary search, 246–248, 249, 270
- Binary search trees, 36, 281, 319–356
  - abstract data types (ADT), 320–322
  - `addElement` operation, 320, 323–325
  - AVL trees, 340–343, 350
  - balance factor for, 340–343, 350
  - balanced, 336–340
  - data structure use of, 36
  - degenerate, 336–337, 350
  - efficiency of implementation of, 281
  - exceptions, 323, 326, 329, `findMax` operation, 320

- `findMin` operation, 320
- implementation of, 322–332, 340–349
- linked implementation, 322–332
- ordered lists implemented using, 332–336
- promoted nodes, 326–329
- red/black trees, 343–349, 351
- `removeAllOccurrences` operation, 320, 329–330
- `removeElement` operation, 320, 326–329
- `removeMax` operation, 320
- `removeMin` operation, 320, 330–332
- replacement method, 326–329
- rotation techniques for, 337–343
- unbalanced, 340–343

Binary trees, 277, 285–313, 315.  
*See also*, Binary search trees; Heaps

- abstract data types (ADT), 285–289
- classification as, 277, 315
- `contains` operation, 285
- decision trees from, 301–305
- evaluate methods, 290–294, 304
- expression trees from, 289–300
- find operation, 285, 310–312
- `getRoot` operation, 285
- implementation of, 305–313
- `isEmpty` operation, 285
- iterator operations, 285, 312–313
- linked structures and, 285–313
- `LinkedBinaryTree` class, 290–313
- postfix expressions and, 295–300
- `size()` operation, 285
- `toString` operation, 285

`BinaryTreeNode` class, 307–310

boolean `hasNext()` method, 196

Border layout, 574, 579–583

Borders, 612–616

Bounding rectangle, 519

Box layout, 574, 586–589

- Breadth-first traversal, 432–433, 436–437, 451
- Bucket, hash tables, 402, 404, 634
- Buttons, 543–548, 555–559
  - action events from, 543–548
  - labels for, 543
  - listener interface for, 544
  - push, 543–548
  - radio, 555–559
- Bytecode, 32, 37
- C**
- Caesar cipher, 126–127, 151
- Call stack trace, 103, 502
- Capacity of arrays, 78
- catch clause, 502–503
- Cell, hash tables, 402, 404, 634
- Chaining method, hashing, 639–642, 646
- Check boxes, 551–554
- Child class (subclass), 489, 492
- Children, 276, 314, 492
- Circular array implementation strategy, 144–151
- Class diagrams, UML, 456–458
- `ClassCastException` class, 329–330, 360
- Classes, 459–460, 464, 468–469, 470–473, 485–486, 488–505
  - abstract class and, 494–495
  - base, 489
  - child (subclass), 489, 492
  - data declarations, 471
  - declaration, 468–469
  - definition of objects, 464
  - derived, 488–490
  - exceptions, 501–505
  - hierarchies, 62–63, 88, 492–495, 497, 504–505
  - import declaration, 468–469
  - inheritance, 464, 488–492
  - instance data, 473
  - is-a relationship, 489–490
  - Java application programmer interface (API), 468
  - libraries, 468
  - members, 471
  - method declarations, 471
  - Object class and, 493–494
  - object-oriented design and, 464, 468–469, 470–473
  - overriding methods and, 491–492
  - packages, 468–469
  - parent (superclass), 489, 492
  - polymorphism, 496–501
  - protected visibility, 490–491
  - super reference and, 491
  - UML relationship of, 459–460
  - using each other, 459–460
  - visibility modifiers, 457, 490–491
  - wrapper, 485–486
- `clear()` operation, 387, 388
- Client, 28, 474
- Code keys, 126–130
  - Caesar cipher, 126–127
  - queues and, 126–130
  - repeating key, 127
- Collections, 55–92, 196–198, 210, 385–408
  - abstract data types (ADT), 57–61, 74–76, 87
  - abstraction as, 57–58
  - application programming interface (API), 59
  - data structure, 58–59, 88
  - data types, 57–64, 74–76, 87
  - elements of, 56
  - encapsulation of, 58, 88
  - exceptions, 72–74

- fail-fast iterator implementation, 198, 210
- interface operations and, 57–58, 74–76, 88
- iterable interface for, 196–197, 210
- iterators and, 196–198, 210
- Java API, 59, 88, 386–389
- linear, 56–57, 276
- maps, 385–408
- nonlinear, 56–57, 276
- objects, 56–58, 61–64
- postfix expressions, 64–72
- sets, 385–408
- stacks, 59–87
- Collisions, 402, 404, 634, 639–645
  - avoidance of, 639–645
  - chaining method, 639–642
  - hash tables, 402, 404, 634
  - open addressing methods, 642–645
- Color chooser dialog boxes, 611
- Color representation, 517
- Combo boxes, 564–569
- Command-line applications, 538
- Commercial Off-The-Shelf (COTS) products, 31–32
- Comparable interface, 182–183, 242, 251, 487–488
  - collection implementation and, 182
  - object-oriented design and, 182, 487–488
  - objects, 183–184
  - searching, 242
  - sorting, 251
- Complete graph, 428, 451
- Complete tree, 278, 315, 360–361, 381
- Components, 523–527, 538–574, 605–623
  - borders, 612–616
  - check boxes, 551–554
  - combo boxes, 564–569
  - dialog boxes, 605–611
  - disabled, 617
  - drawing on, 523–527
  - graphical user interfaces (GUI), 539–574, 605–623
  - labels, 543
  - mnemonics, 616–623
  - radio buttons, 555–559
  - sliders, 559–564
  - Swing package, 539, 605–623
  - text area, 608–610
  - text fields, 548–551
  - timers, 569–574
  - tool tips, 616–623
- Compound border, 615–616
- Computational strategy for array implementation of trees, 279–280
- ConcurrentModificationException class, 198, 203, 205–206
- Confirm dialog boxes, 605, 608
- Connected graph, 429–430, 451
- Connectivity, testing for in graphs, 436–437
- Constructors, 80–81, 111, 476–477
  - array-based implementation, 80–81
  - default, 477
  - object-oriented design and, 476–477
  - linked-list implementation, 111
  - stacks, 80–81, 111
- Containers, 539–543, 574–589
  - applets, 539
  - frames, 539–543
  - heavyweight, 539, 541
  - layout managers, 543, 574–589
  - lightweight, 539
  - panels, 539–543
- Containment hierarchies, 589

- contains() operation, 172, 182–183, 285, 387
  - array-based implementation, 182–183
  - binary trees, 285
  - lists, 172, 182–183
  - sets, 387
- containsAll operation, 387
- containsKey operation, 388
- containsValue operation, 388
- Coordinates, 516
- Correctness, 29, 37
- count method, 48–49
- Cycle, 429, 451
- D**
- Data declarations, 471
- Data structures, 33–36, 57–59, 88
  - collections, 57–59, 88
  - hash tables, 35
  - lists, 36
  - maps, 36
  - objects, 36
  - queues, 36
  - search trees, 36
  - shipping container scenario, 33–36
  - stacks, 35
- Data types, 57–64, 74–76, 87
  - abstract (ADT), 57–61, 74–76, 87
  - generic, 63–64, 81, 88
  - object-oriented, 61–64
  - primitive, 58
  - type checking, 62
  - type compatibility, 61–62
- Decision trees, 301–305
- Default constructor, 477
- Degenerate tree, 336–337, 350
- Depth-first traversal, 432–436, 451
- Deque interface, 100–101, 150
- dequeue operation, 124–125, 141–143, 149–151
  - array-based implementation, 143, 149–150
  - Big-Oh complexity of, 143
  - queue process, 124–125, 151
  - linked-list implementation, 141–142
- Derived classes, 488–490
- Descendants, 277, 314
- Dialog boxes, 605–611
  - color choosers, 611
  - confirm, 605, 608
  - file choosers, 608–611
  - input, 605–606
  - message, 605, 607
  - text area, 608–610
- Digit analysis method, hash functions, 638
- Direct recursion, 219–220, 235
- Directed graph (digraph), 429–430
- Division method, hash functions, 636–637
- Domains blocked using sets, 389–392
- Dot operator ., 467
- Double-ended queues (deque), 150
- Double hashing, 644–645
- Doubly-linked lists, 99–100, 118
- Drawing shapes, 518–532
  - arc angle, 519
  - background and foreground colors, 519
  - bounding rectangle, 519
  - component methods, 523–527
  - graphics content interaction, 520
  - methods for, 518–519
  - MID value, 523
  - polygons, 527–532
  - polylines, 527–532
  - start angle, 519
  - TOP value, 523
- Drop-out stack, 76
- Dummy nodes, 98

- Dynamic (late) binding, 497
- Dynamic structure, 96
- E**
- `E Next()` method, 196
- Edges (connections), 428–430, 451
- Edges, 276, 314
- Efficiency, 32, 37, 281
  - software quality and, 32, 37
  - tree implementation and, 281
- `ElementNotFoundException` class, 326, 329
- Elements, 56–57, 96–97, 99–100, 137–142, 198, 344–347.
  - See also* Nodes; Objects
  - accessing, 96–97
  - adding to lists, 173–178
  - characteristics of, 156
  - collections of, 56–57, 96–97, 99–100
  - head/tail references, 137–142
  - insertion into trees, 344–347
  - iterator order of, 198
  - lists, 156–158, 173–178
  - numerical reference for, 156–158
  - organization of, 56–57
  - queues, 137–142
  - red/black trees, 344–349
  - removal from trees, 347–349
  - separation of stored and linked, 99–100
  - unordered list placement, 156–157
- Empty border, 615
- `EmptyCollectionException` class, 84–86, 141–142
  - array-based implementation, 84–86
  - linked-list implementation, 141–142
- Encapsulation, 58, 88, 464, 474–476
  - collections and, 58, 88
  - local data for, 476
  - object reference variables and, 464, 474
  - visibility modifiers, 474–476
- Engineering goals, 28
- `enqueue` operation, 124–125, 139–140, 143, 147–149, 151
  - array-based implementation, 143, 147–149
  - Big-Oh complexity of, 143
  - queue process, 124–125, 151
  - linked-list implementation, 139–140
- `entrySet()` operation, 388
- `equals()` method, 181–182, 387, 388, 482
- Errors, 73, 81, 216, 501–502
  - base case in recursion, 216
  - exception messages and, 502
  - exceptions compared to, 73, 501
  - generic types in arrays, 81
- Etched border, 615
- `evaluate` method, 70, 290–294, 304
- Events, 538, 543–548, 589–605
  - action, 543–548
  - adapter classes for, 604–605
  - buttons, 543–548
  - item, 551
  - key, 598–604
  - listener interface for, 544–548, 594–598
  - motion (mouse), 589–598
  - mouse, 589–598
  - sources, 545–548
- Exception handler, 502
- Exceptions, 72–74, 84–86, 88, 182, 185, 198, 203, 205–207, 323, 326, 329–330, 501–505
  - array-based implementation and, 84–86, 182, 185, 205–207
  - call stack trace, 502
  - class hierarchy, 504–505
  - `ClassCastException` class, 329–330
  - classes of, 85–86
  - `ConcurrentModificationException` class, 198, 203, 205–206



Exceptions (*Continued*)

- ElementNotFoundException class, 326, 329
- EmptyCollectionException class, 84–86, 141–142
- errors compared to, 73, 501
- iterators and, 198, 203, 205–207
- lists, 182, 185
- messages, 502
- NonComparableElementException class, 323
- NoSuchElementException class, 206–207
- object-oriented design and, 501–505
- objects as, 72–74, 88, 501
- propagation, 503–504
- RuntimeException class, 86
- stacks, 72–74, 84–86
- try statement, 502–503
- UnsupportedOperationException class, 207
- expandCapacity method, 82–83, 149, 449–450
- graph adjacency matrix
  - implementation, 449–450
- queue implementation, 149
- stack implementation, 82–83
- Expert system, 301
- Exponential complexity, 47
- Expression trees, 289–300
- Expressions, 661–663

**F**

- Factorial (N!), 215–216
- Fail-fast implementation, 198, 210
- Failure of software, 30
- File chooser dialog boxes, 608–611
- find() operation, 181–182, 285, 310–312
- arrays, 181–182

- binary trees, 285, 310–312
- lists, 181–182
- findAgain method, 310
- FindMax operation, 320
- FindMin operation, 320, 358, 362, 373, 378
- First in, first out (FIFO) process, 124, 151
- first operation, 124–125, 142, 172
- lists, 172
- queues, 124–125, 142
- Fixed array implementation strategy, 77–86, 143–144
- Flow layout, 574, 576–579
- Folding method, hash functions, 637
- for loop, 258
- for-each loops, 197, 202–203
- Foreground colors, 519
- Frames, 539–543
- Free store, 96
- Freelist, 280, 315
- Full tree, 278, 315

**G**

- Garbage collection, 482–483
- General (n-ary) trees, 277, 315
- Generic search method, 243–244, 270
- Generic type, 63–64, 81, 88
- get() method, 388
- getNewLastNode method, 370–373
- getNextParentAdd method, 368–370
- getRoot operation, 285
- Graphical user interfaces (GUI), 537–631
  - Abstract Windowing Toolkit (AWT), 539
  - action events, 543–548
  - borders, 612–616
  - buttons, 543–548, 555–559
  - check boxes, 551–554

- combo boxes, 564–569
  - command-line applications compared
    - to, 538
  - components, 538–574, 605–623
  - containers, 539–543
  - design, 538–539, 623
  - dialog boxes, 605–611
  - event adapter class, 604–605
  - event-driven programs, 538
  - events, 538, 543–548, 551, 589–604
  - frames, 539–543
  - key events, 598–604
  - layout managers, 543, 574–589
  - listeners, 538, 544–548, 594–598
  - mnemonics, 616–623
  - mouse events, 589–598
  - panels, 539–543
  - sliders, 559–564
  - Swing package components, 539, 605–623
  - text area, 608–610
  - text fields, 548–551
  - timers, 569–574
  - tool tips, 616–623
  - Graphics, *see* Java graphics
  - Graphics class, 518–523
  - Graphs, 427–454
    - addEdge method, 448
    - addVertex method, 448–449
    - adjacency lists, 442
    - adjacency matrices, 442–450
    - algorithms, 432–441
    - connectivity, testing for, 436–437
    - directed (digraph), 429–430
    - edges (connections), 428–430, 451
    - expandCapacity method, 448–449
    - implementation strategies, 441–450
    - minimum spanning trees (MST), 438–440
    - networks (weighted graphs), 431–432, 451
    - path, 429–430, 452
    - shortest path determination, 441
    - traversals, 432–436
    - undirected, 428–429, 443–450
    - vertices (nodes), 428–430, 442–443, 452
  - Grid layout, 574, 583–585
  - Growth functions, 43–47
    - asymptotic complexity and, 43–45
    - Big-Oh notation and, 43–45
    - exponential complexity and, 47
    - speed increase (tenfold) using, 45–47
    - time complexity and, 45–47
- ## H
- Hash functions, 402–403, 404, 634–639
    - digit analysis method, 638
    - division method, 636–637
    - folding method, 637
    - initial capacity, 402, 404
    - Java language and, 639
    - length-dependent method, 638
    - load factor, 402, 404, 636
    - mapping elements using, 402–403
    - mid-square method, 637–638
    - perfect, 402, 634–635
    - radix transformation method, 638
  - Hash tables, 35, 402, 404, 634–636, 646–655
    - bucket, 402, 404, 634
    - cell, 402, 404, 634
    - collision, 402, 404, 634, 639–645
    - deleting elements from, 646–648
    - chain implementation, 639–642, 646
    - dynamic resizing, 636
    - Java API classes, 648–655
    - open address implementation, 642–645, 647–648

- hashCode() method, 387, 388
  - Hashing, 401–403, 404, 633–660.
    - See also* Hash functions; Hash tables
    - chaining method, 639–642, 646
    - collision, 402, 404, 634, 639–645
    - double hashing, 644–645
    - functions, 402–403, 404, 634–639
    - linear probing, 642–643
    - map implementation using, 401–403
    - open addressing methods, 642–645, 647–648
    - quadratic probing, 642–644
    - set implementation using, 401–403
    - tables, 35, 402, 404, 634–636, 646–655
  - HashMap class, 650, 652
  - HashSet class, 650, 651
  - Hashtable class, 648–650
  - hasNext method, 205, 207
  - Head/tail element references, 137–142
  - heapifyAdd method, 368–370, 375–376
  - heapifyRemove method, 370–372, 376–378
  - Heaps, 357–384
    - abstract data types (ADT), 358–362
    - addElement operation, 358, 360–361, 368–370, 375–376
    - array-based implementation, 373–378
    - findMin operation, 358, 362, 373, 378
    - implementation of, 366–378
    - linked implementation of, 366–375
    - maxheap, 358, 379, 382
    - minheap, 358, 360, 379, 382
    - priority queues using, 362–366
    - removeMin operation, 358, 361–362, 370–373, 376–378
    - sorting with, 378–380
  - heapSort method, 379–380
  - Height of trees, 277, 315
  - Hierarchy, 276–278, 492–496, 497, 504–505, 589.
    - See also* Nodes
    - abstract class and, 494–495
    - ancestors, 276–277, 314
    - children, 276, 314, 492
    - classes, 492–495, 497
    - containment, 589
    - Exception class, 504–505
    - inheritance and, 492–496
    - interfaces, 496
    - Object class and, 493–494
    - polymorphism and, 497
    - siblings, 276, 314, 492
    - trees, 276–278
  - Human-Computer Interaction (HCI), 30–31
- I**
- IdentityHashMap class, 651, 653
  - if-else statements, 217–219
  - Image observer, 603
  - Implementation, 77–86, 110–117, 137–150, 178–188, 279–281, 305–313, 322–349, 363, 366–378, 420–421, 441–450, 459–460
    - add() operation, 183–184
    - addAfter operation, 173, 185
    - addEdge method, 448
    - addElement operation, 320, 323–325, 368–370, 375–376
    - addVertex method, 448–449
    - adjacency lists, 442
    - adjacency matrices, 442–450
    - array capacity, 78
    - array-based, 77–86, 143–150, 178–186, 279–281, 373–378
  - ArrayList class, 178–186
  - ArrayStack class, 79–86
  - AVL trees, 340–343, 350

- B-trees, 420–421
- balanced trees, 336–340
- binary search trees used for, 332–336
- binary search trees, 322–332, 340–349
  - computational strategy for, 279–280
  - constructors, 80–81, 111
  - contains() operation, 172, 182–183
  - dequeue operation, 141–143, 149–150
  - efficiency of, 281
  - enqueue operation, 139–140, 143, 147–149
  - exceptions for, 84–86
  - expandCapacity method, 448–449
  - findMin operation, 373, 378
  - freelist, 280, 315
  - graphs, 441–450
  - heaps, 366–378
  - linked structure strategies, 279–281, 305–313, 322–332, 366–375
  - LinkedList class, 186–188
  - linked-list, 110–117, 137–142, 186–188
  - LinkedList class, 110–117
  - lists, 178–188, 332–336
  - node reference, 110–111
  - object references, 77
  - peek operation, 85, 117
  - pop operation, 83–85, 116–117
  - priority queues, 363
  - push operation, 82–83, 114–116
  - queues, 137–150
  - red/black trees, 343–349, 351
  - remove() operations, 180–182, 187–188
  - removeAllOccurrences operation, 320, 329–330
  - removeElement operation, 320, 326–329
  - removeMin operation, 320, 330–332, 370–373, 376–378
  - rotation techniques for, 336–340
  - size() operation, 117, 142
  - StackADT interface, 74–76, 80, 110
  - stacks, 77–86, 110–117
  - trees, 279–281, 305–313, 322–349
    - UML relationship of, 459–460
- import declaration, 468–469
- Indexed lists, 156–158, 170–172, 189
  - element numerical reference for, 156–158
  - Josephus class example, 170–172
- Indirect recursion, 219–220, 235
- Infinite recursion, 214–215, 235
- Infix notation, 64
- Inheritance, 62–63, 88, 458–459, 464, 488–492, 498–499
  - classes and, 464, 488–492
  - collections and, 62–63, 88
  - derived classes, 488–490
  - object-oriented design and, 464, 488–492
  - overriding methods and, 491–492
  - polymorphism from, 498–499
  - protected visibility and, 490–491
  - super reference and, 491
  - UML relationship of, 458–459
- Initial capacity, hash functions, 402, 404
- Inner class, 544
- Inorder traversal, 282–283, 315
- Input dialog boxes, 605–606
- Insertion points, 360–361
- Insertion sort, 250, 254–256, 270
- Instance data, 473
- Instance variable, 484
- Instantiation, 467

- Interfaces, 57–58, 70, 74–76, 88,
    - 100–101, 125–126, 150, 158–159,
    - 161, 196–198, 242, 251, 486–488,
    - 496, 498, 500–501
  - abstract method, 486–487
  - abstraction and, 57–58
  - collections, 57–58, 74–76, 88
  - Comparable, 242, 251, 487–488
  - constants for, 487
  - class implementation of, 486–487
  - Deque, 100–101, 150
  - hierarchies, 496
  - Iterable, 196–198
  - java.util.List, 158
  - java.util.Stack, 70
  - Java standard class library for,
    - 487–488
  - lists, 158–159
  - polymorphism, 498, 500–501
  - object-oriented design and, 486–488,
    - 496
  - queues, 125–126
  - searching, 242
  - Serializable, 161
  - sorting, 251
  - stacks, 74–76, 88, 100–101
  - undo operations, 76
  - Internal nodes, 276, 314
  - Is-a relationship, 489–490
  - isEmpty operation, 60, 76, 117, 125, 142,
    - 172, 285, 387, 388
  - binary trees, 285
  - lists, 172
  - maps, 388
  - queues, 125, 142
  - sets, 387
  - stacks, 60, 76, 117
  - Item events, 551
  - Iterable interface,
    - 196–198, 202
  - iterator methods, 161, 196–197, 285,
    - 387
  - iterator operations, 285, 312–313
  - Iterator<E>iterator() method, 196
  - iteratorInOrder method, 285, 312
  - Iterators, 195–211, 219, 285
    - array-based implementation, 205–207
    - binary tree ATD traversal operations, 285
    - boolean hasNext() method, 196
    - collection definition and, 196–198
    - ENext() method, 196
    - element order and, 198
    - exceptions for, 198, 203, 205–207
    - fail-fast implementation, 198, 210
    - hasNext method, 205, 207
    - Iterable interface, 196–198, 202
    - Iterator<E>iterator() method, 196
    - linked-list implementation, 207–209
    - lists and, 198–204
    - loops for, 197, 202–203
    - modification count for, 205
    - object definition and, 196, 210
    - ProgramOfStudy class management,
      - 198–204
    - recursion compared to, 219
    - remove() method, 197, 202–203
    - void remove() method, 196
- J**
- Java Applications Programming
    - Interface (API), 59, 74–76, 88,
    - 100–101, 125–126, 158–159,
    - 386–389, 468, 648–655
  - ArrayList class, 158
  - collections use of, 59, 88, 386–389
  - Deque interface, 100–101
  - hash tables, 648–655
  - HashMap class, 650, 652
  - HashSet class, 650, 651
  - Hashtable class, 648–650

- IdentityHashMap class, 651, 653
  - interface use of, 74–76, 158
  - LinkedHashMap class, 654–655
  - LinkedHashSet class, 654–655
  - LinkedList class, 100–101, 158–159
  - lists, 158–159
  - maps, 386, 388–389
  - object-oriented design and, 468
  - queues, 125–126
  - sets, 386–388
  - Stack class, 74–76, 100
  - stacks, 74–76, 88, 100–101
  - WeakHashMap class, 652, 654
  - Java graphics, 515–536, 538–631
    - borders, 612–616
    - color representation, 517
    - command-line applications, 538
    - components for, 523–527, 539–574, 605–623
    - coordinates, 516
    - dialogue boxes, 605–611
    - drawing shapes, 518–532
    - event adapter class, 604–605
    - events, 538, 543–548, 589–604
    - graphical user interfaces (GUI), 537–631
    - key events, 598–604
    - layout managers, 543, 574–589
    - mnemonics, 616–623
    - monitor resolution, 516
    - mouse events, 589–598
    - picture resolution, 516
    - pixels, 516
    - polygons, 527–532
    - polylines, 527–532
    - tool tips, 616–623
  - Java language, hash functions in, 639
  - Java standard class library, 468
  - Java Virtual Machine (JVM), 32
  - java.lang package, 469
  - java.util package, 468–469
  - java.util.List interface, 158
  - java.util.Stack interface, 70
  - Javadoc comments (`/**` and `*/`), 71–72
  - Javadoc tags (`@`), 72
  - Josephus problem, 170–172, 189
- ## K
- Key events, 598–604
  - Key repetition, 603
  - Keys, maps and, 386
  - keySet() operation, 388
- ## L
- Labels, 543
  - Last in, first out (LIFO) processing, 59–60, 88
  - Late (dynamic) binding, 497
  - Layout managers, 543, 574–589
    - border layout, 574, 579–583
    - box layout, 574, 586–589
    - container government by, 543, 574–589
    - containment hierarchies, 589
    - flow layout, 574, 576–579
    - grid layout, 574, 583–585
    - Java predefined, 574
    - tabbed panes for, 576
  - Leaf, 276, 314
  - Length-dependent method, hash functions, 638
  - Length of a path, 429, 451
  - Level-order traversal, 282, 284–285, 315
  - Line border, 615
  - Linear collections, 55–193, 276
    - lists, 155–193
    - nonlinear collections compared to, 56–57, 276
    - queues, 123–154
    - stacks, 55–122

- Linear probing, 642–643
- Linear search, 244–246, 248, 271
- Linked lists, 95–118, 137–142, 156–158, 186–188, 207–209
  - accessing elements, 96–97
  - deleting nodes, 98
  - dequeue operation, 124–125, 141–142
  - doubly-linked, 99–100, 118
  - elements without links, 99–100
  - enqueue operation, 139–140
  - head/tail element references, 137–142
  - implementation using, 110–117, 137–142
  - inserting nodes, 97–98
  - iterator implementation using, 207–209
  - Java Collections API and, 100–101
  - list implementation using, 186–188
  - lists compared to, 156–158
  - managing, 96–98
  - nodes, 95, 97–100, 118, 137–138
  - peek operation, 117
  - pop operation, 116–117
  - push operation, 114–116
  - queues and, 137–142
  - recursion and, 102–103
  - remove() operations, 187–188
  - stacks and, 100–117
  - traversing a maze using, 101–109
- Linked structures, 93–122, 279–281, 285–313, 322–332, 366–378
  - addElement operation, 323–325, 368–370
  - ADT operations for, 285–289
  - array implementation using, 279–281, 305–313
  - binary search tree implementation using, 322–332
  - binary tree implementation using, 305–313
  - decision trees from, 301–305
  - dynamic, 96
  - expression trees from, 289–300
  - heap implementation using, 366–378
  - implementation and, 110–117, 285–313
  - linked lists, 95–118
  - nodes, 95, 97–100, 110–111, 118
  - object references as, 94–96
  - pointers, 94–95
  - postfix expressions and, 295–300
  - removeMin operation, 330–332, 370–373
  - replacement method, 326–329
  - self-referential objects, 95
  - stacks and, 93–122
  - trees and, 281, 285–313
- LinkedBinaryTree class, 290–313, 322–332
- LinkedHashMap class, 654–655
- LinkedHashSet class, 654–655
- LinkedList class, 100–101, 158, 198–204, 207–209
- LinkedList class, 110–117
- Listeners, 538, 544–548, 594–598
  - GUI-based program design and, 538
  - mouse event interface, 594–598
  - push button interface, 544–548
- Lists, 36, 155–193, 198–204, 442
  - abstract data types (ADT), 172–178
  - add() operation, 158, 173, 183–184
  - addAfter operation, 173, 185
  - adding elements to, 173–178
  - adjacency, 442
  - addToFront operation, 173, 185
  - addToRear operation, 173, 185
  - array-based implementation, 178–186
  - contains operation, 172, 182–183
  - data structure use of, 36
  - exceptions in, 182, 185
  - graph implementation using, 442
  - indexed, 156–158, 170–172

- iterators and, 198–204
- Java API implementation, 158–159
- `java.util.List` interface, 158
- linked-list implementation, 186–188
- linked lists compared to, 156–158
- ordered, 156, 158, 172–178, 183–184
- `ProgramOfStudy` class management,
  - 159–169, 198–204
- `remove()` operations, 158, 172,
  - 180–182, 187–188
- `set()` operation, 158
- `size()` operation, 158, 172
- unordered, 156–157, 159–169,
  - 172–178, 185–186
- Load factor, 402, 404, 636
- Local data, 476
- Logarithmic algorithm, 249
- Logarithmic sort, 249–250
  - loops for, 261
  - merge sort, 250, 262–265
  - partition element, 258–262, 271
  - quick sort, 250, 258–262, 271
- Loops, 48–49, 197, 202–203, 245, 254,
  - 256, 258, 261, 396
  - execution of, 48
  - `for-each` loops, 197, 202–203
  - `for` loop, 258
  - input files read using, 396
  - iterators and, 197, 202–203
  - maps and, 396
  - nested, 48–49
  - `remove()` method and, 197, 202–203
  - searching and, 245
  - sorting and, 254, 256, 258, 261
  - swapping and, 254, 261
  - `while` loop, 197, 245, 261, 396

## M

- `main` method, 102, 159, 396
- Maintainability, 31, 37

- Maps, 36, 385–408
  - collections as, 386, 404
  - data structure use of, 36
  - hashing for implementation of, 401–403
  - implementation, 401–403
  - Java API, 386, 388–389
  - keys, 386
  - loops used for reading input files, 396
  - sets and, 385–408
  - tracking product sales using, 392–396
  - trees for implementation of, 401
  - user management system development,
    - 396–400
- Mathematical use of recursion, 215–216
- Matrices, graph implementation using,
  - 442–450
- Matte border, 615–616
- Maxheap, 358, 379, 382
- Members of a class, 471
- Merge sort, 250, 262–265
- Message dialog boxes, 605, 607
- Message encoding and decoding, 126–130
- Messages, exceptions and, 502
- Method calls, 49–51
- Method declarations, 471
- Methods, 243, 271, 464, 475–476,
  - 477–478, 484–485, 491–492
  - class inheritance and, 491–492
  - compiler and declaration of, 477–478
  - encapsulation and, 475–476
  - local data, 476
  - object-oriented design and, 464, 476,
    - 477–478, 484–485, 491–492
  - overloading, 477–478
  - overriding, 491–492
  - static (class), 243, 271, 484–485
  - searching use of, 243, 271
  - signature, 477
  - support, 475
- MID value, 523



- Mid-square method, hash functions, 637–638
- Minheap, 358, 360, 379, 382
- Minimum spanning trees (MST), 438–440
- Mnemonics, 616–623
- Modification count, 205
- Modifiers, 474–476, 483–485, 490–491
  - private, 474–476
  - protected, 475, 490–491
  - public, 474–476
  - static, 483–485
  - visibility, 474–476
- Monitor resolution, 516
- Motion, mouse events, 589–598
- Mouse events, 589–598
- Multiplication (\*) operator, 65
- Multi-way search trees, 36, 409–426
  - B-, 418–421, 422
  - data structure use of, 36
  - implementation strategies for B-trees, 420–421
  - node characteristics in, 410–420
  - 28-4, 416–417, 423
  - 28-3, 410–416, 423
- N**
- n-ary (general) trees, 277, 315
- Natural ordering, 189
- Neighbors, 428
- Nested loops, 48–49
- Networks (weighted graphs), 431–432, 451
- new operator, 467
- Nodes, 95, 97–100, 110–111, 118, 276–278, 314, 326–329, 340–349, 350, 360–362, 410–420, 428–430, 442–443
  - ancestors, 276–277, 314
  - AVL trees, 340–343
  - B-trees, 418–420
  - balance factor for, 340–343, 350
  - binary search tree implementation, 326–329, 340–349
  - children, 276, 314
  - color of, 343–349
  - deleting, 98
  - descendants, 277, 314
  - doubly linked lists, 99–100
  - dummy, 98
  - graphs, 428–430, 442–443
  - heap implementation, 360–362
  - implementation reference, 110–111
  - inserting, 97–98, 360–361, 411–413
  - internal, 276, 314
  - leaf, 276, 314
  - link lists and, 95, 97–98, 110–111, 118
  - multi-way search trees, 410–420
  - order and classification of trees from, 277–278
  - path length, 277
  - promoted, 326–329
  - red/black trees, 343–349
  - removing, 361–362, 413–416
  - root, 277, 314
  - sentinel, 98, 118
  - separation of stored and linked elements, 99–100
  - siblings, 276, 314
  - tree hierarchy, 276–278, 314
  - 28-4 trees, 416–417
  - 28-3 trees, 410–416
  - vertices as, 428–430, 442–443
- NonComparableElementException
  - class, 323
- Nonlinear collections, 56–57, 276–356
  - linear collections compared to, 56–57, 276
  - trees, 276–356
- NoSuchElementException class, 206–207
- null reference, 478–479

## O

- Object class, 493–494
- Object-oriented design, 463–513
  - classes, 464, 467–469, 470–473, 485–486, 488–505
  - constructors, 476–477
  - encapsulation, 464, 474–476
  - exceptions, 501–505
  - hierarchies, 492–497
  - inheritance, 464, 488–492, 498–499
  - interfaces, 486–488, 496, 498, 500–501
  - methods, 464, 476, 477–478
  - objects, 464–467, 474–476, 478–483
  - polymorphism, 496–501
  - primitive data and, 464
  - reference variables, 466–467, 474–476, 478–483
  - static modifier, 483–485
  - visibility modifiers, 474–476
  - wrapper classes, 485–486
- Objects, 36, 56–58, 61–64, 77, 94–96, 196–198, 210, 464–467, 469–470, 474–476, 478–483, 520–527
  - abstraction of, 57–58, 87, 465–466
  - aliases, 481–482
  - garbage collection, 482–483
  - graphics context interaction, 520–527
  - array of references, 77
  - attributes, 469
  - behavior, 469–470
  - class definition of, 464
  - class hierarchy, 62–63
  - clients, 474
  - collections of, 56–57, 61–64
  - creation of, 466–467
  - data structure use of, 36
  - dot operator `.`, 467
  - dynamic structures, 96
  - elements as, 57
  - encapsulation of, 58, 464, 474–476
  - garbage collection, 482–483
  - generic type, 63–64, 88
  - inheritance, 62–63
  - instantiation, 467
  - iterator interface for, 196–198, 210
  - linked structure references, 94–96
  - nodes, 95
  - object-oriented design use of, 464–467, 469–470, 474–476, 478–483
  - passing as parameters, 483
  - pointers, 94–95
  - polymorphism, 62–63
  - reference variables, 94–96, 466–467, 478–483
  - self-governing, 474–475
  - self-referential, 95
  - stacks, 61–64, 77
  - state of, 470
  - type checking, 62
  - type compatibility, 61–62
  - visibility modifiers, 474–476, 490–491
- `offer` method, 126
- Open addressing methods, 642–645, 647–648
  - deleting elements from, 647–648
  - double hashing, 644–645
  - hashing collisions avoided using, 642–645
  - linear probing, 642–643
  - quadratic probing, 642–644
- Operation, 456

- Ordered lists, 36, 156, 158, 172–178, 183–184, 189, 332–336
  - add() operation, 158, 173, 183–184, 333, 336
  - ADT operations, 172–178, 332–333, 336
  - array-based implementation, 183–184
  - binary search tree implementation of, 332–336
  - data structure use of, 36
  - element characteristics and, 156
- Overriding methods, 491–492
- P**
- Packages, classes and, 468–469
- Panels, 539–543
- Parameter list, 457
- Parent class (superclass), 489
- Partition element, 258–262, 271
- Passing objects as parameters, 483
- Paths, 276–277, 315, 429–430, 441, 451
  - connected graphs, 429–430
  - cycle, 429, 451
  - graphs, 429–430, 441, 451
  - length, 277, 315
  - shortest, determination of, 441
  - tree height and, 277
  - trees, 276–277, 315
- peek operation, 60–61, 76, 85, 117
  - array-based implementation using, 85
  - exceptions for, 61
  - linked-list implementation using, 117
  - stack operations, 60–61, 76
- Perfect hash function, 402, 634–635
- Picture resolution, 516
- Pixels, 516
- Pointers, 94–95
- poll method, 126
- Polygons, 527–532
- Polylines, 527–532
- Polymorphism, 62–63, 88, 496–501
  - class hierarchies and, 497
  - collections and, 62–63, 88
  - inheritance for, 498–499
  - interfaces for, 498, 500–501
  - late (dynamic) binding, 497
  - object-oriented design and, 496–501
  - polymorphic reference, 496–497
- pop operation, 60–61, 76, 83–85, 88, 116–117
  - array-based implementation using, 83–85
  - exceptions for, 61
  - linked-list implementation using, 116–117
  - stack operations, 60–61, 76
- Portability, 32, 37
- Postfix expressions, 64–72, 295–300
- Postorder traversal, 282–284, 315
- Preorder traversal, 282–283, 315
- printsum method, 49–51
- Priority queues, 36, 362–366, 382
  - data structure use of, 36
  - heaps used for, 362–366
  - implementation, 363
- private visibility modifier, 474–476
- Processor speed, algorithms and, 45–47
- Program of study management, 159–169, 198–204
  - iterators for, 198–204
  - lists for, 159–169
  - printing courses, 202
  - removing courses, 202–204
- Program stack, 102, 118
- Programming, recursive, 216–220
- ProgramOfStudy class management, 159–169, 198–204
- Promoted nodes, 326–329, 351

Propagating the exception, 503–504  
 protected visibility modifier, 475, 490–491  
 public visibility modifier, 474–476  
 push operation, 60, 76, 82–83, 88, 114–116  
   array-based implementation using, 82–83  
   linked-list implementation using, 114–116  
   stack interfaces, 60, 76, 88  
 put () operation, 389  
 putAll operation, 389

## Q

Quadratic probing, 642–644  
 QueueADT interface, 135–137  
 Queues, 36, 123–154, 265–269, 362–366  
   abstract data types (ADT), 135–137  
   array-based implementation of, 143–150  
   circular array implementation strategy, 144–150  
   code keys for, 126–130  
   concepts of, 124–125  
   data structure use of, 36  
   deque interface, 150  
   dequeue operation, 124–125, 141–143, 149–150  
   double-ended (deque), 150  
   enqueue operation, 124–125, 139–140, 143, 147–149  
   first in, first out (FIFO) process, 124  
   first operation, 124–125, 142  
   fixed array implementation strategy, 143–144  
   head/tail element references, 137–142  
   interfaces, 135–137  
   isEmpty operation, 125, 142  
   Java API implementation, 125–126

  linked implementation of, 137–142  
   message encoding and decoding with, 126–130  
   priority, 362–366  
   operations on, 124–125  
   radix sort, 265–269  
   size operation, 125, 142  
   toString operation, 125, 142, 150  
   waiting line simulation, 130–135  
 Quick sort, 250, 258–262, 271

## R

Radio buttons, 555–559  
 Radix sort, 265–269, 271  
 Radix transformation method, hash functions, 638  
 Recursion, 102–103, 213–239  
   algorithms, analysis of, 233–234  
   base case, 215–216, 235  
   concept of, 214–216, 235  
   direct, 219–220, 235  
   error in, 216  
   factorial (N!), 215–216  
   indirect, 219–220, 235  
   infinite, 214–215, 235  
   iteration compared to, 219  
   mathematical use of, 215–216  
   programming, 216–220  
   stacks used for, 102–103  
   Towers of Hanoi puzzle, 228–233, 235  
   tracing, 214–215, 218–219  
   traversing a maze, 220–228  
 Red/black tree, 343–349, 351  
   binary tree implementation of, 343–349  
   element removal from, 347–349  
   insertion into, 344–347  
   node color, 343–349

- Reference variables, 94–96, 464, 466–467, 474–476, 478–483, 496–497
    - aliases, 481–482
    - encapsulation and, 474–476
    - garbage collection, 482–483
    - linked structures, 94–96
    - local data, 476
    - null reference, 478–479
    - object-oriented design and, 464, 466–467, 474–476, 478–483, 496–497
    - passing objects as parameters, 483
    - polymorphic, 496–497
    - this reference, 479–480
    - visibility modifiers, 474–476
  - Regular expressions, 661–663
  - Reliability, 29–30, 37
  - remove() operations, 126, 158, 172, 180–182, 187–188, 197, 202–203, 320, 326–332, 387, 389
    - array-based list implementation, 180–182
    - binary search tree implementation, 320, 326–332
    - iterators and, 197, 202–203
    - linked-list implementation, 187–188
    - loops and, 197, 202–203
    - list interfaces, 158, 172
    - maps, 389
    - queue interfaces, 126
    - sets, 387
  - removeAll() operation, 387
  - removeAllOccurrences operation, 320, 329–330
  - removeElement operation, 320, 326–329
  - removeMax operation, 320
  - removeMin operation, 320, 330–332, 358, 361–362, 370–373, 376–378
    - array-based implementation using, 376–378
    - binary search trees, 320, 330–332
    - heaps, 358, 361–362, 370–373, 376–378
    - linked implementation using, 330–332, 370–373
  - Repeating key, 127, 151
  - replacement method, 326–329
  - retainAll() operation, 387
  - Return-type value, 457
  - Reusability, 31–32, 37
  - RGB value, 517
  - Robustness, 30, 37
  - Root, 277, 314
  - Rotation techniques, 337–343
    - AVL tree, 340–343
    - balance factor for, 340–343, 350
    - binary search tree balance, 336–340
    - degenerate tree, 336–337, 350
    - left rotation, 338–339, 341, 351
    - leftright rotation, 339–340, 343, 351
    - right rotation, 337–338, 341, 351
    - rightleft rotation, 339, 341–343, 351
  - Rubberbanding, 598
  - Runtime stack, 102
  - RuntimeException class, 86
- ## S
- Search trees, 36, 281, 319–356, 409–426
    - B-, 418–421, 422
    - binary, 36, 281, 319–356
    - multi-way, 36, 409–426
    - 28-4, 416–457, 423
    - 28-3, 410–416, 423
  - Searching, 241–249, 270–274
    - algorithm comparisons, 248–249
    - binary search, 246–248, 249, 270
    - Comparable interface, 242
    - generic method, 243–244, 270
    - linear search, 244–246, 248, 271
    - search pool, 242, 271

- static (class) methods, 243, 271
- target element, 242, 271
- viable candidates, 246–248, 271
- while loop for, 245
- Selection sort, 250, 252–254, 271
- Self-governing objects, 474–475
- Self-loop, 428, 453
- Self-referential objects, 95
- Sentinel nodes, 98, 118
- Sequential sort, 249–250
  - bubble sort, 250, 256–258, 270
  - insertion sort, 250, 254–256, 270
  - loops for, 254, 256, 258
  - selection sort, 250, 252–254, 271
  - swapping, 254, 257–258
- Serialization, 161, 189
- set () operation, 158
- Sets, 385–408
  - blocked domains using, 389–392
  - collections as, 386, 404
  - hashing for implementation of, 401–403
  - Java API, 386–388
  - maps and, 385–408
  - trees for implementation of, 401
- Siblings, 276, 314, 492
- Signature, 477
- size () operation, 60, 76, 117, 125, 142, 158, 172, 285, 387, 389
  - binary trees, 285
  - lists, 158, 172
  - maps, 389
  - queues, 125, 142
  - sets, 387
  - stacks, 60, 76, 117
- Sliders, 559–564
- Sling, 428
- Software quality, 27–39
  - aspects of, 29
  - client, 28
  - correctness, 29
  - data structures, 33–36
  - efficiency, 32
  - engineering goals, 28
  - failure, 30
  - maintainability, 31
  - portability, 32
  - stakeholder issues, 32–33
  - reliability, 29–30
  - reusability, 31–32
  - robustness, 30
  - software engineering and, 28
  - usability, 30–31
  - user, 28
- Software reuse, 464. *See also*
  - Inheritance
- Sort key, 265
- Sorting, 249–274, 378–380
  - bubble sort, 250, 256–258, 270
  - Comparable interface, 251
  - heap sort, 378–380
  - insertion sort, 250, 254–256, 270
  - logarithmic, 249–250
  - loops for, 254, 256, 258, 261
  - partition element, 258–262, 271
  - merge sort, 250, 262–265
  - process, 249–252
  - quick sort, 250, 258–262, 271
  - radix sort, 265–269, 271
  - selection sort, 250, 252–254, 271
  - sequential, 249–250
  - sort key, 265
  - swapping, 254, 257–258
- Spanning tree, 438, 452
- StackADT interface, 74–76, 80, 110
- Stacks, 35, 55–92, 93–122
  - abstract data type (ADT), 59–61, 74–76
  - activation record, 102–103
  - array-based implementation, 77–86

- Stacks (*Continued*)
    - collections as, 55–92
    - constructors, 80–81, 111
    - data structure use of, 35
    - drop-out, 76
    - exceptions, 72–74, 84
    - implementation, 77–86, 110–117
    - `isEmpty` operation, 60, 76, 117
    - Java API implementation, 74–76
    - last in, first out (LIFO) processing of, 59–60, 88
    - linked-list implementation, 110–117
    - linked structures for, 93–122
    - object-oriented concepts, 61–64
    - `peek` operation, 60–61, 76, 85, 117
    - `pop` operation, 60–61, 76, 83–85, 88, 116–117
    - postfix expressions, 64–72
    - program, 102
    - `push` operation, 60, 76, 82–83, 88, 114–116
    - recursion and, 102–103
    - runtime, 102
    - `size()` operation, 60, 76, 117
    - `toString()` operation, 60, 76, 117
    - traversing a maze using, 101–109
    - undo operations, 76
  - Stakeholder, 32–33, 37
  - Start angle, 519
  - State of an object, 470
  - Static (class) methods, 243, 271, 484–485
  - Static (class) variables, 484
  - `static` modifier, 483–485
  - Stereotypes `<` `>`, 457
  - `sum` method, 217–219
  - `super` reference, 491
  - Support methods, 475
  - Swapping, 254, 257–258
  - Swing package components, 539, 605–623
    - borders, 612–616
    - dialog boxes, 605–611
    - mnemonics, 616–623
    - tool tips, 616–623
  - System development for user
    - management using maps, 396–400
  - System heap, 96
- T**
- Tabbed panes, 576
  - Target element, 242, 271
  - Text area, 608–610
  - Text fields, 548–551
  - `this` reference, 479–480
  - Time complexity, 43, 48–51. *See also* Complexity of algorithms
  - Timers, 569–574
  - Titled border, 615–616
  - `toArray()` operations, 388
  - Tool tips, 616–623
  - `TOP` value, 523
  - Topological order, 430, 452
  - `toString()` operation, 60, 76, 117, 125, 142, 150, 285
    - binary trees, 285
    - queues, 125, 142, 150
    - stacks, 60, 76, 117
  - Towers of Hanoi puzzle, 228–233, 235
  - Tracing recursion process, 214–215, 218–219
  - Tracking product sales using maps, 392–396
  - Traversals, 101–109, 220–228, 282–285, 432–436
    - binary tree ATD operations for, 285
    - breadth-first, 432–433, 436–437, 451
    - depth-first, 432–436, 451
    - graph algorithms, 432–436

- inorder, 282–283, 315
- level-order, 282, 284–285, 315
- maze, 101–109, 220–228
- postorder, 282–284, 315
- preorder, 282–283, 315
- recursion for, 220–228
- stacks for, 101–109
- trees, 282–285
- traverse method, 103, 227–228
- Trees, 36, 275–318, 319–356, 357–380, 401, 409–426, 438–440
  - abstract data types (ADT), 285–289, 320–322
  - array-based implementation, 279–281
  - AVL, 340–343, 350
  - balanced, 277–278, 315, 336–340
  - binary search, 36, 281, 319–356
  - binary, 277, 285–313
  - classification of, 277–278
  - complete, 278, 315, 360–361, 381
  - data structure use of, 36
  - decision, 301–305
  - degenerate, 336–337, 350
  - expression, 289–300
  - full, 278, 315
  - graph algorithms for, 438–440
  - heaps, 357–380
  - height of, 277
  - hierarchy, 276–278
  - implementation strategies, 279–281, 305–313
  - linked implementation, 285–313, 322–332
  - map implementation using, 401
  - minimum spanning (MST), 438–440
  - multi-way search, 36, 409–426
  - n-ary (general), 277, 315
  - nodes, 276–278, 326–329
  - nonlinear structure of, 276–277, 314
  - red/black, 343–349, 351

- rotation of, 337–343
- search, 36
- set implementation using, 401
- spanning, 438, 452
- traversals, 282–285
- unbalanced, 278, 315, 340–343
- try statement, 502–503
- 28-4 trees, 416–417, 423
- 28-3 trees, 410–416, 423
  - inserting elements in, 411–413
  - removing elements from, 413–416
  - 29-node characteristics, 410, 423
  - 28-node characteristics, 410, 423
  - underflow, 414, 423
- Type checking, 62
- Type compatibility, 61–62

## U

- Unbalanced tree, 278, 315, 340–343
  - and() and remove() operations and, 336
  - AVL tree, 340–343
  - binary search tree implementation and, 336–343
  - classification as, 277–278, 315
  - degenerate tree, 336–337, 350
  - left rotation, 338–339, 341, 351
  - leftright rotation, 339–340, 343, 351
  - right rotation, 337–338, 341, 351
  - rightleft rotation, 339, 341–343, 351
- Underflow, 414, 423
- Undirected graphs, 428–429, 443–450
  - adjacency matrices for, 443–450
  - complete, 428
  - connected, 429, 451
  - cycle, 429, 451
  - edges, 428–429
  - implementation of, 443–450
  - path, 429, 452
  - vertices, 428



- Undo operations, 76
  - Unified modeling language (UML),
    - 455–461, 476
    - aggregation, 459–460
    - association, 459–460
    - class diagrams, 456–458, 476
    - class visibility, 457
    - classes using each other, 459–460
    - implementation, 459–460
    - inheritance, 458–459
    - parameter list, 457
    - relationships, 458–460
    - return-type value, 457
    - stereotypes `< >`, 457
  - Unordered lists, 156–157, 159–169,
    - 172–178, 185–186, 189
    - `addAfter` operation, 173, 185
    - `addToFront` operation, 173, 185
    - `addToRear` operation, 173, 185
    - ADT operations, 172–178
    - array-based implementation, 185
    - element placement and, 156–157
    - iterator method, 161
    - `ProgramOfStudy` class example,
      - 159–169
    - `Serializable` interface, 161
  - `UnsupportedOperationException`
    - class, 207
  - Usability, 30–31, 37
  - User, 28
- V**
- `values()` operation, 389
  - Variables, 58, 94–96, 466–467. *See also*
    - Reference variables
      - object references, 94–96, 466–467
      - primitive data types, 58
  - Vertices (nodes), 428–430, 452
  - Viable candidates, 246–248, 271
  - Visibility modifiers, 457, 474–476,
    - 490–491
      - class inheritance and, 490–491
      - encapsulation and, 474–476
      - `private`, 474–476
      - `protected`, 475, 490–491
      - `public`, 474–476
  - `void remove()` method, 196
- W**
- Waiting line simulation, 130–135
  - `WeakHashMap` class, 652, 654
  - Weighted graphs, 431–432
  - `while` loop, 197, 245, 261, 396
  - Wrapper classes, 485–486