

FOURTH  
EDITION

USING  
J2SE 7.0

*DATA*  
**STRUCTURES**  
&  
*OTHER*  
**OBJECTS**

Using Java<sup>TM</sup>

---

***MICHAEL MAIN***

*University of Colorado at Boulder*

**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

Editorial Director: Marcia Horton  
Editor in Chief: Michael Hirsch  
Acquisitions Editor: Tracy Dunkelberger  
Editorial Assistants: Stephanie Sellinger/Emma Snider  
Director of Marketing: Patrice Jones  
Marketing Manager: Yezan Alayan  
Marketing Coordinator: Kathryn Ferranti  
Vice President, Production: Vince O'Brien  
Managing Editor: Jeff Holcomb  
Associate Managing Editor: Robert Engelhardt

Manufacturing Manage: Nick Sklitsis  
Operations Specialist: Lisa McDowell  
Creative Director: Jayne Conte  
Cover Designer: Bruce Kenselaar  
Manager, Rights and Permissions: Karen Sanatar  
Cover Art: © Shutterstock/Ford Prefect  
Media Editor: Dan Sandin  
Printer/Binder: RR Donnelley, Harrisonburg  
Cover Printer: RR Donnelley, Harrisonburg

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

Java is a trademark of the Oracle Corporation, 500 Oracle Parkway, Redwood Shores, CA 94065.

Copyright © 2012, 2006, 2003, 1999 by Pearson Education, Inc., publishing as Addison-Wesley. All rights reserved. Manufactured in the United States of America. This publication is protected by Copyright, and permission should be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission(s) to use material from this work, please submit a written request to Pearson Education, Inc., Permissions Department, One Lake Street, Upper Saddle River, New Jersey 07458, or you may fax your request to 201-236-3290.

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

#### Library of Congress Cataloging-in-Publication Data

Main, M. (Michael)

Data structures & other objects using Java : using J2SE 7.0 / Michael

Main.-- 4th ed.

p. cm.

Includes bibliographic references and index.

ISBN 978-0-13-257624-6 (alk. paper)

1. Java (Computer program language) 2. Data structures (Computer science)

I. Title. II. Title: Data structures and other objects using Java.

QA76.73.J38M33 2012

005.13'3--dc23

2011037942

10 9 8 7 6 5 4 3 2 1



[www.pearsonhighered.com](http://www.pearsonhighered.com)

ISBN-10: 0-13-257624-4  
ISBN-13: 978-0-13257624-6



## Preface

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

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

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

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



variable arity methods that are newly discussed in Chapter 5 of this edition.

- **Enhanced for-loops** (Chapters 3 and 5) were introduced in the previous edition of this textbook. They allow easy iteration through the elements of an array, through a collection of elements that has implemented the `Iterable` interface, or through the elements of an enum type.
- **Autoboxing and auto-unboxing** of primitive values (Chapter 5) continue to be used in this edition to allow the storage and retrieval of primitive data values (such as an `int`) in Java's generic collections (such as the `Vector`).
- **Java's priority queue** (Chapter 7) has been added as a new topic in the fourth edition.
- **The binary search and sort methods from the Java Class Libraries** have a new presentation in Chapters 11 (searching) and 12 (sorting).
- **The important new topic of concurrency** is introduced through a concurrent sort in Chapter 12. The approach uses the `RecursiveAction` and `ForkJoinPool` classes from Java2SE 7, which provides an extremely clean introduction with very few distractions for the first-year student.
- **Covariant return types** allow the return type of an overridden inherited method to be any descendant class of the original return type. We use this idea for clone methods throughout the text, thereby avoiding a typecast with each use of a clone method and increasing type security. The larger use of the technique is postponed until the chapter on inheritance (Chapter 13).
- **Enum types** (Chapter 13) provide a convenient way to define a new data type whose objects may take on a small set of discrete values.

Beyond these new, coherent language features, the fourth edition of this text presents the same foundational data structures material as the earlier editions. In particular, the data structures curriculum emphasizes the ability to specify, design, and analyze data structures independently of any particular language, as well as the ability to implement new data structures and use existing data structures in any modern language. You'll see this approach in the first four chapters as students review Java and then go on to study and implement the most fundamental data structures (bags and sequence classes) using both arrays and linked-list techniques that are easily applied to any high-level language.

Chapter 5 is a bit of a departure, bringing forward several Java-specific techniques that are particularly relevant to data structures: how to build generic collection classes based on Java's new generic types, using Java interfaces and the API classes, and building and using Java iterators. Although much of the subse-

*new material on  
Java interfaces  
and the API  
classes*









## x Preface

### Advanced Projects, Including Concurrency

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

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

### Java Language Versions

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

### Flexibility of Topic Ordering

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

Here are some suggested orderings of the material:

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



## xii *Preface*

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

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

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

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

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

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

### Supplements Via the Internet

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

- Source code
- Errata

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

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

## Acknowledgments

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

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

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

---

Michael Main ([main@colorado.edu](mailto:main@colorado.edu))  
*Boulder, Colorado*

## Chapter List

<b>Chapter 1</b>	<b>THE PHASES OF SOFTWARE DEVELOPMENT</b>	<b>1</b>
<b>Chapter 2</b>	<b>JAVA CLASSES AND INFORMATION HIDING</b>	<b>38</b>
<b>Chapter 3</b>	<b>COLLECTION CLASSES</b>	<b>103</b>
<b>Chapter 4</b>	<b>LINKED LISTS</b>	<b>175</b>
<b>Chapter 5</b>	<b>GENERIC PROGRAMMING</b>	<b>251</b>
<b>Chapter 6</b>	<b>STACKS</b>	<b>315</b>
<b>Chapter 7</b>	<b>QUEUES</b>	<b>360</b>
<b>Chapter 8</b>	<b>RECURSIVE THINKING</b>	<b>409</b>
<b>Chapter 9</b>	<b>TREES</b>	<b>453</b>
<b>Chapter 10</b>	<b>TREE PROJECTS</b>	<b>520</b>
<b>Chapter 11</b>	<b>SEARCHING</b>	<b>567</b>
<b>Chapter 12</b>	<b>SORTING</b>	<b>614</b>
<b>Chapter 13</b>	<b>SOFTWARE REUSE WITH EXTENDED CLASSES</b>	<b>675</b>
<b>Chapter 14</b>	<b>GRAPHS</b>	<b>728</b>
<b>APPENDIXES</b>		<b>775</b>
<b>INDEX</b>		<b>815</b>

## Contents

### CHAPTER 1 THE PHASES OF SOFTWARE DEVELOPMENT 1

1.1	Specification, Design, Implementation 4
	Design Technique: Decomposing the Problem 5
	How to Write a Specification for a Java Method 6
	Pitfall: Throw an Exception to Indicate a Failed Precondition 9
	Temperature Conversion: Implementation 10
	Programming Tip: Use Javadoc to Write Specifications 13
	Programming Tip: Use Final Variables to Improve Clarity 13
	Programming Tip: Make Exception Messages Informative 14
	Programming Tip: Format Output with System.out.printf 14
	Self-Test Exercises for Section 1.1 15
1.2	Running Time Analysis 16
	The Stair-Counting Problem 16
	Big-O Notation 21
	Time Analysis of Java Methods 23
	Worst-Case, Average-Case, and Best-Case Analyses 25
	Self-Test Exercises for Section 1.2 26
1.3	Testing and Debugging 26
	Choosing Test Data 27
	Boundary Values 27
	Fully Exercising Code 28
	Pitfall: Avoid Impulsive Changes 29
	Using a Debugger 29
	Assert Statements 29
	Turning Assert Statements On and Off 30
	Programming Tip: Use a Separate Method for Complex Assertions 32
	Pitfall: Avoid Using Assertions to Check Preconditions 34
	Static Checking Tools 34
	Self-Test Exercises for Section 1.3 34
	Chapter Summary 35
	Solutions to Self-Test Exercises 36

### CHAPTER 2 JAVA CLASSES AND INFORMATION HIDING 38

2.1	Classes and Their Members 40
	Defining a New Class 41
	Instance Variables 41
	Constructors 42
	No-Arguments Constructors 43
	Methods 43
	Accessor Methods 44
	Programming Tip: Four Reasons to Implement Accessor Methods 44
	Pitfall: Division Throws Away the Fractional Part 45
	Programming Tip: Use the Boolean Type for True or False Values 46
	Modification Methods 46
	Pitfall: Potential Arithmetic Overflows 48
	Complete Definition of Throttle.java 48
	Methods May Activate Other Methods 51
	Self-Test Exercises for Section 2.1 51

**xvi** *Contents*

2.2	Using a Class	52
	Creating and Using Objects	52
	A Program with Several Throttle Objects	53
	Null References	54
	NullPointerException	55
	Assignment Statements with Reference Variables	55
	Clones	58
	Testing for Equality	58
	Terminology Controversy: "The Throttle That t Refers To"	59
	Self-Test Exercises for Section 2.2	59
2.3	Packages	60
	Declaring a Package	60
	The Import Statement to Use a Package	63
	The JCL Packages	63
	More about Public, Private, and Package Access	63
	Self-Test Exercises for Section 2.3	65
2.4	Parameters, Equals Methods, and Clones	65
	The Location Class	66
	Static Methods	72
	Parameters That Are Objects	73
	Methods May Access Private Instance Variables of Objects in Their Own Class	74
	The Return Value of a Method May Be an Object	75
	Programming Tip: How to Choose the Names of Methods	76
	Java's Object Type	77
	Using and Implementing an Equals Method	77
	Pitfall: ClassCastException	80
	Every Class Has an Equals Method	80
	Using and Implementing a Clone Method	81
	Pitfall: Older Java Code Requires a Typecast for Clones	81
	Programming Tip: Always Use super.clone for Your Clone Method	85
	Programming Tip: When to Throw a Runtime Exception	85
	A Demonstration Program for the Location Class	85
	What Happens When a Parameter Is Changed Within a Method?	86
	Self-Test Exercises for Section 2.4	89
2.5	The Java Class Libraries	90
	Chapter Summary	92
	Solutions to Self-Test Exercises	93
	Programming Projects	95

**CHAPTER 3 COLLECTION CLASSES 103**

3.1	A Review of Java Arrays	104
	Pitfall: Exceptions That Arise from Arrays	106
	The Length of an Array	106
	Assignment Statements with Arrays	106
	Clones of Arrays	107
	The Arrays Utility Class	108
	Array Parameters	110
	Programming Tip: Enhanced For-Loops for Arrays	111
	Self-Test Exercises for Section 3.1	112
3.2	An ADT for a Bag of Integers	113
	The Bag ADT—Specification	114
	OutOfMemoryError and Other Limitations for Collection Classes	118
	The IntArrayBag Class—Specification	118
	The IntArrayBag Class—Demonstration Program	122
	The IntArrayBag Class—Design	125
	The Invariant of an ADT	126
	The IntArrayBag ADT—Implementation	127
	Programming Tip: Cloning a Class That Contains an Array	136
	The Bag ADT—Putting the Pieces Together	137
	Programming Tip: Document the ADT Invariant in the Implementation File	141
	The Bag ADT—Testing	141
	Pitfall: An Object Can Be an Argument to Its Own Method	142
	The Bag ADT—Analysis	142
	Self-Test Exercises for Section 3.2	144
3.3	Programming Project: The Sequence ADT	145
	The Sequence ADT—Specification	146
	The Sequence ADT—Documentation	150
	The Sequence ADT—Design	150
	The Sequence ADT—Pseudocode for the Implementation	156
	Self-Test Exercises for Section 3.3	158
3.4	Programming Project: The Polynomial	159
	Self-Test Exercises for Section 3.4	162
3.5	The Java HashSet and Iterators	162
	The HashSet Class	162
	Some of the HashSet Members	162
	Iterators	163
	Pitfall: Do Not Access an Iterator's next Item When hasNext Is False	164
	Pitfall: Changing a Container Object Can Invalidate Its Iterator	164
	Invalid Iterators	164
	Self-Test Exercises for Section 3.5	165
	Chapter Summary	165
	Solutions to Self-Test Exercises	166
	Programming Projects	169

**xviii** *Contents***CHAPTER 4 LINKED LISTS 175**

4.1	Fundamentals of Linked Lists	176
	Declaring a Class for Nodes	177
	Head Nodes, Tail Nodes	177
	The Null Reference	178
	Pitfall: NullPointerExceptions with Linked Lists	179
	Self-Test Exercises for Section 4.1	179
4.2	Methods for Manipulating Nodes	179
	Constructor for the Node Class	180
	Getting and Setting the Data and Link of a Node	180
	Public Versus Private Instance Variables	181
	Adding a New Node at the Head of a Linked List	182
	Removing a Node from the Head of a Linked List	183
	Adding a New Node That Is Not at the Head	185
	Removing a Node That Is Not at the Head	188
	Pitfall: NullPointerExceptions with removeNodeAfter	191
	Self-Test Exercises for Section 4.2	191
4.3	Manipulating an Entire Linked List	192
	Computing the Length of a Linked List	193
	Programming Tip: How to Traverse a Linked List	196
	Pitfall: Forgetting to Test the Empty List	197
	Searching for an Element in a Linked List	197
	Finding a Node by Its Position in a Linked List	198
	Copying a Linked List	200
	A Second Copy Method, Returning Both Head and Tail References	204
	Programming Tip: A Method Can Return an Array	205
	Copying Part of a Linked List	206
	Using Linked Lists	207
	Self-Test Exercises for Section 4.3	214
4.4	The Bag ADT with a Linked List	215
	Our Second Bag—Specification	215
	The grab Method	219
	Our Second Bag—Class Declaration	219
	The Second Bag—Implementation	220
	Programming Tip: Cloning a Class That Contains a Linked List	223
	Programming Tip: How to Choose between Different Approaches	225
	The Second Bag—Putting the Pieces Together	229
	Self-Test Exercises for Section 4.4	232
4.5	Programming Project: The Sequence ADT with a Linked List	232
	The Revised Sequence ADT—Design Suggestions	232
	The Revised Sequence ADT—Clone Method	235
	Self-Test Exercises for Section 4.5	238
4.6	Beyond Simple Linked Lists	239
	Arrays Versus Linked Lists and Doubly Linked Lists	239
	Dummy Nodes	240
	Java's List Classes	241
	ListIterators	242
	Making the Decision	243
	Self-Test Exercises for Section 4.6	244
	Chapter Summary	244
	Solutions to Self-Test Exercises	245
	Programming Projects	248

**CHAPTER 5 GENERIC PROGRAMMING 251**

5.1	Java's Object Type and Wrapper Classes	252
	Widening Conversions	253
	Narrowing Conversions	254
	Wrapper Classes	256
	Autoboxing and Auto-Unboxing Conversions	256
	Advantages and Disadvantages of Wrapper Objects	257
	Self-Test Exercises for Section 5.1	257
5.2	Object Methods and Generic Methods	258
	Object Methods	259
	Generic Methods	259
	Pitfall: Generic Method Restrictions	260
	Self-Test Exercises for Section 5.2	261
5.3	Generic Classes	262
	Writing a Generic Class	262
	Using a Generic Class	262
	Pitfall: Generic Class Restrictions	263
	Details for Implementing a Generic Class	263
	Creating an Array to Hold Elements of the Unknown Type	263
	Retrieving E Objects from the Array	264
	Warnings in Generic Code	264
	Programming Tip: Suppressing Unchecked Warnings	265
	Using ArrayBag as the Type of a Parameter or Return Value	266
	Counting the Occurrences of an Object	266
	The Collection Is Really a Collection of References to Objects	267
	Set Unused References to Null	269
	Steps for Converting a Collection Class to a Generic Class	269
	Deep Clones for Collection Classes	271
	Using the Bag of Objects	279
	Details of the Story-Writing Program	282
	Self-Test Exercises for Section 5.3	282
5.4	Generic Nodes	283
	Nodes That Contain Object Data	283
	Pitfall: Misuse of the equals Method	283
	Other Collections That Use Linked Lists	285
	Self-Test Exercises for Section 5.4	285
5.5	Interfaces and Iterators	286
	Interfaces	286
	How to Write a Class That Implements an Interface	287
	Generic Interfaces and the Iterable Interface	287
	How to Write a Generic Class That Implements a Generic Interface	288
	The Lister Class	289
	Pitfall: Don't Change a List While an Iterator Is Being Used	291
	The Comparable Generic Interface	292
	Parameters That Use Interfaces	293
	Using instanceof to Test Whether a Class Implements an Interface	294
	The Cloneable Interface	295
	Self-Test Exercises for Section 5.5	295

**xx Contents**

5.6	A Generic Bag Class That Implements the Iterable Interface (Optional Section)	296
	Programming Tip: Enhanced For-Loops for the Iterable Interface	297
	Implementing a Bag of Objects Using a Linked List and an Iterator	298
	Programming Tip: External Iterators Versus Internal Iterators	298
	Summary of the Four Bag Implementations	299
	Self-Test Exercises for Section 5.6	299
5.7	The Java Collection Interface and Map Interface (Optional Section)	300
	The Collection Interface	300
	The Map Interface and the TreeMap Class	300
	The TreeMap Class	302
	The Word Counting Program	305
	Self-Test Exercises for Section 5.7	306
	Chapter Summary	309
	Solutions to Self-Test Exercises	310
	Programming Projects	312

**CHAPTER 6 STACKS 315**

6.1	Introduction to Stacks	316
	The Stack Class—Specification	317
	We Will Implement a Generic Stack	319
	Programming Example: Reversing a Word	319
	Self-Test Exercises for Section 6.1	320
6.2	Stack Applications	320
	Programming Example: Balanced Parentheses	320
	Programming Tip: The Switch Statement	324
	Evaluating Arithmetic Expressions	325
	Evaluating Arithmetic Expressions—Specification	325
	Evaluating Arithmetic Expressions—Design	325
	Implementation of the Evaluate Method	329
	Evaluating Arithmetic Expressions—Testing and Analysis	333
	Evaluating Arithmetic Expressions—Enhancements	334
	Self-Test Exercises for Section 6.2	334
6.3	Implementations of the Stack ADT	335
	Array Implementation of a Stack	335
	Linked List Implementation of a Stack	341
	Self-Test Exercises for Section 6.3	344
6.4	More Complex Stack Applications	345
	Evaluating Postfix Expressions	345
	Translating Infix to Postfix Notation	348
	Using Precedence Rules in the Infix Expression	350
	Correctness of the Conversion from Infix to Postfix	353
	Self-Test Exercises for Section 6.4	354
	Chapter Summary	354
	Solutions to Self-Test Exercises	355
	Programming Projects	356

**CHAPTER 7 QUEUES 360**

7.1	Introduction to Queues 361
	The Queue Class 362
	Uses for Queues 364
	Self-Test Exercises for Section 7.1 365
7.2	Queue Applications 365
	Java Queues 365
	Programming Example: Palindromes 366
	Programming Example: Car Wash Simulation 369
	Car Wash Simulation—Specification 369
	Car Wash Simulation—Design 369
	Car Wash Simulation—Implementing the Car Wash Classes 374
	Car Wash Simulation—Implementing the Simulation Method 375
	Self-Test Exercises for Section 7.2 375
7.3	Implementations of the Queue Class 383
	Array Implementation of a Queue 383
	Programming Tip: Use Helper Methods to Improve Clarity 386
	Linked List Implementation of a Queue 393
	Pitfall: Forgetting Which End Is Which 398
	Self-Test Exercises for Section 7.3 398
7.4	Deques and Priority Queues (Optional Section) 399
	Double-Ended Queues 399
	Priority Queues 400
	Priority Queue ADT—Specification 400
	Priority Queue Class—An Implementation That Uses an Ordinary Queue 402
	Priority Queue ADT—A Direct Implementation 403
	Java's Priority Queue 403
	Self-Test Exercises for Section 7.4 404
	Chapter Summary 404
	Solutions to Self-Test Exercises 404
	Programming Projects 406

**CHAPTER 8 RECURSIVE THINKING 409**

8.1	Recursive Methods 410
	Tracing Recursive Calls 413
	Programming Example: An Extension of writeVertical 413
	A Closer Look at Recursion 415
	General Form of a Successful Recursive Method 418
	Self-Test Exercises for Section 8.1 419
8.2	Studies of Recursion: Fractals and Mazes 420
	Programming Example: Generating Random Fractals 420
	A Method for Generating Random Fractals—Specification 421
	The Stopping Case for Generating a Random Fractal 426
	Putting the Random Fractal Method in an Applet 426
	Programming Example: Traversing a Maze 429
	Traversing a Maze—Specification 429
	Traversing a Maze—Design 432
	Traversing a Maze—Implementation 433
	The Recursive Pattern of Exhaustive Search with Backtracking 435
	Programming Example: The Teddy Bear Game 437
	Self-Test Exercises for Section 8.2 437

**xxii** *Contents*

8.3	Reasoning about Recursion	439
	How to Ensure That There Is No Infinite Recursion in the General Case	442
	Inductive Reasoning about the Correctness of a Recursive Method	444
	Self-Test Exercises for Section 8.3	445
	Chapter Summary	446
	Solutions to Self-Test Exercises	446
	Programming Projects	448

**CHAPTER 9 TREES 453**

9.1	Introduction to Trees	454
	Binary Trees	454
	Binary Taxonomy Trees	457
	More Than Two Children	458
	Self-Test Exercises for Section 9.1	459
9.2	Tree Representations	459
	Array Representation of Complete Binary Trees	459
	Representing a Binary Tree with a Generic Class for Nodes	462
	Self-Test Exercises for Section 9.2	464
9.3	A Class for Binary Tree Nodes	464
	Programming Example: Animal Guessing	473
	Animal-Guessing Program—Design and Implementation	475
	Animal-Guessing Program—Improvements	481
	Self-Test Exercises for Section 9.3	481
9.4	Tree Traversals	484
	Traversals of Binary Trees	484
	Printing a Tree with an Indentation to Show the Depths	488
	BTNode, IntBTNode, and Other Classes	497
	Self-Test Exercises for Section 9.4	497
9.5	Binary Search Trees	498
	The Binary Search Tree Storage Rules	498
	The Binary Search Tree Bag—Implementation of Some Simple Methods	503
	Counting the Occurrences of an Element in a Binary Search Tree	504
	Adding a New Element to a Binary Search Tree	505
	Removing an Element from a Binary Search Tree	506
	The addAll, addMany, and union Methods	510
	Pitfall: Violating the addTree Precondition	511
	Implementing addAll	512
	Time Analysis and an Internal Iterator	513
	Self-Test Exercises for Section 9.5	513
	Chapter Summary	514
	Solutions to Self-Test Exercises	514
	Programming Projects	517

**CHAPTER 10 TREE PROJECTS 520**

10.1	Hoops	521	The Heap Storage Rules	521
			The Priority Queue Class with Heaps	522
			Adding an Element to a Heap	523
			Removing an Element from a Heap	524
			Self-Test Exercises for Section 10.1	527
10.2	B-Trees	527	The Problem of Unbalanced Trees	527
			The B-Tree Rules	528
			An Example B-Tree	529
			The Set Class with B-Trees	530
			Searching for an Element in a B-Tree	535
			Programming Tip: Private Methods to Simplify Implementations	537
			Adding an Element to a B-Tree	537
			The Loose Addition Operation for a B-Tree	538
			A Private Method to Fix an Excess in a Child	540
			Back to the add Method	541
			Employing Top-Down Design	543
			Removing an Element from a B-Tree	543
			The Loose Removal from a B-Tree	544
			A Private Method to Fix a Shortage in a Child	546
			Removing the Biggest Element from a B-Tree	549
			Programming Tip: Write and Test Small Pieces	549
			External B-Trees	550
			Self-Test Exercises for Section 10.2	551
10.3	Java Support for Trees	552	The DefaultMutableTreeNode from javax.swing.tree	552
			Using the JTree Class to Display a Tree in an Applet	552
			The JApplet Class	552
			Programming Tip: Adding a Component to a JApplet	553
			What the TreeExample Applet Displays	553
			Self-Test Exercises for Section 10.3	553
10.4	Trees, Logs, and Time Analysis	558	Time Analysis for Binary Search Trees	559
			Time Analysis for Heaps	559
			Logarithms	562
			Logarithmic Algorithms	563
			Self-Test Exercises for Section 10.4	563
	Chapter Summary	563		
	Solutions to Self-Test Exercises	564		
	Programming Projects	566		



*Contents xxv*

12.2	Recursive Sorting Algorithms	629
	Divide-and-Conquer Using Recursion	629
	Mergesort	630
	The merge Function	631
	Mergesort—Analysis	638
	Mergesort for Files	640
	Quicksort	640
	The Partition Method	643
	Quicksort—Analysis	646
	Quicksort—Choosing a Good Pivot Element	648
	Self-Test Exercises for Section 12.2	648
12.3	An $O(n \log n)$ Algorithm Using a Heap	649
	Heapsort	649
	Making the Heap	656
	Reheapification Downward	657
	Heapsort—Analysis	658
	Self-Test Exercise for Section 12.3	659
12.4	Java's Sort Methods	660
	Self-Test Exercises for Section 12.4	660
12.5	Concurrent Recursive Sorting	660
	Mergesort with a Threshold	661
	Pitfall: The RecursiveAction Class Did Not Appear Until Java 7	662
	Java's RecursiveAction Class	662
	The Sorter's Constructor	663
	The Sorter's compute Method	664
	Programming Tip: Using InvokeAll	664
	Using a ForkJoinPool to Get Concurrent Recursion Started	665
	Beyond the Simple Sorter Class	668
	Self-Test Exercises for Section 12.5	668
	Chapter Summary	669
	Solutions to Self-Test Exercises	669
	Programming Projects	671

**CHAPTER 13 SOFTWARE REUSE WITH EXTENDED CLASSES 675**

13.1	Extended Classes	676
	How to Declare an Extended Class	678
	The Constructors of an Extended Class	679
	Using an Extended Class	680
	Descendants and Ancestors	681
	Overriding Inherited Methods	682
	Covariant Return Values	684
	Widening Conversions for Extended Classes	684
	Narrowing Conversions for Extended Classes	685
	Self-Test Exercises for Section 13.1	686
13.2	Generic Type Parameters and Inheritance	686
	Self-Test Exercise for Section 13.2	688

**xxvi Contents**

13.3	Simulation of an Ecosystem	688
	Programming Tip: When to Use an Extended Class	689
	Implementing Part of the Organism Object Hierarchy	689
	The Organism Class	689
	The Animal Class: An Extended Class with New Private Instance Variables	693
	How to Provide a New Constructor for an Extended Class	693
	The Other Animal Methods	695
	Self-Test Exercises for the Middle of Section 13.3	696
	The Herbivore Class	700
	The Pond Life Simulation Program	703
	Pond Life—Implementation Details	708
	Using the Pond Model	708
	Self-Test Exercises for the End of Section 13.3	709
13.4	Abstract Classes and a Game Class	710
	Introduction to the AbstractGame Class	710
	Protected Methods	716
	Final Methods	716
	Abstract Methods	717
	An Extended Class to Play Connect Four	718
	The Private Member Variables of the Connect Four Class	718
	Three Connect Four Methods That Deal with the Game's Status	720
	Three Connect Four Methods That Deal with Moves	721
	The clone Method	722
	Writing Your Own Derived Games from the AbstractGame Class	723
	Self-Test Exercises for Section 13.4	724
	Chapter Summary	724
	Further Reading	724
	Solutions to Self-Test Exercises	725
	Programming Projects	727

**CHAPTER 14 GRAPHS 728**

14.1	Graph Definitions	729
	Undirected Graphs	729
	Programming Example: Undirected State Graphs	730
	Directed Graphs	733
	More Graph Terminology	734
	Airline Routes Example	735
	Self-Test Exercises for Section 14.1	736
14.2	Graph Implementations	736
	Representing Graphs with an Adjacency Matrix	736
	Using a Two-Dimensional Array to Store an Adjacency Matrix	737
	Representing Graphs with Edge Lists	738
	Representing Graphs with Edge Sets	739
	Which Representation Is Best?	739
	Programming Example: Labeled Graph ADT	740
	The Graph Constructor and size Method	741
	Methods for Manipulating Edges	742
	Methods for Manipulating Vertex Labels	743
	Labeled Graph ADT—Implementation	743
	Self-Test Exercises for Section 14.2	749

*Contents xxvii*

14.3	Graph Traversals	749
	Depth-First Search	750
	Breadth-First Search	754
	Depth-First Search—Implementation	756
	Breadth-First Search—Implementation	758
	Self-Test Exercises for Section 14.3	759
14.4	Path Algorithms	759
	Determining Whether a Path Exists	759
	Graphs with Weighted Edges	759
	Shortest-Distance Algorithm	760
	Shortest-Path Algorithm	770
	Self-Test Exercises for Section 14.4	771
	Chapter Summary	771
	Solutions to Self-Test Exercises	772
	Programming Projects	773

**APPENDIXES**

A.	JAVA'S PRIMITIVE TYPES AND ARITHMETIC OVERFLOW	775
B.	JAVA INPUT AND OUTPUT	778
C.	THROWING AND CATCHING JAVA EXCEPTIONS	782
D.	ARRAYLIST, VECTOR, HASHTABLE, AND HASHMAP CLASSES	787
E.	A CLASS FOR NODES IN A LINKED LIST	791
F.	A CLASS FOR A BAG OF OBJECTS	799
G.	FURTHER BIG-O NOTATION	805
H.	JAVADOC	807
I.	APPLETS FOR INTERACTIVE TESTING	814

**INDEX 815**

*This page intentionally left blank*



# CHAPTER 1

# The Phases of Software Development

---

## LEARNING OBJECTIVES

When you complete Chapter 1, you will be able to ...

- use Javadoc to write a method's complete specification, including a precondition/postcondition contract.
- recognize quadratic, linear, and logarithmic runtime behavior in simple algorithms, and write big-O expressions to describe this behavior.
- create and recognize test data that is appropriate for a problem, including testing boundary conditions and fully exercising code.

---

## CHAPTER CONTENTS

- 1.1 Specification, Design, Implementation
- 1.2 Running Time Analysis
- 1.3 Testing and Debugging
  - Chapter Summary
  - Solutions to Self-Test Exercises

## CHAPTER

## 1

# The Phases of Software Development

*Chapter the first which explains how, why, when, and where there was ever any problem in the first place*

NOEL Langley

*The Land of Green Ginger*

This chapter illustrates the phases of software development. These phases occur for all software, including the small programs you'll see in this first chapter. In subsequent chapters, you'll go beyond these small programs, applying the phases of software development to organized collections of data. These organized collections of data are called **data structures**, and the main topics of this book revolve around proven techniques for representing and manipulating such data structures.

## Data Structure

A **data structure** is a collection of data, organized so that items can be stored and retrieved by some fixed techniques. For example, a Java array is a simple data structure that allows individual items to be stored and retrieved based on an index ([0], [1], [2], ...) that is assigned to each item.

Throughout this book, you will be presented with many forms of data structures with organizations that are motivated by considerations such as ease of use or speed of inserting and removing items from the structure.

Years from now, you may be a software engineer writing large systems in a specialized area, perhaps artificial intelligence or computational biology. Such futuristic applications will be exciting and stimulating, and within your work you will still see the data structures that you learn and practice now. You will still be following the same phases of software development that you learned when designing and implementing your first programs. Here is a typical list of the software development phases:

## The Phases of Software Development

- Specification of the task
- Design of a solution
- Implementation (coding) of the solution
- Analysis of the solution
- Testing and debugging
- Maintenance and evolution of the system
- Obsolescence

You don't need to memorize this list; throughout the book, your practice of these phases will achieve far better familiarity than mere memorization. Also, memorizing an "official list" is misleading because it suggests that there is a single sequence of discrete steps that always occur one after another. In practice, the phases blur into each other. For instance, the analysis of a solution's efficiency may occur hand in hand with the design, before any coding, or low-level design decisions may be postponed until the implementation phase. Also, the phases might not occur one after another. Typically, there is back and forth travel between the phases.

*the phases blur  
into each other*

Most work in software development does not depend on any particular programming language. Specification, design, and analysis can all be carried out with few or no ties to a particular programming language. Nevertheless, when we get down to implementation details, we do need to decide on one particular programming language. The language we use in this book is **Java™**, and the particular version we use is Java Standard Edition 7.

## What You Should Know About Java Before Starting This Text

The Java language was conceived by a group of programmers at Sun Microsystems™ in 1991. The group, led by James Gosling, had an initial design called Oak that was motivated by a desire for a single language in which programs could be developed and easily moved from one machine to another. Over the next four years, many Sun programmers contributed to the project, and Gosling's Oak evolved into the Java language. Java's goal of easily moving programs between machines was met by introducing an intermediate form called **byte codes**. To run a Java program, the program is first translated into byte codes; the byte codes are then given to a machine that runs a controlling program called the **Java Runtime Environment (JRE)**. Because the JRE is freely available for a wide variety of machines, Java programs can be moved from one machine to another. Because the JRE controls all Java programs, there is an added level of security that comes from avoiding potential problems from running unknown programs.

*the origin of  
Java*

In addition to the original goals of program transportability and security, the designers of Java also incorporated ideas from other modern programming languages. Most notably, Java supports **object-oriented programming (OOP)** in a manner that was partly taken from the C++ programming language. OOP is a programming approach that encourages strategies of information hiding and component reuse. In this book, you will be introduced to these important OOP principles to use in your designs and implementations.

*this book gives  
an introduction  
to OOP  
principles for  
information  
hiding and  
component  
reuse*

All of the programs in this book have been developed and tested with Sun's **Java Development Kit (JDK 7)**, but many other Java programming environments may be successfully used with this text. You should be comfortable writing, compiling, and running short Java application programs in your environment. You should know how to use the Java primitive types (the number types, char, and boolean), and you should be able to use arrays.

The rest of this chapter will prepare you to tackle the topic of data structures in Java. Section 1.1 focuses on a technique for specifying program behavior, and

## 4 Chapter 1 / The Phases of Software Development

you'll also see some hints about design and implementation. Section 1.2 illustrates a particular kind of analysis: the running time analysis of a program. Section 1.3 provides some techniques for testing and debugging Java programs.

you should  
already know  
how to write,  
compile, and run  
short Java  
programs in  
some  
programming  
environment

### 1.1 SPECIFICATION, DESIGN, IMPLEMENTATION

*One begins with a list of difficult design decisions which are likely to change. Each module is then designed to hide such a decision from the others.*

D. L. PARNAS

"On the Criteria to Be Used in Decomposing Systems into Modules"

As an example of software development in action, let's examine the specification, design, and implementation for a particular problem. The **specification** is a precise description of the problem; the **design** phase consists of formulating the steps (or **algorithm**) to solve the problem; and the **implementation** is the actual Java code to carry out the design.

The problem we have in mind is to display a table for converting Celsius temperatures to Fahrenheit, similar to the table shown here. For a small problem, a sample of the desired output is a reasonable specification. Such a sample is *precise*, leaving no doubt as to what the program must accomplish. The next step is to design a solution.

An **algorithm** is a procedure or sequence of directions for solving a problem. For example, an algorithm for the temperature problem will tell how to produce the conversion table. An algorithm can be

expressed in many different ways, such as in English, in a mixture of English and mathematical notation, or in a mixture of English with a programming language. This mixture of English and a programming language is called **pseudocode**. Using pseudocode allows us to avoid programming language details that may obscure a simple solution, but at the same time we can use Java code (or another language) when the code is clear. Keep in mind that the reason for pseudocode is to improve *clarity*.

TEMPERATURE CONVERSION	
Celsius	Fahrenheit
-50.00C	The equivalent
-40.00C	Fahrenheit
-30.00C	temperatures will be
-20.00C	computed and
-10.00C	displayed on this side
0.00C	of the table.
10.00C	
20.00C	
30.00C	
40.00C	
50.00C	

#### Algorithm

An **algorithm** is a procedure or sequence of instructions for solving a problem. Any algorithm may be expressed in many different ways: in English, in a particular programming language, or (most commonly) in a mixture of English and programming called **pseudocode**.

We'll use pseudocode to design a solution for the temperature problem, and we'll also use the important design technique of decomposing the problem, which we'll discuss now.

### Design Technique: Decomposing the Problem

A good technique for designing an algorithm is to break down the problem at hand into a few subtasks, then decompose each subtask into smaller subtasks, then replace the smaller subtasks with even smaller subtasks, and so forth. Eventually the subtasks become so small that they are trivial to implement in Java or whatever language you are using. When the algorithm is translated into Java code, each subtask is implemented as a separate Java method. In other programming languages, methods are called "functions" or "procedures," but it all boils down to the same thing: The large problem is decomposed into subtasks, and subtasks are implemented as separate pieces of your program.

For example, the temperature problem has at least two good subtasks: converting a temperature from Celsius degrees to Fahrenheit and printing a number with a specified accuracy (such as rounding to the nearest hundredth). Using these two subproblems, the first draft of our pseudocode might look like this:

1. Display the labels at the top of the table.
2. For each line in the table (using variables `celsius` and `fahrenheit`):
  - 2a. Set `celsius` equal to the next Celsius temperature of the table.
  - 2b. `fahrenheit` = the `celsius` temperature converted to Fahrenheit.
  - 2c. Print one line of the output table with each temperature rounded to the nearest hundredth and labeled (by the letter C or F).
3. Print the line of dashes at the bottom of the table.

The underlined steps (2b and 2c) are the major subtasks. But aren't there other ways to decompose the problem into subtasks? What are the aspects of a good decomposition? One primary guideline is that the subtasks should help you produce short pseudocode—no more than a page of succinct description to solve the entire problem and ideally much less than a page. In your first designs, you can also keep in mind two considerations for selecting good subtasks: the potential for code reuse and the possibility of future changes to the program. Let's see how our subtasks embody these considerations.

Step 2c is a form of a common task: printing some information with a specified format. This task is so common that newer versions of Java have included a method, `System.out.printf`, that can be used by any program that produces formatted output. We'll discuss and use this function when we implement Step 2c.

The `printf` method is an example of **code reuse**, in which a single method can be used by many programs for similar tasks. In addition to Java's many packages of reusable methods, programmers often produce packages of their own Java methods that are intended to be reused over and over with many different application programs.

#### Key Design Concept

Break down a task into a few subtasks; then decompose each subtask into smaller subtasks.

*what makes a good decomposition?*

*the printf method*

*code reuse*

## 6 Chapter 1 / The Phases of Software Development

### easily modified code

Decomposing problems also produces a good final program in the sense that the program is easy to understand, and subsequent maintenance and modifications are relatively easy. For example, our temperature program might later be modified to convert to Kelvin degrees instead of Fahrenheit. Since the conversion task is performed by a separate Java method, most of the modification will be confined to this one method. Easily modified code is vital since real-world studies show that a large proportion of programmers' time is spent maintaining and modifying existing programs.

For a problem decomposition to produce easily modified code, the Java methods you write need to be genuinely separated from one another. An analogy can help explain the notion of "genuinely separated." Suppose you are moving a bag of gold coins to a safe hiding place. If the bag is too heavy to carry, you might divide the coins into three smaller bags and carry the bags one by one. Unless you are a character in a comedy, you would not try to carry all three bags at once. That would defeat the purpose of dividing the coins into three groups. This strategy works only if you carry the bags one at a time. Something similar happens in problem decomposition. If you divide your programming task into three subtasks and solve these subtasks by writing three Java methods, you have traded one hard problem for three easier problems. Your total job has become easier—provided that you design the methods separately. When you are working on one method, you should not worry about how the other methods perform their jobs. But the methods do interact. So when you are designing one method, you need to know something about what the other methods do. The trick is to know *only as much as you need but no more*. This is called **information hiding**. One technique for information hiding involves specifying your methods' behavior using *preconditions* and *postconditions*, which we discuss next.

### How to Write a Specification for a Java Method

When you implement a method in Java, you give complete instructions for how the method performs its computation. However, when you are *using a method* in your pseudocode or writing other Java code, you only need to think about *what the method does*. You need not think about *how the method* does its work. For example, suppose you are writing the temperature-conversion program and you are told that the following method is available for you to use:

```
// Convert a Celsius temperature c to Fahrenheit degrees.  
public static double celsiusToFahrenheit(double c)
```

### signature of a method

This information about the method is called its *signature*. A **signature** includes the method name (`celsiusToFahrenheit`), its parameter list (`double c`), its return type (a `double` number), and any modifiers (`public` and `static`).

In your program, you might have a `double` variable called `celsius` that contains a Celsius temperature. Knowing this description, you can confidently write the following statement to convert the temperature to Fahrenheit degrees, storing the result in a `double` variable called `fahrenheit`:

```
fahrenheit = celsiusToFahrenheit(celsius);
```

When you use the `celsiusToFahrenheit` method, you do not need to know the details of how the method carries out its work. You need to know *what* the method does, but you do not need to know *how* the task is accomplished.

When we pretend that we do not know how a method is implemented, we are using a form of information hiding called **procedural abstraction**. This simplifies your reasoning by abstracting away irrelevant details (that is, by hiding them). When programming in Java, it might make more sense to call it “method abstraction” since you are abstracting away irrelevant details about how a method works. However, computer scientists use the term *procedure* for any sequence of instructions, so they also use the term *procedural abstraction*. Procedural abstraction can be a powerful tool. It simplifies your reasoning by allowing you to consider methods one at a time rather than all together.

*procedural abstraction*

To make procedural abstraction work for us, we need some techniques for documenting what a method does without indicating how the method works. We could just write a short comment as we did for `celsiusToFahrenheit`. However, the short comment is a bit incomplete; for instance, the comment doesn’t indicate what happens if the parameter `c` is smaller than the lowest Celsius temperature ( $-273.15^{\circ}\text{C}$ , also called **absolute zero**). For better completeness and consistency, we will follow a fixed format that is guaranteed to provide the same kind of information about any method you may write. The format has five parts, which are illustrated below and on the next page for the `celsiusToFahrenheit` method.

◆ **celsiusToFahrenheit**

```
public static double celsiusToFahrenheit(double c)
```

Convert a temperature from Celsius degrees to Fahrenheit degrees.

**Parameters:**

`c` – a temperature in Celsius degrees

**Precondition:**

`c >= -273.15`.

**Returns:**

the temperature `c` converted to Fahrenheit degrees

**Throws: IllegalArgumentException**

Indicates that `c` is less than the smallest Celsius temperature ( $-273.15$ ).

This documentation is called the method’s **specification**. Let’s look at the five parts of this specification.

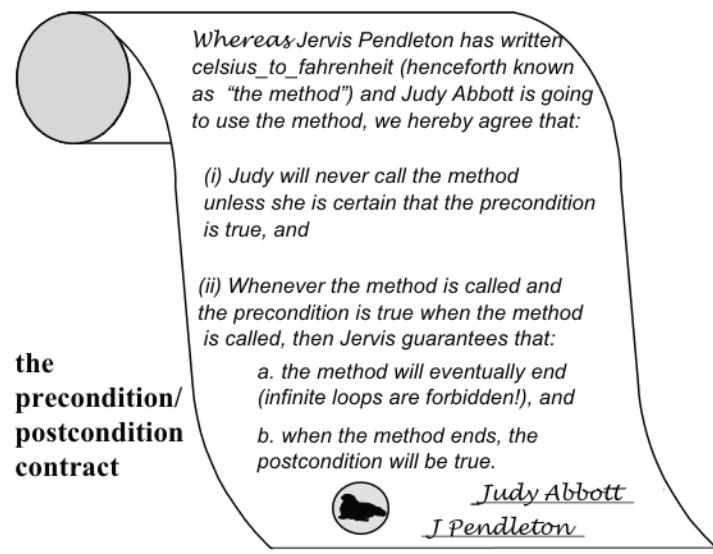
**1. Short Introduction.** The specification’s first few lines are a brief introduction. The introduction includes the method’s name, the complete heading (`public static double celsiusToFahrenheit(double c)`), and a short description of the action that the method performs.

**2. Parameter Description.** The specification’s second part is a list of the method’s parameters. We have one parameter, `c`, which is a temperature in Celsius degrees.



Jervis is one of her programmers, who writes various functions for Judy to use in large programs. If Judy and Jervis were lawyers, the contract might look like the scroll shown in the margin. As a programmer, the contract tells them precisely what the function does. It states that if Judy makes sure that the precondition is met when the function is called, then Jervis ensures that the function returns with the postcondition satisfied.

Before long, we will provide both the specification and the implementation of the `celsiusToFahrenheit` method. But keep in mind that we need only the specification in order to know how to use the method.



**5. The Throws List.** It is always the responsibility of the programmer who *uses* a method to ensure that the precondition is valid. Calling a method without ensuring a valid precondition is a programming error. Once the precondition fails, the method's behavior is unpredictable—the method *could* do anything at all. Nonetheless, the person who writes a method should make every effort to avoid the more unpleasant behaviors, even if the method is called incorrectly. As part of this effort, the first action of a method is often to check that its precondition has been satisfied. If the precondition fails, then the method *throws an exception*. You may have used exceptions in your previous programming, or maybe not. In either case, the next programming tip describes the exact meaning of *throwing an exception* to indicate that a precondition has failed.

## PROGRAMMING TIP

### THROW AN EXCEPTION TO INDICATE A FAILED PRECONDITION

It is a programming error to call a method when the precondition is invalid. For example, `celsiusToFahrenheit` should not be called with an argument that is below -273.15. Despite this warning, some programmer may try `celsiusToFahrenheit(-9001)` or `celsiusToFahrenheit(-273.16)`. In such a case, our `celsiusToFahrenheit` method will detect that the precondition has been violated, immediately halt its own work, and pass a "message" to the calling program to indicate that an illegal argument has occurred. Such messages for programming errors are called **exceptions**. The act of halting your own work and passing a message to the calling program is known as **throwing an exception**.

## 10 Chapter 1 / The Phases of Software Development

how to throw an exception

The Java syntax for throwing an exception is simple. You begin with the keyword `throw` and follow this pattern:

```
throw new                  ("                ");
```

This is the type of exception we are throwing. To begin with, all of our exceptions will be the type `IllegalArgumentException`, which is provided as part of the Java language. This type of exception tells a programmer that one of the method's arguments violated a precondition.

This is an error message that will be passed as part of the exception. The message should describe the error in a way that will help the programmer fix the programming error.

what happens when an exception is thrown?

When an exception is thrown in a method, the method stops its computation. A new "exception object" is created, incorporating the indicated error message. The exception, along with its error message, is passed up to the method or program that made the illegal call in the first place. At that point, where the illegal call was made, there is a Java mechanism to "catch" the exception, try to fix the error, and continue with the program's computation. You can read about exception catching in Appendix C. However, exceptions that arise from precondition violations should never be caught because they indicate programming errors that must be fixed. When an exception is not caught, the program halts, printing the error message along with a list of the method calls that led to the exception. This error message can help the programmer fix the programming error.

Now you know the meaning of the specification's "throws list." It is a list of all the exceptions that the method can throw, along with a description of what causes each exception. Certain kinds of exceptions must also be listed in a method's implementation, after the parameter list, but an `IllegalArgumentException` is listed only in the method's specification.

### Temperature Conversion: Implementation

Our specification and design are now in place. The subtasks are small enough to implement, though during the implementation you may need to finish small design tasks such as finding the conversion formula. In particular, we can now implement the temperature program as a Java application program with two methods:

- a `main` method that follows the pseudocode from page 5. This `main` method prints the temperature conversion table using the method `celsiusToFahrenheit` to carry out some of its work.
- the `celsiusToFahrenheit` method.

The Java application program with these two methods appears in Figure 1.1. The program produces the output from our initial specification on page 4.

A few features of the implementation might be new to you, so we will discuss these in some programming tips after the figure.

**FIGURE 1.1** Specification and Implementation for the Temperature Conversion Application

### Class TemperatureConversion

#### ❖ **public class TemperatureConversion**

The TemperatureConversion Java application prints a table converting Celsius to Fahrenheit degrees.

### Specification

#### ◆ **main**

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

The **main** method prints a Celsius-to-Fahrenheit conversion table. The **String** arguments (**args**) are not used in this implementation. The bounds of the table range from  $-50^{\circ}\text{C}$  to  $+50^{\circ}\text{C}$  in 10-degree increments.

#### ◆ **celsiusToFahrenheit**

```
public static double celsiusToFahrenheit(double c)
```

Convert a temperature from Celsius degrees to Fahrenheit degrees.

**Parameters:**

**c** – a temperature in Celsius degrees

**Precondition:**

**c**  $\geq -273.15$ .

**Returns:**

the temperature **c** converted to Fahrenheit degrees

**Throws: IllegalArgumentException**

Indicates that **c** is less than the smallest Celsius temperature ( $-273.15$ ).

*These specifications were automatically produced by the Javadoc tool. Appendix H describes how to use Javadoc to produce similar information.*

(continued)

**12 Chapter 1 / The Phases of Software Development**

(FIGURE 1.1 continued)

**Java Application Program**

```
// File: TemperatureConversion.java from
// www.cs.colorado.edu/~main/applications/
// A Java application to print a
// temperature conversion table.
// Additional Javadoc information is available on page 11 or at
// http://www.cs.colorado.edu/~main/docs/TemperatureConversion.html.

public class TemperatureConversion
{
    public static void main(String[ ] args)
    {
        // Declare values that control the table's bounds.
        final double TABLE_BEGIN = -50.0; // The table's first Celsius temperature
        final double TABLE_END    = 50.0; // The table's final Celsius temperature
        final double TABLE_STEP   = 10.0; // Increment between temperatures in table

        double celsius;                  // A Celsius temperature
        double fahrenheit;               // The equivalent Fahrenheit temperature

        System.out.println("TEMPERATURE CONVERSION");
        System.out.println("-----");
        System.out.println("Celsius      Fahrenheit");
        for (celsius = TABLE_BEGIN; celsius <= TABLE_END; celsius += TABLE_STEP)
        { // The for-loop has set celsius equal to the next Celsius temperature of the table.
            fahrenheit = celsiusToFahrenheit(celsius);
            System.out.printf("%6.2fC", celsius);
            System.out.printf("%14.2fF\n", fahrenheit);
        }
        System.out.println("-----");
    }

    public static double celsiusToFahrenheit(double c)
    {
        final double MINIMUM_CELSIUS = -273.15;
        if (c < MINIMUM_CELSIUS)
            throw new IllegalArgumentException("Argument " + c + " is too small.");
        return (9.0/5.0) * c + 32;
    }
}
```

The actual implementation includes Javadoc comments that are omitted here but are listed in Appendix H. The comments allow Javadoc to produce nicely formatted information automatically. See Appendix H to read about using Javadoc for your programs.

**PROGRAMMING TIP****USE JAVADOC TO WRITE SPECIFICATIONS**

The specification at the start of Figure 1.1 was produced in an interesting way. Most of it was automatically produced from the Java code by a tool called **Javadoc**. With a web browser, you can access this specification over the Internet at: <http://www.cs.colorado.edu/~main/docs/TemperatureConversion.html>.

To use Javadoc, the .java file that you write needs special **Javadoc comments**. These necessary comments are not listed in Figure 1.1, but Appendix H is a short manual on how to write these comments. Before you go on to Chapter 2, you should read through the appendix and be prepared to produce Javadoc comments for your own programs.

*the Javadoc tool automatically produces nicely formatted information about a class*

**PROGRAMMING TIP****USE FINAL VARIABLES TO IMPROVE CLARITY**

The method implementation in Figure 1.1 has a local variable declared this way:

```
final double MINIMUM_CELSIUS = -273.15;
```

This is a declaration of a double variable called MINIMUM\_CELSIUS, which is given an initial value of -273.15. The keyword **final**, appearing before the declaration, makes MINIMUM\_CELSIUS more than just an ordinary declaration. It is a **final variable**, which means that its value will never be changed while the program is running. A common programming style is to use all capital letters for the names of final variables. This makes it easy to determine which variables are final and which may have their values changed.

There are several advantages to defining MINIMUM\_CELSIUS as a final variable rather than using the constant -273.15 directly in the program. Using the name MINIMUM\_CELSIUS makes the comparison ( $c < \text{MINIMUM\_CELSIUS}$ ) easy to understand; it's clear that we are testing whether  $c$  is below the minimum valid Celsius temperature. If we used the direct comparison ( $c < -273.15$ ) instead, a person reading our program would have to stop to remember that -273.15 is "absolute zero," the smallest Celsius temperature.

To increase clarity, some programmers declare all constants as final variables. This is a good rule, particularly if the same constant appears in several different places in the program or if you plan to later compile the program with a different value for the constant. However, well-known formulas may be more easily recognized in their original form. For example, the conversion from Celsius to Fahrenheit is recognizable as  $F = \frac{9}{5}C + 32$ . Thus, Figure 1.1 uses this statement:

```
return (9.0/5.0) * c + 32;
```

This return statement is clearer and less error prone than a version that uses final variables for the constants  $\frac{9}{5}$  and 32.

Advice: Use final variables instead of constants. Make exceptions when constants are clearer or less error prone. When in doubt, write both solutions and interview several colleagues to decide which is clearer.

## 14 Chapter 1 / The Phases of Software Development

### PROGRAMMING TIP

#### MAKE EXCEPTION MESSAGES INFORMATIVE

If the `celsiusToFahrenheit` method detects that its parameter, `c`, is too small, then it throws an `IllegalArgumentException`. The message in the exception is:

`"Argument " + c + " is too small."`

The parameter, `c`, is part of the message. If a programmer attempts to call `celsiusToFahrenheit(-300)`, the message will be "Argument -300 is too small."

### PROGRAMMING TIP

#### FORMAT OUTPUT WITH `SYSTEM.OUT.PRINTF`

Our pseudocode for the temperature problem includes a step to print the Celsius and Fahrenheit temperatures rounded to a specified accuracy. Java has a method, `System.out.printf`, that we can use to print formatted output such as rounded numbers. The method, which is based on a similar function in the C language, always has a "format string" as its first argument. The format string tells how subsequent arguments should be printed to `System.out` (which is usually the computer monitor). For example, you could create an appropriate format string and print two numbers called `age` and `height`:

```
System.out.println(format string, age, height);
```

Two things can appear in a format string:

1. Most ordinary characters are just printed as they appear in the format string. For example, if the character `F` appears in the format string, then this will usually just print an `F`. Some sequences of characters are special, such as `\n` (which moves down to start the next new line). The complete list of special characters is in Appendix B.
2. Parts of the format string that begin with the `%` character are called **format specifiers**. Each format specifier indicates how the next item should be printed. Some of the common format specifiers are shown here:

Format specifier	Causes the next item to be printed as ...
<code>%d</code>	... a decimal (base 10) whole number
<code>%f</code>	... a floating point number with six digits after the decimal point
<code>%g</code>	... a floating point number using scientific notation for large exponents
<code>%s</code>	... a string

The format specifiers have several additional options that are described in Appendix B. For our program, we used a format specifier of the form `%6.2f`,

which means that we want to print a floating point number using six total spaces, two of which are after the decimal point. Within these six spaces, the number will be rounded to two decimal points and right justified. For example, the number 42.129 will print these six characters (the first of which is a blank space):

	4	2	.	1	3
--	---	---	---	---	---

In the format specifier `%6.2f`, the number 6 is the **field width**, and the number 2 is the **precision**.

As an example, `System.out.println("%14.2f\n", fahrenheit)` prints the value of the `fahrenheit` variable using a field width of 14 and a precision of 2. After printing this number, the character F and a new line will appear.

### Self-Test Exercises for Section 1.1

Each section of this book finishes with a few self-test exercises. Answers to these exercises are given at the end of each chapter. The first exercise refers to a method that Judy has written for *you* to use. Here is the specification:

◆ **dateCheck**

```
public static int dateCheck(int year, int month, int day)  
Compute how long until a given date will occur.
```

**Parameters:**

year – the year for a given date  
month – the month for a given date (using 1 for Jan, 2 for Feb, etc.)  
day – the day of the month for the given date

**Precondition:**

The three arguments are a legal year, month, and day of the month in the years 1900 to 2099.

**Returns:**

If the given date has been reached on or before today, the return value is zero. Otherwise, the return value is the number of days until the given date returns.

**Throws: `IllegalArgumentException`**

Indicates the arguments are not a legal date in the years 1900 to 2099.

1. Suppose you call the method `dateCheck(2013, 7, 29)`. What is the return value if today is July 22, 2013? What if today is July 30, 2013? What about February 1, 2014?
2. Can you use `dateCheck` even if you don't know how it is implemented?
3. Suppose that `boddle` is a `double` variable. Write two statements that will print `boddle` to `System.out` in the following format: \$ 42,567.19  
The whole dollars part of the output can contain up to nine characters (including the comma), so this example has three spaces before the six characters of 42,567. The fractional part has just two digits. Appendix B describes the format string that you'll need in order to have the number include a comma separator between the thousands and hundreds.

## 16 Chapter 1 / The Phases of Software Development

4. Consider a method with this heading:

```
public static void printSqrt(double x)
```

The method prints the square root of  $x$  to the standard output. Write a reasonable specification for this method and compare your answer to the solution at the end of the chapter. The specification should forbid a negative argument.

5. Consider a method with this heading:

```
public static double sphereVolume(double radius)
```

The method computes and returns the volume of a sphere with the given radius. Write a reasonable specification for this method and compare your answer to the solution at the end of the chapter. The specification should require a positive radius.

6. Write an if-statement to throw an `IllegalArgumentException` when  $x$  is less than zero. (Assume that  $x$  is a `double` variable.)
7. When can a `final` variable be used?
8. How was the specification produced at the start of Figure 1.1?
9. What are the components of a Java signature?
10. Write a Java statement that will print two variables called `age` and `height`. The `age` should be printed as a whole number using 10 output spaces; the `height` should be printed using 12 output spaces and three decimal digits. Label each part of the output as “Age” and “Height”. Use Appendix B if necessary.

## 1.2 RUNNING TIME ANALYSIS

**Time analysis** consists of reasoning about an algorithm’s speed. *Does the algorithm work fast enough for my needs? How much longer does the algorithm take when the input gets larger? Which of several different algorithms is fastest?* These questions can be asked at any stage of software development. Some analysis of an algorithm is useful before any implementation is done to avoid the wasted work of implementing inappropriately slow solutions. Further analysis can be carried out during or after an implementation. This section discusses time analysis, starting with an example that involves no implementation in the usual sense.

### The Stair-Counting Problem

Suppose you and your friend Judy are standing at the top of the Eiffel Tower. As you gaze out over the French landscape, Judy turns to you and says, “I wonder how many steps there are to the bottom?” You, of course, are the ever-accommodating host, so you reply, “I’m not sure ... but I’ll find out.” We’ll look at three techniques that you could use and analyze the time requirements of each.

**Technique 1: Walk Down and Keep a Tally.** In the first technique, Judy gives you a pen and a sheet of paper. “I’ll be back in a moment,” you say as you dash down the stairs. Each time you take a step down, you make a mark on the sheet of paper. When you reach the bottom, you run back up, show Judy the piece of paper, and say, “There are this many steps.”

**Technique 2: Walk Down, but Let Judy Keep the Tally.** In the second technique, Judy is unwilling to let her pen or paper out of her sight. But you are undaunted. Once more you say, “I’ll be back in a moment,” and you set off down the stairs. But this time you stop after one step, lay your hat on the step, and run back to Judy. “Make a mark on the piece of paper!” you exclaim. Then you run back to your hat, pick it up, take one more step, and lay the hat down on the second step. Then back up to Judy: “Make another mark on the piece of paper!” you say. You run back down the two stairs, pick up your hat, move to the third step, and lay down the hat. Then back up the stairs to Judy: “Make another mark!” you tell her. This continues until your hat reaches the bottom, and you speed back up the steps one more time. “One more mark, please.” At this point, you grab Judy’s piece of paper and say, “There are this many steps.”

**Technique 3: Jervis to the Rescue.** In the third technique, you don’t walk down the stairs at all. Instead, you spot your friend Jervis by the staircase, holding the sign shown here. The translation is *There are 2689 steps in this stairway (really!).* So you take the paper and pen from Judy, write the number 2689, and hand the paper back to her, saying, “There are this many steps.”

This is a silly example, but even so, it does illustrate the issues that arise when performing a time analysis for an algorithm or program. The first issue is deciding exactly how you will measure the time spent carrying out the work or executing the program. At first glance, the answer seems easy: For each of the three stair-counting techniques, just measure the actual time it takes to carry out the work. You could do this with a stopwatch. But there are some drawbacks to measuring actual time. Actual time can depend on various irrelevant details, such as whether you or somebody else carried out the work. The actual elapsed time may vary from person to person, depending on how fast each person can run the stairs. Even if we decide that *you* are the runner, the time may vary depending on other factors such as the weather, what you had for breakfast, and what other things are on your mind.

So, instead of measuring the actual elapsed time, we count certain operations that occur while carrying out the work. In this example, we will count two kinds of operations:

1. Each time you walk up or down one step, that is one operation.
2. Each time you or Judy marks a symbol on the paper, that is also one operation.



*decide what operations to count*

## 18 Chapter 1 / The Phases of Software Development

Of course, each of these operations takes a certain amount of time, and making a mark may take a different amount of time than taking a step. But this doesn't concern us because we won't measure the actual time taken by the operations. Instead, we will ask: *How many operations are needed for each of the three techniques?*

For the first technique, you take 2689 steps down, another 2689 steps up, and you also make 2689 marks on the paper, for a total of  $3 \times 2689$  operations—that is 8067 total operations.

For the second technique, there are also 2689 marks made on Judy's paper, but the total number of operations is much greater. You start by going down one step and back up one step. Then down two and up two. Then down three and up three, and so forth. The total number of operations taken is shown below.

$$\begin{aligned}\text{Downward steps} &= 3,616,705 \text{ (which is } 1 + 2 + \dots + 2689\text{)} \\ \text{Upward steps} &= 3,616,705 \\ \text{Marks made} &= 2689 \\ \text{Total operations} &= \text{Downward steps} \\ &\quad + \text{Upward steps} \\ &\quad + \text{Marks made} \\ &= 7,236,099\end{aligned}$$

The third technique is the quickest of all: Only four marks are made on the paper (that is, we're counting one "mark" for each digit of 2689), and there is no going up and down stairs. The number of operations used by each of the techniques is summarized here:

Technique 1	8067 operations
Technique 2	7,236,099 operations
Technique 3	4 operations

Doing a time analysis for a program is similar to the analysis of the stair-counting techniques. For a time analysis of a program, usually we do not measure the actual time taken to run the program, because the number of seconds can depend on too many extraneous factors—such as the speed of the processor and whether the processor is busy with other tasks. Instead, the analysis counts the number of operations required. There is no precise definition of what constitutes an **operation**, although an operation should satisfy your intuition of a "small step." An operation can be as simple as the execution of a single program statement. Or we could use a finer notion of operation that counts each arithmetic operation (addition, multiplication, etc.) and each assignment to a variable as a separate operation.

For most programs, the number of operations depends on the program's input. For example, a program that sorts a list of numbers is quicker with a short list than with a long list. In the stairway example, we can view the Eiffel Tower as the input to the problem. In other words, the three different techniques all work on the Eiffel Tower, but the techniques also work on Toronto's CN Tower or on the stairway to the top of the Statue of Liberty or on any other stairway.

dependence on  
input size

When a time analysis depends on the size of the input, then the time can be given as an expression, where part of the expression is the input's size. The time expressions for our three stair-counting techniques are:

Technique 1	$3n$
Technique 2	$n + 2(1 + 2 + \dots + n)$
Technique 3	The number of digits in the number $n$

The expressions in this list give the number of operations performed by each technique when the stairway has  $n$  steps.

The expression for the second technique is not easy to interpret. It needs to be simplified to become a formula that we can easily compare to other formulas. So let's simplify it. We start with the following subexpression:

$$(1 + 2 + \dots + n)$$

There is a trick that will enable us to find a simplified form for this expression. The trick is to *compute twice the amount of the expression and then divide the result by 2*. Unless you've seen this trick before, it sounds crazy. But it works fine. The trick is illustrated in Figure 1.2. Let's go through the computation of that figure step by step.

simplification of  
the time analysis  
for Technique 2

We write the expression  $(1 + 2 + \dots + n)$  twice and add the two expressions. But as you can see in Figure 1.2, we also use another trick: When we write the expression twice, we *write the second expression backward*. After we write down the expression twice, we see the following:

$$\begin{aligned} &(1 + 2 + \dots + n) \\ &+(n + \dots + 2 + 1) \end{aligned}$$

We want the sum of the numbers on these two lines. That will give us twice the value of  $(1 + 2 + \dots + n)$ , and we can then divide by 2 to get the correct value of the subexpression  $(1 + 2 + \dots + n)$ .

Now, rather than proceeding in the most obvious way, we instead add pairs of numbers from the first and second lines. We add the 1 and the  $n$  to get  $n + 1$ . Then we add the 2 and the  $n - 1$  to again get  $n + 1$ . We continue until we reach the last pair consisting of an  $n$  from the top line and a 1 from the bottom line. All the pairs add up to the same amount, namely  $n + 1$ . Now that is handy! We get  $n$  numbers, and all the numbers are the same, namely  $n + 1$ . So the total of all the numbers on the preceding two lines is:

$$n(n + 1)$$

**20** Chapter 1 / The Phases of Software Development

The value of twice the expression is  $n$  multiplied by  $n + 1$ . We are now essentially done. The number we computed is twice the quantity we want. So, to obtain our simplified formula, we only need to divide by 2. The final simplification is thus:

$$(1 + 2 + \dots + n) = \frac{n(n+1)}{2}$$

We will use this simplified formula to rewrite the Technique 2 time expression, but you'll also find that the formula occurs in many other situations. The simplification for the Technique 2 expression is as follows:

Number of operations for Technique 2

$$\begin{aligned} &= n + 2(1 + 2 + \dots + n) \\ &= n + 2 \left( \frac{n(n+1)}{2} \right) \quad \text{Plug in the formula for } (1 + 2 + \dots + n) \\ &= n + n(n+1) \quad \text{Cancel the } 2\text{s} \\ &= n + n^2 + n \quad \text{Multiply out} \\ &= n^2 + 2n \quad \text{Combine terms} \end{aligned}$$

So, Technique 2 requires  $n^2 + 2n$  operations.

**FIGURE 1.2** Deriving a Handy Formula

$(1 + 2 + \dots + n)$  can be computed by first computing the sum of twice  $(1 + 2 + \dots + n)$ , as shown here:

$$\begin{array}{r} 1 \quad + \quad 2 \quad + \dots + (n-1) + \quad n \\ + \quad n \quad + (n-1) + \dots + \quad 2 \quad + \quad 1 \\ \hline (n+1) + (n+1) + \dots + (n+1) + (n+1) \end{array}$$

The sum is  $n(n+1)$ , so  $(1 + 2 + \dots + n)$  is half this amount:

$$(1 + 2 + \dots + n) = \frac{n(n+1)}{2}$$



## 22 Chapter 1 / The Phases of Software Development

quadratic time  
 $O(n^2)$

linear time  $O(n)$

logarithmic time  
 $O(\log n)$

tower will have  $10n$  steps. The number of operations needed for Technique 1 on the taller tower increases tenfold (from  $3n$  to  $3 \times (10n) = 30n$ ); the time for Technique 2 increases approximately 100-fold (from about  $n^2$  to about  $(10n)^2 = 100n^2$ ); and Technique 3 increases by only one operation (from the number of digits in  $n$  to the number of digits in  $10n$ , or to be very concrete, from the four digits in 2689 to the five digits in 26,890). We can express this kind of information in a format called **big- $O$  notation**. The symbol  $O$  in this notation is the letter O, so big- $O$  is pronounced “big Oh.”

We will describe three common examples of the big- $O$  notation. In these examples, we use the notion of “the largest term in a formula.” Intuitively, this is the term with the largest exponent on  $n$  or the term that grows the fastest as  $n$  itself becomes larger. For now, this intuitive notion of “largest term” is enough.

**Quadratic Time.** If the largest term in a formula is no more than a constant times  $n^2$ , then the algorithm is said to be “**big- $O$  of  $n^2$** ,” written  $O(n^2)$ , and the algorithm is called **quadratic**. In a quadratic algorithm, doubling the input size makes the number of operations increase approximately fourfold (or less). For a concrete example, consider Technique 2, requiring  $n^2 + 2n$  operations. A 100-step tower requires 10,200 operations (that is,  $100^2 + 2 \times 100$ ). Doubling the tower to 200 steps increases the time approximately fourfold, to 40,400 operations (that is,  $200^2 + 2 \times 200$ ).

**Linear Time.** If the largest term in the formula is a constant times  $n$ , then the algorithm is said to be “**big- $O$  of  $n$** ,” written  $O(n)$ , and the algorithm is called **linear**. In a linear algorithm, doubling the input size makes the time increase approximately twofold (or less). For example, a formula of  $3n + 7$  is linear, so  $3 \times 200 + 7$  is about twice  $3 \times 100 + 7$ .

**Logarithmic Time.** If the largest term in the formula is a constant times a logarithm of  $n$ , then the algorithm is “**big- $O$  of the logarithm of  $n$** ,” written  $O(\log n)$ , and the algorithm is called **logarithmic**. (The base of the logarithm may be base 10 or possibly another base. We’ll talk about the other bases in Section 10.3.) In a logarithmic algorithm, doubling the input size will make the time increase by no more than a fixed number of new operations, such as one more operation or two more operations—or in general by  $c$  more operations, where  $c$  is a fixed constant. For example, Technique 3 for stair counting has a logarithmic time formula. And doubling the size of a tower (perhaps from 500 stairs to 1000 stairs) never requires more than one extra operation.

Using big- $O$  notation, we can express the time requirements of our three stair-counting techniques as follows:

Technique 1	$O(n)$
Technique 2	$O(n^2)$
Technique 3	$O(\log n)$

**FIGURE 1.3** Number of Operations for Three Techniques

Number of stairs ( $n$ )	Logarithmic $O(\log n)$ Technique 3, with $\lfloor \log_{10} n \rfloor + 1$ operations	Linear $O(n)$ Technique 1, with $3n$ operations	Quadratic $O(n^2)$ Technique 2, with $n^2 + 2n$ operations
10	2	30	120
100	3	300	10,200
1000	4	3000	1,002,000
10,000	5	30,000	100,020,000

When a time analysis is expressed with big- $O$ , the result is called the **order** of the algorithm. We want to reinforce one important point: Multiplicative constants are ignored in the big- $O$  notation. For example, both  $2n$  and  $42n$  are linear formulas, so both are expressed as  $O(n)$ , ignoring the multiplicative constants 2 and 42. As you can see, this means that a big- $O$  analysis loses some information about relative times. Nevertheless, a big- $O$  analysis does provide some useful information for comparing algorithms. The stair example illustrates the most important kind of information provided by the order of an algorithm.

*order of an algorithm*

The order of an algorithm generally is more important than the speed of the processor.

For example, using the quadratic technique (Technique 2), the fastest stair climber in the world is still unlikely to do better than a slowpoke—provided that the slowpoke uses one of the faster techniques. In an application such as sorting a list, a quadratic algorithm can be impractically slow on even moderately sized lists, regardless of the processor speed. To see this, notice the comparisons showing actual numbers for our three stair-counting techniques, which are shown in Figure 1.3.

### Time Analysis of Java Methods

The principles of the stair-climbing example can be applied to counting the number of operations required by code written in a high-level language such as Java. As an example, consider the method implemented in Figure 1.4. The method searches through an array of numbers to determine whether a particular number occurs.

**24 Chapter 1 / The Phases of Software Development****FIGURE 1.4** Specification and Implementation of a search MethodSpecification◆ **search**

```
public static boolean search(double[ ] data, double target)  
Search an array for a specified number.
```

Notice that there is  
no precondition.

**Parameters:**

data – an array of double numbers in no particular order  
target – a specific number that we are searching for

**Returns:**

true (to indicate that target occurs somewhere in the array)  
or false (to indicate that target does not occur in the array)

Implementation

```
public static boolean search(double[ ] data, double target)  
{  
    int i;  
  
    for (i = 0; i < data.length; i++)  
    { // Check whether the target is at data[i].  
        if (data[i] == target)  
            return true;  
    }  
  
    // The loop finished without finding the target.  
    return false;  
}
```

**Examples:**

Suppose that the data array has the five numbers {2, 14, 6, 8, 10}. Then search(data, 10) returns true, but search(data, 42) returns false.

for our first analysis, the number that we are searching for does not occur in the array

As with the stair-climbing example, the first step of the time analysis is to decide precisely what we will count as a single operation. For Java, a good choice is to count the total number of Java operations (such as an assignment, an arithmetic operation, or the < comparison). If a method calls other methods, we would also need to count the operations that are carried out in the other methods.

With this in mind, let's do a first analysis of the search method for the case in which the array's length is a non-negative integer  $n$  and (just to be difficult) the number that we are searching for does not occur in the array. How many operations does the search method carry out in all? Our analysis has three parts:

1. When the for-loop starts, there are two operations: an assignment to initialize the variable  $i$  to 0 and an execution of the test to determine whether  $i$  is less than  $\text{data.length}$ .

2. We then execute the body of the loop, and because the number that we are searching for does not occur, we will execute this body  $n$  times. How many operations occur during each execution of the loop body? We could count this number, but let's just say that each execution of the loop body requires  $k$  operations, where  $k$  is some number around 3 or 4 (including the work at the end of the loop where  $i$  is incremented and the termination test is executed). If necessary, we'll figure out  $k$  later, but for now it is enough to know that we execute the loop body  $n$  times and each execution takes  $k$  operations, for a total of  $kn$  operations.
3. After the loop finishes, there is one more operation (a return statement).

The total number of operations is now  $kn + 3$ . The  $+3$  is from the two operations before the loop and the one operation after the loop. Regardless of how big  $k$  is, this formula is always linear time. So, in the case where the sought-after number does not occur, the search method takes linear time. In fact, this is a frequent pattern that we summarize here:

#### Frequent Linear Pattern

A loop that does a fixed amount of operations  $n$  times  
requires  $O(n)$  time.

Later you will see additional patterns, resulting in quadratic, logarithmic, and other times. In fact, in Chapter 11 you will rewrite the `search` method in a way that uses an array that is sorted from smallest to largest but requires only logarithmic time.

### Worst-Case, Average-Case, and Best-Case Analyses

The `search` method has another important feature: For any particular array size  $n$ , the number of required operations can differ depending on the exact parameter values. For example, with  $n$  equal to 100, the target could be 27, and the very first array element could also be 27—so the loop body executes just one time. On the other hand, maybe the number 27 doesn't occur until `data[99]` and the loop body executes the maximum number of times ( $n$  times). In other words, for any fixed  $n$ , different possible parameter values result in a different number of operations. When this occurs, we usually count the *maximum* number of required operations for inputs of a given size. Counting the maximum number of operations is called the **worst-case** analysis. In fact, the worst case for the `search` method occurs when the sought-after number is not in the array, which is the reason why we used the “not in array” situation in our previous analysis.

During a worst-case time analysis, you may sometimes find yourself unable to provide an exact count of the number of operations. If the analysis is a worst-case analysis, you may estimate the number of operations, always making your

worst-case  
analysis

## 26 Chapter 1 / The Phases of Software Development

estimate on the high side. In other words, the actual number of operations must be guaranteed to be less than the estimate that you use in the analysis.

In Chapter 11, when we begin the study of searching and sorting, you'll see two other kinds of time analysis: **average-case** analysis, which determines the average number of operations required for a given  $n$ , and **best-case** analysis, which determines the fewest number of operations required for a given  $n$ .

### Self-Test Exercises for Section 1.2

11. Write code for a method that computes the sum of all the numbers in an integer array. If the array's length is zero, the sum should also be zero. Do a big- $O$  time analysis of your method.
12. Each of the following are formulas for the number of operations in some algorithm. Express each formula in big- $O$  notation.

a. $n^2 + 5n$	e. $5n + 3n^2$
b. $3n^2 + 5n$	f. The number of digits in $2n$
c. $(n + 7)(n - 2)$	g. The number of times that $n$ can be divided by 10 before dropping below 1.0
d. $100n + 5$	
13. Determine which of the following formulas is  $O(n)$ :

a. $16n^3$	c. $\lfloor n^2/2 \rfloor$
b. $n^2 + n + 2$	d. $10n + 25$
14. What is meant by *worst-case analysis*?
15. What is the worst-case big- $O$  analysis of the following code fragment?

```
k = 0;
for (i = 0; i < n; ++i) {
    for (j = i; j < n; ++j) {
        k += n;
    }
}
```
16. List the following formulas in order of running time analysis, from greatest to least time requirements, assuming that  $n$  is very large:  
 $n^2 + 1$ ;  $50 \log n$ ; 1,000,000;  $10n + 10,000$ .

### 1.3 TESTING AND DEBUGGING

*Always do right. This will gratify some people, and astonish the rest.*

MARK TWAIN

To the Young People's Society, February 16, 1901

*program testing*

**Program testing** occurs when you run a program and observe its behavior. Each time you execute a program using some input, you are testing to see how the program works for that particular input. The topic of this section is the construction of test inputs that are likely to discover errors.

## Choosing Test Data

To serve as good test data, your test inputs need two properties.

### Properties of Good Test Data

1. You must know what output a correct program should produce for each test input.
2. The test inputs should include those inputs that are most likely to cause errors.

Do not take the first property lightly; you must choose test data for which you know the correct output. Just because a program compiles, runs, and produces output that looks about right does not mean the program is correct. If the correct answer is 3278 and the program outputs 3277, then something is wrong. How do you know the correct answer is 3278? The most obvious way to find the correct output value is to work it out with pencil and paper using some method other than that used by the program. To aid this, you might choose test data for which it is easy to calculate the correct answer, perhaps by using smaller input values or by using input values for which the answer is well known.

## Boundary Values

We will focus on two approaches for finding test data that are most likely to cause errors. The first approach is based on identifying and testing inputs called *boundary values*, which are particularly apt to cause errors. A **boundary value** of a problem is an input that is one step away from a different kind of behavior. For example, recall the `dateCheck` method from the first self-test exercise on page 15. It has the following precondition:

### Precondition:

The three arguments are a legal year, month, and day of the month in the years 1900 to 2099.

Two boundary values for `dateCheck` are January 1, 1900 (since one step below this date is illegal) and December 31, 2099 (since one step above this date is illegal). If we expect the method to behave differently for “leap days,” we should try days such as February 28, 2000 (just before a leap day); February 29, 2000 (a leap day); and March 1, 2000 (just after a leap day).

Frequently zero has special behavior, so it is a good idea to consider zero to be a boundary value whenever it is a legal input. For example, consider the `search` method from Figure 1.4 on page 24. This method should be tested with a data array that contains no elements (`data.length` is 0). For example:

*test zero as a boundary value*

## 28 Chapter 1 / The Phases of Software Development

```
double[ ] EMPTY = new double[0]; // An array with no elements  
  
// Searching the EMPTY array should always return false.  
if (search(EMPTY, 0))  
    System.out.println("Wrong answer for an empty array.");  
else  
    System.out.println("Right answer for an empty array.");
```

*test 1 and -1 as boundary values*

The numbers 1 and -1 also have special behavior in many situations, so they should be tested as boundary values whenever they are legal input. For example, the search method should be tested with an array that contains just one element (`data.length` is 1). In fact, it should be tested twice with a one-element array: once when the target is equal to the element and once when the target is different from the element.

*boundary values are “one step away from different behavior”*

In general, there is no precise definition of a boundary value, but you should develop an intuitive feel for finding inputs that are “one step away from different behavior.”

### Test Boundary Values

If you cannot test all possible inputs, at least test the boundary values. For example, if legal inputs range from zero to one million, be sure to test input 0 and input 1000000. It is a good idea also to consider 0, 1, and -1 to be boundary values whenever they are legal input.

## Fully Exercising Code

The second widely used testing technique requires intimate knowledge of how a program has been implemented. The technique, called **fully exercising code**, is stated with two rules:

### Fully Exercising Code

1. Make sure that each line of your code is executed at least once by some of your test data. Make sure that this rare situation is included among your set of test data.
2. If there is part of your code that is sometimes skipped altogether, make sure there is at least one test input that actually does skip this part of your code. For example, there might be a loop where the body is sometimes executed zero times. Make sure that there is a test input that causes the loop body to be executed zero times.

*profiler*

Some Java programming environments have a software tool called a **profiler** to help fully exercise code. A typical profiler will generate a listing indicating how

many times each method was called, so you can easily check that each method has been executed at least once. Some profilers offer more complete information, telling how often each individual statement of your program was executed. This can help you spot parts of your program that were not tested.

Keep in mind that fully exercised code may still have bugs. Code that has been fully exercised has had each line of code tested at least once, but one test does not guarantee that there are no errors.

## PITFALL



### AVOID IMPULSIVE CHANGES

Finding a test input that causes an error is only half the problem of testing and debugging. After an erroneous test input is found, you still must determine exactly why the “bug” occurs and then “debug the program.” When you have found an error, there is an impulse to dive right in and start changing code. It is tempting to look for suspicious parts of your code and change these suspects to something “that might work better.”

Avoid the temptation.

Impulsively changing suspicious code almost always makes matters worse. Instead, you must discover exactly why a test case is failing and limit your changes to corrections of known errors. Once you have corrected a known error, all test cases should be rerun.

### Using a Debugger

Tracking down the reason why a test case is failing can be difficult. For large programs, tracking down errors is nearly impossible without the help of a software tool called a **debugger**. A debugger executes your code one line at a time, or it may execute your code until a certain condition arises. Using a debugger, you can specify what conditions should cause the program execution to pause. You can also keep a continuous watch on the location of the program execution and on the values of specified variables.

### Assert Statements

*An early advocate of using assertions in programming was none other than Alan Turing himself. On 24 June 1950 at a conference in Cambridge, he gave a short talk entitled Checking a Large Routine, which explains the idea with great clarity: “How can one check a large routine in the sense that it’s right? In order that the man who checks may not have too difficult a task, the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole program easily follows.”*

C.A.R. HOARE  
1980 Turing Award Lecture

## 30 Chapter 1 / The Phases of Software Development

Assert statements (also called **assertions**) are boolean expressions that can be checked for validity while a program is running. They were introduced in Java 2, Version 1.4. Assertions can assist in debugging and maintaining programs by documenting conditions that the programmer intends to be valid at particular locations in the program. When a program is running, invalid assertions can be automatically flagged by the Java Runtime Environment, allowing a programmer to spot potential problems as early as possible.

The statement uses the new keyword `assert`, usually following the pattern shown here:

```
assert _____ : "_____";
```

*This is a boolean expression that  
we want to make sure is true at  
this point in the program.*

*This is an error message that  
will be generated if the  
boolean expression is false.*

When an assert statement is reached, the boolean expression is tested. If the expression is true, then no action is taken: the program merely continues executing. But if the expression is false, then an exception called `AssertionError` is thrown. When an `AssertionError` occurs, the method where the failure occurred will stop its computation and usually the program will stop, printing the indicated error message along with an indication of the line number where the `AssertionError` occurred.

For example, consider the `maxOf3` method in Figure 1.5. The computation finds the largest of three numbers; it could be done in many different ways, such as the `if`-statements we have written, or perhaps we could make use of the two-argument `max` method from `java.lang.math`. But however the computation is carried out, it's an easy matter to use assertions to check the validity of the answer with the highlighted assertions at the bottom of the figure.

Notice that the error message (which follows the colon in each assertion) does not need to be on the same line as the assertion's boolean expression. Together, the two assertions verify that the answer was computed correctly—or, if there was an error, the program will stop and print an error message to guide the programmer's debugging effort. This particular example uses the “or” operation (`||`) to ensure that the answer is one of the three original parameters, and it uses the “and” operation (`&&`) to ensure that the answer is not smaller than any of the parameters.

### Turning Assert Statements On and Off

By using assertions at key points (particularly for postconditions), a programmer finds programming errors at an early point when the errors are often easier to correct. However, once a program is ready to release for public use, Java environments permit assertion checking to be turned on or off as needed.

When a program is run with Java, assertions are *not* normally checked. During debugging, however, a programmer can turn on assertion checking by using the `-enableassertions` option (or `-ea`) for the Java runtime system. For example:

```
java -ea TemperatureConversion
```

There are other options that permit the programmer to turn assertions on or off in specific locations, but the general `-ea` option is sufficient to start.

---

**FIGURE 1.5** Two Examples of Assertions

### Specification

◆ **maxOf3**

```
public static int maxOf3(int a, int b, int c)
```

Returns the largest of three `int` values.

**Parameters:**

`a`, `b`, `c` – any `int` numbers

**Returns:**

The return value is the largest of the three arguments `a`, `b`, and `c`.

### Implementation

```
public static int maxOf3(int a, int b, int c)
{
    int answer;

    // Set answer to the largest of a, b, and c:
    answer = a;          // Initially set answer to a
    if (b > answer)    // Maybe change the answer to b
        answer = b;
    if (c > answer)    // Maybe change the answer to c
        answer = c;

    // Check that the computation did what we expected:
    assert (answer == a) || (answer == b) || (answer == c)
        : "maxOf3 answer is not equal to one of the arguments";
    assert (answer >= a) && (answer >= b) && (answer >= c)
        : "maxOf3 answer is not equal to the largest argument";

    return answer;
}
```

---

## 1 PROGRAMMING TIP

### USE A SEPARATE METHOD FOR COMPLEX ASSERTIONS

Some assertions can be implemented with a small boolean expression, such as the two assertions in `maxOf3`. But when the necessary checking becomes more complex, you should write a separate private method (or several methods) that carry out the checking. The method should return a boolean value that can be used in the assertion, as shown in Figure 1.6. Notice that the boolean methods are private (there is no `public` keyword), which means they can be used only within the class in which they appear.

**FIGURE 1.6** The `maxOfArray` Method Together with Methods to Implement Assertions

#### Specification

##### ◆ **maxOfArray**

```
public static int maxOfArray(int[ ])  
    Returns the largest value in an array.
```

##### **Parameters:**

a – a non-empty array (the length must not be zero)

##### **Precondition:**

a.length > 0.

##### **Returns:**

The return value is the largest value in the array a.

##### **Throws:** `ArrayIndexOutOfBoundsException`

Indicates that the array length is zero.

#### Implementation

```
// This private method checks to make sure that the specified value is contained somewhere  
// in the array a. The return value is true if the value is found; otherwise, the return value  
// is false.  
static boolean contains(int[ ] a, int value)  
{  
    int i;  
  
    for (i = 0; i < a.length; i++)  
    {  
        if (a[i] == value)  
            return true;  
    }  
  
    // The loop finished without finding the specified value, so we return false:  
    return false;  
}
```

(continued)

(FIGURE 1.6 continued)

```
// This private method checks to make sure that the specified value is greater than or equal to
// every element in the array a. In this case, the method returns true. On the other hand, if the
// specified value is less than some element in the array, then the method returns false.
static boolean greaterOrEqual(int[ ] a, int value)
{
    int i;

    for (i = 0; i < a.length; i++)
    {
        if (a[i] > value)
            return false;
    }

    // The loop finished without finding an array element that exceeds the value,
    // so we can return true:
    return true;
}

public static int maxOfArray(int[ ] a)
{
    int answer;
    int i;

    // Set answer to the largest value in the array.
    answer = a[0];          // Initially set answer to the first element.
    for (i = 1; i < a.length; i++)
    {
        if (a[i] > answer) // Maybe change the answer to a[i].
            answer = a[i];
    }

    // Check that the computation did what we expected:
    assert contains(a, answer)
        : "maxOfArray answer is not equal in the array";
    assert greaterOrEqual(a, answer)
        : "maxOfArray answer is less than an array element";

    return answer;
}
```

---



## CHAPTER SUMMARY

- The first step in producing a program is to write down a precise description of what the program is supposed to do.
- Pseudocode is a mixture of Java (or some other programming language) and English (or some other natural language). Pseudocode is used to express algorithms so that you are not distracted by details about Java syntax.
- One good method for specifying what a method is supposed to do is to provide a *precondition* and *postcondition* for the method. These form a contract between the programmer who uses the method and the programmer who writes the method.
- *Time analysis* is an analysis of how many operations an algorithm requires. Often, it is sufficient to express a time analysis in big-*O* notation, which is the *order* of an algorithm. The order analysis is often enough to compare algorithms and estimate how running time is affected by changing input size.
- Three important examples of big-*O* analyses are *linear* (i.e.,  $O(n)$ ), *quadratic* (i.e.,  $O(n^2)$ ), and *logarithmic* (i.e.,  $O(\log n)$ ).
- An important testing technique is to identify and test *boundary values*. These are values that lie on a boundary between different kinds of behavior for your program.
- A second testing technique is to ensure that test cases are *fully exercising* the code. A software tool called a *profiler* can aid in fully exercising code.
- During debugging, you should discover exactly why a test case is failing and limit your changes to corrections of known errors. Once you have corrected a known error, all test cases should be rerun. Use a software tool called a *debugger* to help track down exactly why an error occurs.
- Assertions can assist in debugging and maintaining programs by documenting conditions that the programmer intends to be valid at particular locations in the program.



## Solutions to Self-Test Exercises

1. The method returns 7 on July 22, 2013. On both July 30, 2013, and February 1, 2014, the method returns 0 (since July 29, 2013, has already passed).
2. Yes. To use a method, all you need to know is the specification.
3. `System.out.printf("%,.12.2f", bdouble);`
4. **printSqrt**  
`public static void printSqrt(double x)`  
Prints the square root of a number.  
**Parameter:**  
`x` – any non-negative double number  
**Precondition:**  
`x >= 0.`  
**Postcondition:**  
The positive square root of `x` has been printed to standard output.  
**Throws: IllegalArgumentException**  
Indicates that `x` is negative.
5. **sphereVolume**  
`public static double sphereVolume(double radius)`  
Compute the volume of a sphere.  
**Parameter:**  
`radius` – any positive double number  
**Precondition:**  
`radius > 0.`  
**Returns:**  
the volume of a sphere with the specified radius  
**Throws: IllegalArgumentException**  
Indicates that `radius` is negative.
6. `if (x < 0)  
 throw new IllegalArgumentException  
 ("x is negative: " + x);`
7. A **final** variable is given an initial value when it is declared, and this value will never change while the program is running.
8. Most of the specification was automatically produced using the Javadoc tool described in Appendix H.
9. The method name, the parameter list, the return type, and any modifiers.
10. `System.out.println(  
 "Age %10d Height %12.3f",  
 age,  
 height  
)`
11. Here is a specification for the method, along with an implementation:  
**sum**  
`public static int sum(int[] a)`  
Compute the sum of the numbers in an array.  
**Parameter:**  
`a` – an array whose numbers are to be summed  
**Returns:**  
the sum of the numbers in the array (which is zero if the array has no elements)  
`public static int sum(int[] a)  
{  
 int answer, i;  
  
 answer = 0;  
 for (i = 0; i < a.length; i++)  
 answer += a[i];  
 return answer;  
}`  
Our solution uses `answer += a[i]`, which causes the current value of `a[i]` to be added to what's already in `answer`.

*Solutions to Self-Test Exercises* 37

For a time analysis, let  $n$  be the array size. There are two assignment operations ( $i = 0$  and  $answer = 0$ ). The  $<$  test is executed  $n + 1$  times. (The first  $n$  times it is `true`, and the final time, with  $i$  equal to  $n$ , it is `false`.) The `++` and `+=` operations are each executed  $n$  times, and an array subscript ( $a[i]$ ) is taken  $n$  times. The entire code is  $O(n)$ .

12. Part (d) is linear (i.e.,  $O(n)$ ); parts (f) and (g) are logarithmic (i.e.,  $O(\log n)$ ); all of the others are quadratic (i.e.,  $O(n^2)$ ).
13. The only  $O(n)$  formula is (d).
14. Worst-case analysis counts the maximum required number of operations for a function. If the exact count of the number of operations cannot be determined, the number of operations may be estimated, provided that the estimate is guaranteed to be higher than the actual number of operations.
15. This is a nested loop in which the number of times the inner loop executes is one more than the value of the outer loop index. The inner loop statements execute  $n + (n - 1) + \dots + 2 + 1$  times. This sum is  $n(n + 1)/2$  and gives  $O(n^2)$ .
16.  $n^2 + 1$ ;  $10n + 10,000$ ;  $50 \log n$ ; 1,000,000.
17. As always, 0, 1, and -1 are boundary values. In this problem, -20 (smallest value) and 20 (largest value) are also boundary values. Also 9 and 10 (the number changes from a single digit to two digits) and -9 and -10. (By the way, this particular problem is small enough that it would be reasonable to test *all* legal inputs rather than just testing the boundary values.)
18. You should include an empty line (with no characters before the carriage return) and lines with 0, 1, 2, and 3 A's. Also include a line with 4 A's (the smallest case with more than three) and a line with more than 4 A's. For the lines with 1 or more A's, include lines that have only the A's and also include lines that have A's together with other characters. Also test the case in which all the A's appear at the front or the back of the line.
19. See the box "Fully Exercising Code" on page 28.
20. **assert**  
`(java.math.abs(x-z*z*z) < epsilon)`  
`: "Incorrect z value";`
21. If an invalid precondition is detected, it is best to throw an exception that cannot be turned off.



# CHAPTER 2

## Java Classes and Information Hiding

### LEARNING OBJECTIVES

---

When you complete Chapter 2, you will be able to...

- specify, design, and implement classes following a pattern of information hiding that uses private instance variables, accessor methods, and modification methods.
- consistently identify situations in which a class should use a static method.
- place your classes in Java packages and use those packages in other programs.
- implement methods to test for equality of two objects and identify situations that require using this method rather than the `==` operator.
- implement clone methods to make a copy of an object and identify situations that require using this method rather than an assignment statement.
- implement and use methods that have objects for parameters and return values.
- use classes (such as the `java.util.Date` class) from the Java Class Libraries.

### CHAPTER CONTENTS

---

- 2.1 Classes and Their Members
- 2.2 Using a Class
- 2.3 Packages
- 2.4 Parameters, Equals Methods, and Clones
- 2.5 The Java Class Libraries
  - Chapter Summary
  - Solutions to Self-Test Exercises
  - Programming Projects

# Java Classes and Information Hiding

CHAPTER

2

*The happiest way to deal with a man is never to tell him anything he does not need to know.*

ROBERT A. HEINLEIN  
*Time Enough for Love*

**O**bject-oriented programming (**OOP**) is an approach to programming in which data occurs in tidy packages called *objects*. Manipulation of an object happens with functions called *methods*, which are part and parcel of their objects. The Java mechanism to create objects and methods is called a **class**. The keyword `class` at the start of each Java application program indicates that the program is itself a class with its own methods to carry out tasks.

This chapter moves you beyond small Java application programs. Your goal is to be able to write general-purpose classes that can be used by many different programs. Each general-purpose class will capture a certain functionality, and an application programmer can look through the available classes to select those that are useful for the job at hand.

For example, consider a programmer who is writing an application to simulate a Martian lander as it goes from orbit to the surface of Mars. This programmer could use classes to simulate the various mechanical components of the lander—the throttle that controls fuel flow, the rocket engine, and so on. If such classes are readily available in a package of “mechanical component classes,” then the programmer could select and use the appropriate classes. Typically, one programming team designs and implements such classes, and other programmers use the classes. The programmers who use the classes must be provided with a *specification* of how the classes work, but they need no knowledge of how the classes are *implemented*.

The separation of specification from implementation is an example of *information hiding*, which was presented as a cornerstone of program design in Chapter 1. Such a strong emphasis on information hiding is partly motivated by mathematical research into how programmers can improve their reasoning about data types that are used in programs. These mathematical data types are called **abstract data types**, or ADTs—and therefore, programmers sometimes use the term “**ADT**” to refer to a class that is presented to other programmers with information hiding. This chapter presents two examples of such classes. The examples illustrate the features of Java classes, with emphasis on information hiding. By the end of this chapter, you will be able to implement your own classes in Java. Other programmers could *use* one of your classes without knowing the details of *how* you implemented the class.

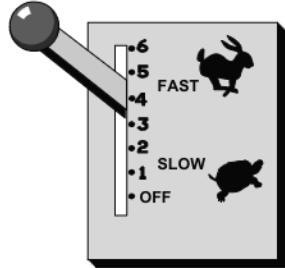
*ADTs emphasize the specification rather than the implementation*

## 2.1 CLASSES AND THEIR MEMBERS

A class is a new kind of data type. Each object of a class includes various *data*, such as integers, characters, and so on. In addition, a class has the ability to include two other kinds of items: *constructors* and *methods*. Constructors are designed to provide initial values to the class's data; methods are designed to manipulate the data. Taken together, the data, constructors, and methods of a class are called the class **members**.

But this abstract discussion does not really tell you what a class *is*. We need some examples. As you read the first example, concentrate on learning the techniques for implementing a class. Also notice how you use a class that was written by another programmer, without knowing details of the class's implementation.

### PROGRAMMING EXAMPLE: The Throttle Class



Our first example of a class is a new data type to store and manipulate the status of a mechanical throttle. An object of this new class holds information about one throttle, as shown in the picture. The throttle is a lever that can be moved to control fuel flow. The throttle we have in mind has a single shutoff point (where there is no fuel flow) and a sequence of several on positions where the fuel is flowing at progressively higher rates. At the topmost position, the fuel flow is fully on. At the intermediate positions, the fuel flow is proportional to the location of the lever. For example, with six possible positions and the lever in the fourth position, the fuel flows at  $\frac{4}{6}$  of its maximum rate.

A constructor is designed to provide initial values to a class's data. The throttle constructor permits a program to create a new throttle with a specified number of "on positions" above the shutoff point. For instance, a throttle for a lawn mower could specify six positions, whereas a throttle for a Martian lander could specify 1000 positions. The throttle's lever is initially placed at the shutoff point.

Once a throttle has been initialized, there are two methods to shift the throttle's lever: One of the methods shifts the lever by a given amount, and the other method returns the lever to the shutoff position. We also have two methods to examine the status of a throttle. The first of these methods returns the amount of fuel currently flowing, expressed as a proportion of the maximum flow. For example, this method will return approximately 0.667 when a six-position throttle is in its fourth position. The other method returns a true or false value, telling whether the throttle is currently on (that is, whether the lever is above the shutoff point). Thus, the throttle has one constructor and four methods:

one throttle  
constructor and  
four throttle  
methods

- A constructor to create a new throttle with one shutoff point and a specified number of on positions (the lever starts at the shutoff point)
- A method that returns the fuel flow, expressed as a proportion of the maximum flow
- A method to tell us whether the throttle is currently on
- A method to shift a throttle's lever by a given amount
- A method to set the throttle's lever back to the shutoff point

## Defining a New Class

We're ready to define a new Java class called `Throttle`. The new class includes data (to store information about the throttle) plus the constructor and methods previously listed. Once the `Throttle` class is defined, a programmer can create objects of type `Throttle` and manipulate those objects with the methods.

Here's an outline of the `Throttle` class definition:

```
public class Throttle
{
    private int top;      // The topmost position of the lever
    private int position; // The current position of the lever
```

This part of the class definition provides the implementations  
of the constructor and methods.

declaring the  
`Throttle` class

This class definition defines a new data type called `Throttle`. The definition starts with the **class head**, which consists of the Java keywords `public class` followed by the name of the new class. The keyword `public` is necessary before the `class` because we want to allow all other programmers (the “public”) to use the new class. The name of the class can be any legal identifier. We chose the name `Throttle`. We always use a capital letter for the first character of names of new classes. This isn't required by Java, but it's a common programming style, making it easy to identify class names.

The rest of the class definition, between the two brackets, is a list of all the components of the class. These components are called **members** of the class, and they come in three varieties: instance variables, constructors, and methods.

three varieties of  
class members  
appear in the  
class definition

## Instance Variables

The first kind of member is a variable declaration. These variables are called **instance variables** (or sometimes “member variables”). The `Throttle` class has two instance variables:

```
private int top;      // The topmost position of the lever
private int position; // The current position of the lever
```

Each instance variable stores some piece of information about the status of an object. For example, consider a throttle with six possible positions where the lever is in the fourth position. This throttle would have `top=6` and `position=4`.

The keyword `private` occurs in front of each of our instance variables. This keyword means that programmers who use the new class have no way to read or assign values directly to the private instance variables. It is possible to have public instance variables that can be accessed directly, but public instance variables tend to reveal too much information about how a class is implemented, violating

## 42 Chapter 2 / Java Classes and Information Hiding

the principle of information hiding. Therefore, our examples will use private instance variables. Other programmers must access the private instance variables through the constructors and methods provided with the class.

### Constructors

The second kind of member is a constructor. (Technically, a constructor is not a member, but the reason is a small concern that we'll see in Chapter 13, and most programmers do think of constructors as members.) A constructor is a method that is responsible for initializing the instance variables. For example, our constructor creates a throttle with a specified number of on positions above the shutoff position. This constructor sets the instance variable `top` to a specified number and sets `position` to zero (so that initially the throttle is shut off).

For the most part, implementing a constructor is no different than your past work (such as implementing a method for a Java application). The primary difference is that a constructor has access to the class's instance variables, and it is responsible for initializing these variables. Thus, a throttle constructor must provide initial values to `top` and `position`. Before you implement the throttle constructor, you must know several rules that make constructors special:

- Before any constructor begins its work, all instance variables are assigned Java "default values." For example, the Java default value for any number variable is zero.
- If an instance variable has an initialization value with its declaration, the initialization value replaces the default value. For example, suppose we have this instance variable:

```
int jackie = 42;
```

The instance variable `jackie` is first given its default value of zero; then the zero is replaced by the initialization value of 42.

- The name of a constructor must be the same as the name of the class. In our example, the name of the constructor is `Throttle`. This seems strange: Normally we *avoid* using the same name for two different things. But it is a requirement of Java that the constructor use the same name as the class.
- A constructor is not really a method, and therefore it does not have *any* return value. Because of this, you must *not* write `void` (or any other return type) at the front of the constructor's head. The compiler knows that every constructor has no return value, but a compiler error occurs if you actually write `void` at the front of the constructor's head.

With these rules, we can write the throttle's constructor as shown here (with its specification following the format from Section 1.1):

#### ◆ Constructor for the Throttle

```
public Throttle(int size)
```

Construct a Throttle with a specified number of on positions.

**Parameter:**

size – the number of on positions for this new Throttle

**Precondition:**

size > 0

**Postcondition:**

This Throttle has been initialized with the specified number of on positions above the shutoff point, and it is currently shut off.

**Throws: IllegalArgumentException**

Indicates that size is not positive.

```
public Throttle(int size)
{
    if (size <= 0)
        throw new IllegalArgumentException("Size <= 0: " + size);
    top = size;
    // No assignment needed for position -- it gets the default value of zero.
}
```

This constructor sets `top` according to the parameter, `size`. It does not explicitly set `position`, but the comment in the implementation indicates that we did not just forget about `position`; the default value of zero is its correct initial value. The implementation is preceded by the keyword `public` to make it available to all programmers.

The throttle has just one constructor, and so just one way of setting the initial values of the instance variables. Some classes may have many different constructors that set initial values in different ways. If there are several constructors, then each constructor must have a distinct sequence of parameters to distinguish it from the other constructors.

a class can have  
many different  
constructors

## No-Arguments Constructors

Some classes have a constructor with no parameters, called a **no-arguments** constructor. In effect, a no-arguments constructor does not need any extra information to set the initial values of the instance variables.

If you write a class with no constructors at all, then Java automatically provides a no-arguments constructor that initializes each instance variable to its initialization value (if there is one) or to its default value (if there is no specified initialization value). There is one situation in which Java does not provide an automatic no-arguments constructor, and you'll see this situation when you write subclasses in Chapter 13.

## Methods

The third kind of class member is a method. A method does computations that access the class's instance variables. Classes tend to have two kinds of methods:

## 44 Chapter 2 / Java Classes and Information Hiding

1. An **accessor method** gives information about an object without altering the object. In the case of the throttle, an accessor method can return information about the status of a throttle, but it must not change the position of the lever.
2. A **modification method** can change the status of an object. For a throttle, a modification method can shift the lever up or down.

Each class method is designed for a specific manipulation of an object—in our case, the manipulation of a throttle. To carry out the manipulations, each of the throttle methods has access to the throttle's instance variables, `top` and `position`. The methods can examine `top` and `position` to determine the current status of the throttle, or `top` and `position` can be changed to alter the status of the throttle. Let's look at the details of the implementations of the throttle methods, beginning with the accessor methods.

### Accessor Methods

Accessor methods provide information about an object without changing the object. Accessor methods are often short, just returning the value of an instance variable or performing a computation with a couple of instance variables. The first of the throttle accessor methods computes the current flow as a proportion of the maximum flow. The specification and implementation are:

◆ **getFlow**

```
public double getFlow( )  
Get the current flow of this Throttle.
```

**Returns:**

the current flow rate (always in the range [0.0 ... 1.0] ) as a proportion of the maximum flow

```
public double getFlow( )  
{  
    return (double) position / (double) top;  
}
```

*accessor  
methods often  
have no  
parameters*

Accessor methods often have no parameters, no precondition, and only a simple return condition in the specification. How does an accessor method manage with no parameters? It needs no parameters because all of the necessary information is available in the instance variables.



### PROGRAMMING TIP

#### FOUR REASONS TO IMPLEMENT ACCESSOR METHODS

Many classes could be written without accessor methods by making the instance variables public instead of private. For example, the `getFlow` method would not be needed if `position` and `top` were public instance variables (because a programmer could then use those variables directly). However, providing private

instance variables and accessor methods does a better job of information hiding, accomplishing these things:

1. A programmer who uses the throttle need not worry about how it is implemented. (It's always good to have fewer details to worry about.)
2. Using accessor methods allows us to later change the implementation of the throttle, perhaps adding a new member variable that always keeps track of the current flow. Programs that use the throttle will still work correctly after we make such changes, provided we have one set of accessor methods whose implementations do not change.
3. When a class implements an accessor method, that method can be thoroughly tested, increasing reliability. Without accessor methods, each program that uses the class is prone to the same potential errors. For example, one potential error in the `getFlow` method is a division error that we'll describe in a moment.
4. The pattern of "private data, public methods" forbids other programmers from using our instance variables in unintended ways (such as setting `position` to a negative value).

Java provides many classes for programmers to use. If you read the specifications for these classes (called the **Application Programmers Interface**, or **API**), you'll see the prevalence of accessor methods in professionally written classes.

## PITFALL



### INTEGER DIVISION THROWS AWAY THE FRACTIONAL PART

The `getFlow` implementation computes and returns a fractional value. For example, if `position` is 4 and `top` is 6, then `getFlow` returns approximately 0.667. To get a fractional result in the answer, the integer numbers `position` and `top` cannot simply be divided using the expression `position/top` because this would result in an integer division ( $\frac{4}{6}$  results in the quotient 0, discarding any remainder). Instead, we must force Java to compute a fractional division by changing the integer values to double values. For example, the expression `(double) position` is a "cast" that changes the integer value of `position` to a double value to use in the division.

The throttle's second accessor method returns a true or false value indicating whether the fuel flow is on. Here is this method with its specification:

#### ◆ **isOn**

```
public boolean isOn()
```

Check whether this Throttle is on.

#### **Returns:**

If this Throttle's flow is above zero, then the return value is `true`; otherwise, the return value is `false`.



off the flow. However, most modification methods do have parameters. For example, consider a throttle method to shift the throttle's lever by a specified amount. This `shift` method has one integer parameter called `amount`. If `amount` is positive, then the throttle's lever is moved up by that amount (but never beyond the topmost position). A negative `amount` causes the lever to move down (but never below zero). Here are the specification and implementation:

◆ **shift**

```
public void shift(int amount)  
Move this Throttle's position up or down.
```

**Parameter:**

`amount` – the amount to move the position up or down (a positive amount moves the position up; a negative amount moves it down)

**Postcondition:**

This Throttle's position has been moved by the specified amount. If the result is more than the topmost position, then the position stays at the topmost position. If the result is less than the zero position, then the position stays at the zero position.

```
public void shift(int amount)  
{  
    if (amount > top - position)  
        // Adding amount would put the position above the top.  
        position = top;  
    else if (position + amount < 0)  
        // Adding amount would put the position below zero.  
        position = 0;  
    else  
        // Adding amount puts position in the range [0...top].  
        position += amount;  
}
```

This might be the first time you've seen the `+=` operator. Its effect is to take the value on the right side (such as `amount`) and add it to what's already in the variable on the left (such as `position`). This sum is then stored back in the variable on the left side of `+=`.

The `shift` method requires care to ensure that the position does not go above the topmost position nor below zero. For example, the first test in the method checks whether `(amount > top - position)`. If so, then adding `amount` to `position` would push the position over `top`. In this case, we simply set `position` to `top`.

It is tempting to write the test `(amount > top - position)` in a slightly different way, like this:

```
if (position + amount > top)  
    // Adding amount would put the position above the top.  
    position = top;
```

## 48 Chapter 2 / Java Classes and Information Hiding

This seems okay at first glance, but there is a potential problem: What happens if both `position` and `amount` are large integers, such as 2,000,000,000? The subexpression `position + amount` should be 4,000,000,000, but Java tries to temporarily store the subexpression as a Java integer, which is limited to the range -2,147,483,648 to 2,147,483,647. The result is an **arithmetic overflow**, which is defined as trying to compute or store a number that is beyond the legal range of the data type. When an arithmetic overflow occurs, the program might stop with an error message, or it might continue computing with wrong data.

We avoided the arithmetic overflow by rearranging the first test to avoid the troublesome subexpression. The test we use is:

```
if (amount > top - position)
    // Adding amount would put the position above the top.
    position = top;
```

This test uses the subexpression `top - position`. Because `top` is never negative and `position` is in the range [0...`top`], the subexpression `top - position` is always a valid integer in the range [0...`top`].

What about the second test in the method? In the second test, we use the subexpression `position + amount`, but at this point, `position + amount` can no longer cause an arithmetic overflow. Do you see why? If `position + amount` is bigger than `top`, then the first test would have been `true`, and the second test is never reached. Therefore, by the time we reach the second test, the subexpression `position + amount` is guaranteed to be in the range [`amount`...`top`], and arithmetic overflow cannot occur.



### PITFALL

#### POTENTIAL ARITHMETIC OVERFLOWS

Check arithmetic expressions for potential arithmetic overflow. The limitations for Java variables and subexpressions are given in Appendix A. Often you can rewrite an expression to avoid overflow, or you can use `long` variables (with a range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807). If overflow cannot be avoided altogether, then include a note in the documentation to describe the situation that causes overflow.

#### Complete Definition of Throttle.java

*the name of the Java file must match the name of the class*

We have completed the `Throttle` class implementation and can now put the complete definition in a file called `Throttle.java`, as shown in Figure 2.1. The name of the file must be `Throttle.java` since the class is named `Throttle`.

**FIGURE 2.1** Specification and Implementation for the Throttle Class

### Class Throttle

#### ❖ public class Throttle

A Throttle object simulates a throttle that is controlling fuel flow.

### Specification

#### ◆ Constructor for the Throttle

```
public Throttle(int size)
```

Construct a Throttle with a specified number of on positions.

**Parameter:**

size – the number of on positions for this new Throttle

**Precondition:**

size > 0

**Postcondition:**

This Throttle has been initialized with the specified number of on positions above the shutoff point, and it is currently shut off.

**Throws:** IllegalArgumentException

Indicates that size is not positive.

#### ◆ getFlow

```
public double getFlow()
```

Get the current flow of this Throttle.

**Returns:**

the current flow rate (always in the range [0.0 ... 1.0] ) as a proportion of the maximum flow

#### ◆ isOn

```
public boolean isOn()
```

Check whether this Throttle is on.

**Returns:**

If this Throttle's flow is above zero, then the return value is true; otherwise, the return value is false.

#### ◆ shift

```
public void shift(int amount)
```

Move this Throttle's position up or down.

**Parameter:**

amount – the amount to move the position up or down (a positive amount moves the position up; a negative amount moves it down)

**Postcondition:**

This Throttle's position has been moved by the specified amount. If the result is more than the topmost position, then the position stays at the topmost position. If the result is less than the zero position, then the position stays at the zero position.

(continued)

**50 Chapter 2 / Java Classes and Information Hiding**

(FIGURE 2.1 continued)

♦ **shutOff**

```
public void shutOff()
```

Turn off this Throttle.

**Postcondition:**

This Throttle has been shut off.

Implementation

```
// File: Throttle.java

public class Throttle
{
    private int top;      // The topmost position of the throttle
    private int position; // The current position of the throttle

    public Throttle(int size)
    {
        if (size <= 0)
            throw new IllegalArgumentException("Size <= 0: " + size);
        top = size;
        // No assignment needed for position -- it gets the default value of zero.
    }

    public double getFlow()
    {
        return (double) position / (double) top;
    }

    public boolean isOn()
    {
        return (getFlow() > 0);
    }

    public void shift(int amount)
    {
        if (amount > top - position) // Adding amount makes position too big.
            position = top;
        else if (position + amount < 0) // Adding amount makes position below zero.
            position = 0;
        else // Adding amount puts position in the range [0 ... top].
            position += amount;
    }

    public void shutOff()
    {
        position = 0;
    }
}
```

---

## Methods May Activate Other Methods

The throttle's `isOn` method in Figure 2.1 has one change from the original implementation. The change is highlighted here:

```
public boolean isOn( )
{
    return (getFlow( ) > 0);
}
```

In this implementation, we have checked whether the flow is on by calling the `getFlow` method rather than by looking directly at the `position` instance variable. Both implementations work: Using `position` directly probably executes quicker, but you could argue that using `getFlow` makes the method's intent clearer. Anyway, the real purpose of this change is just to illustrate that one method can call another to carry out a subtask. In this example, the `isOn` method calls `getFlow`. An OOP programmer usually would use slightly different terminology, saying that the `isOn` method **activated** the `flow` method. **Activating a method** is nothing more than OOP jargon for “calling a method.”

## Self-Test Exercises for Section 2.1

1. Describe the three kinds of class members we have used. In this section, which kinds of members were public and which were private?
2. We talked about accessor methods and modification methods. Which kind of method often has a `void` return type? Why?
3. Write a new throttle constructor with no arguments. The constructor sets the top position to 1 and sets the current position to off.
4. Write another throttle constructor with two arguments: the total number of positions for the throttle, and its initial position.
5. Describe the difference between a modification method and an accessor method.
6. Add a new throttle method that will return `true` when the current flow is more than half of the maximum flow. The body of your implementation should activate the `getFlow` method.
7. Design and implement a class called `Clock`. A `Clock` object holds one instance of a time such as 9:48 P.M. Have at least these public methods:
  - A no-arguments constructor that initializes the time to midnight (see page 43 for the discussion of a no-arguments constructor)
  - A method to explicitly assign a given time (you will have to give some thought to appropriate arguments for this method)
  - Methods to retrieve information: the current hour, the current minute, and a boolean method that returns true if the time is at or before noon
  - A method to advance the time forward by a given number of minutes (which could be negative to move the clock backward or positive to move the clock forward)

**52 Chapter 2 / Java Classes and Information Hiding**

8. Suppose you implement a class, but you do not write a constructor. What kind of constructor does Java usually provide automatically?

## 2.2 USING A CLASS

*programs can  
create new  
objects of a  
class*

How do you use a new class such as `Throttle`? Within any program, you can create new throttles and refer to these throttles by names that you define. We will illustrate the syntax for creating and using these objects with an example.

### Creating and Using Objects

Suppose a program needs a new throttle with 100 positions above the shutoff. Within the program, we want to refer to the throttle by the name `control`. The Java syntax has these parts:

```
Throttle control = new Throttle(100);
```

The first part of this statement—`Throttle control`—declares a new variable called `control`. The `control` variable is capable of referring to a throttle. The second part of the statement—`new Throttle(100)`—creates a new throttle and initializes `control` to refer to this new throttle. A new throttle that is created in this way is called a **Throttle object**.

There are a few points to notice about the syntax for creating a new `Throttle` object: `new` is a keyword to create a new object; `Throttle` is the data type of the new object; and `(100)` is the list of arguments for the constructor of the new object. We are creating a new throttle and 100 is the argument for the constructor, so the new throttle will have 100 positions above the shutoff.

*using a no-  
arguments  
constructor*

If the class has a no-arguments constructor, then the syntax for creating an object is the same, except that the list of arguments is empty. For example, suppose that `Thermometer` is a class with a no-arguments constructor. Then we could write this:

```
Thermometer t = new Thermometer();
```

Once an object is created, we can refer to that object by the name we declared. For example, suppose we have created a throttle called `control` and we want to shift the lever up to its third notch. We do this by calling the `shift` method:

```
control.shift(3);
```

Calling a method always involves these four pieces:

1. Start with a reference to the object you are manipulating. In this example, we want to manipulate `control`, so we begin with “`control`”. Remember

that you cannot just call a method; you must always indicate which object is being manipulated.

2. Next, place a single period.
3. Next, write the name of the method. In our example, we call the `shift` method, so we write “`control.shift`”—which you can pronounce “control dot shift.”
4. Finally, list the parameters for the method call. In our example, `shift` requires one parameter, which is the amount (3) that we are shifting the throttle. Thus, the entire statement is `control.shift(3);`

*how to use a method*

Our example called the `shift` method. As you’ve seen before, OOP programmers like their own terminology, and they would say that we **activated** the `shift` method. In the rest of the text, we’ll try to use “activate” rather than “call.” (This will keep us on the good side of OOP programmers.)

As another example, here is a sequence of several statements to set a throttle to a certain point and then print the throttle’s flow:

```
final int SIZE = 8; // The size of the Throttle
final int SPOT = 3; // Where to move the Throttle's lever

Throttle small = new Throttle(SIZE);

small.shift(SPOT);
System.out.print("My small throttle is now at position ");
System.out.println(SPOT + " out of " + SIZE + ".");
System.out.println("The flow is now: " + small.getFlow());
```

Notice how the return value of `small.getFlow` is used directly in the output statement. As with any other method, the return value of an accessor method can be used as part of an output statement or other expression. The output from this code is:

```
My small throttle is now at position 3 out of 8.
The flow is now: 0.375
```

### A Program with Several Throttle Objects

A single program can have many throttle objects. For example, the following code will declare two throttle objects, shifting each throttle to a different point:

```
Throttle tiny = new Throttle(4);
Throttle huge = new Throttle(10000);

tiny.shift(2);
huge.shift(2500);
```

## 54 Chapter 2 / Java Classes and Information Hiding

Here's an important concept to keep in mind:

When a program has several objects of the same class, each object has its own copies of the class's instance variables.

In the preceding example, `tiny` has its own instance variables (`top` will be 4 and `position` will be 2); `huge` also has its own instance variables (`top` will be 10,000 and `position` will be 2500). When we activate a method such as `tiny.shift`, the method uses the instance variables from `tiny`; when we activate `huge.shift`, the method uses the instance variables from `huge`.

The variables in our examples—`control`, `small`, `tiny`, `huge`—are called **reference variables** because they are used to *refer* to objects (in our case, `Throttles`). There are several differences between a reference variable (used by Java for all classes) and an ordinary variable (used by Java for the primitive data types of `int`, `char`, and so on). Let's look at these differences, beginning with a special value called `null` that is used only with reference variables.

### Null References

The creation of a new object can be separated from the declaration of a variable. For example, the following two statements can occur far apart in a program:

```
Throttle control;  
...  
control = new Throttle(100);
```

Once both statements finish, `control` refers to a newly created `Throttle` with 100 positions. But what is the status of `control` between the statements? At that point, `control` does not yet refer to any `Throttle` because we haven't yet created a `Throttle`. In this situation, we can assign a special value to `control`, indicating that `control` does not yet refer to anything. The value is called the **null reference**, written with the keyword `null` in Java. So we could change the example to this:

```
Throttle control = null;  
...  
control = new Throttle(100);
```

*In this area, control  
does not refer to  
anything.*

#### Null Reference

Sometimes a reference variable does not refer to anything. This is a **null reference**, and the value of the variable is called **null**.

Sometimes a program finishes using an object. In this case, the program may explicitly set a reference variable to `null`, as shown here:

```
Throttle control = new Throttle(100);  
// Various statements that use the Throttle appear next...  
...  
// Now we are done with the control Throttle, so we can set the reference to null.  
control = null;
```

Once a reference variable is no longer needed, it can be set to `null`, which allows Java to economize on certain resources (such as the memory used by a throttle).

## PITFALL

### NULLPOINTEREXCEPTION

When a variable such as `control` becomes `null`, it no longer refers to any throttle. If `control` is `null`, then it is a programming error to activate a method such as `control.shift`. The result is an exception called `NullPointerException`.

### Assignment Statements with Reference Variables

The usual assignment statement can be used with reference variables. For example, we might have two `Throttle` variables `t1` and `t2`, and an assignment such as `t2 = t1` is permitted. But what is the effect of the assignment? For starters, if `t1` is `null`, then the assignment `t2 = t1` also makes `t2` `null`. Here is a more complicated case in which `t1` is not `null`:

```
Throttle t1;  
Throttle t2;  
  
t1 = new Throttle(100);  
t1.shift(25);  
t2 = t1;
```

The effect of the assignment `t2 = t1` is somewhat different than assignments for integers or other primitive data types. The effect of `t2 = t1` is to “make `t2` refer to the same object that `t1` is already referring to.” In other words, we have two reference variables (`t1` and `t2`), but we created only one throttle (with one `new` statement). This one throttle has 100 positions and is currently in the 25<sup>th</sup> position. After the assignment statement, both `t1` and `t2` refer to this one throttle.

To explore this example in more detail, let’s start with the two declarations:

```
Throttle t1;  
Throttle t2;
```

## 56 Chapter 2 / Java Classes and Information Hiding

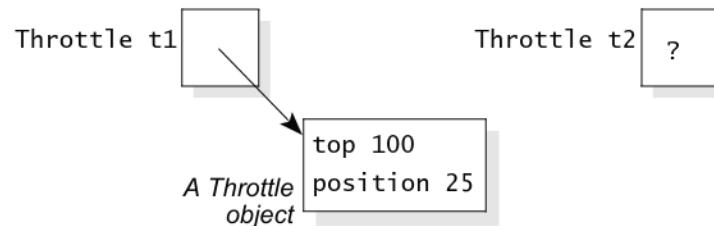
We now have two variables, `t1` and `t2`. If these variables are declared in a method, then they don't yet have an initial value (not even `null`). We can draw this situation with a question mark for each value:



The next two statements are:

```
t1 = new Throttle(100);  
t1.shift(25);
```

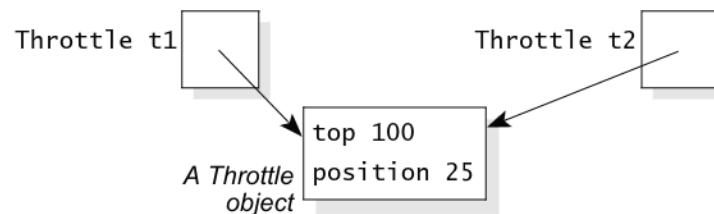
These statements create a new throttle for `t1` to refer to and shift the throttle's position to 25. We will draw a separate box for the throttle and indicate its instance variables (`top` at 100 and `position` at 25). To show that `t1` refers to this throttle, we draw an arrow from the `t1` box to the throttle:



At this point, we can execute the assignment:

```
t2 = t1;
```

After the assignment, `t2` will refer to the same object that `t1` refers to:

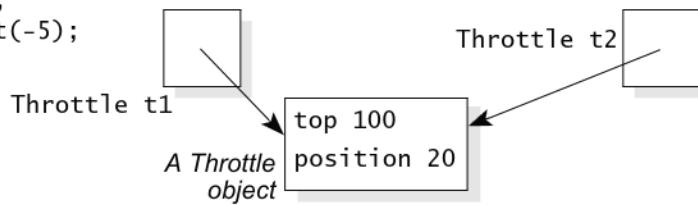


There are now two references to the same throttle, which can cause some surprising results. For example, suppose we shift `t2` down five notches and then print the flow of `t1`, like this:

```
t2.shift(-5);  
System.out.println("Flow of t1 is: " + t1.getFlow());
```

What flow rate is printed? The `t1` throttle was set to position 25 out of 100, and we never *directly* altered its position. But `t2.shift(-5)` moves the throttle's position down to 20. Since `t1` refers to this same throttle, `t1.getFlow` now returns 20/100, and the output statement prints "Flow of t1 is: 0.2". Here's the entire code that we executed and the final situation drawn as a picture:

```
Throttle t1;  
Throttle t2;  
  
t1 = new Throttle(100);  
t1.shift(25);  
t2 = t1;  
t2.shift(-5);
```



### Assignment Statements with Reference Variables

If `t1` and `t2` are reference variables, then the assignment `t2 = t1` is allowed.

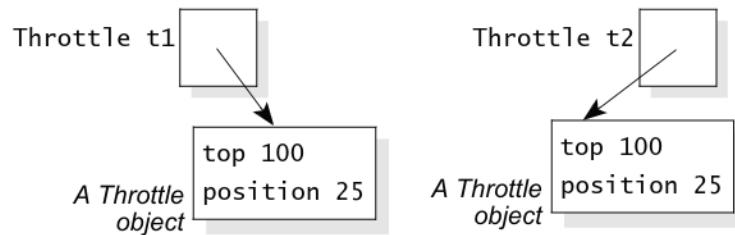
If `t1` is `null`, then the assignment makes `t2` `null` also.

If `t1` is not `null`, then the assignment changes `t2` so that it refers to the same object that `t1` already refers to. At this point, changes can be made to that one object through either `t1` or `t2`.

The situation of an assignment statement contrasts with a program that actually creates two separate throttles for `t1` and `t2`. For example, two separate throttles can be created with each throttle in the 25<sup>th</sup> position out of 100:

```
Throttle t1;  
Throttle t2;  
  
t1 = new Throttle(100);  
t1.shift(25);  
t2 = new Throttle(100);  
t2.shift(25);
```

With this example, we have two separate throttles, as shown on the next page.

**58 Chapter 2 / Java Classes and Information Hiding**

Changes that are now made to one throttle will not affect the other, because there are two completely separate throttles.

### Clones

A programmer sometimes needs to make an exact copy of an existing object. The copy must be just like the existing object, but separate. Subsequent changes to the copy should not alter the original, nor should subsequent changes to the original alter the copy. A separate copy such as this is called a **clone**.

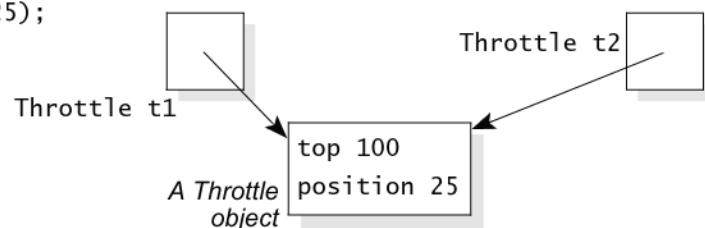
An assignment operation `t2 = t1` does not create a clone, and in fact the `Throttle` class does not permit the easy creation of clones. But many other classes have a special method called `clone` for just this purpose. Writing a useful `clone` method has some requirements that may not be evident just now, so we will postpone a complete discussion until Section 2.4.

### Testing for Equality

A test for equality (`t1 == t2`) can be carried out with reference variables. The equality test (`t1 == t2`) is true if both `t1` and `t2` are null or if they both refer to the exact same object (not two different objects that happen to have the same values for their instance variables). An inequality test (`t1 != t2`) can also be carried out. The result of an inequality test is always the opposite of an equality test. Let's look at two examples.

The first example creates just one throttle; `t1` and `t2` both refer to this throttle, as shown in the following picture:

```
Throttle t1;  
Throttle t2;  
  
t1 = new Throttle(100);  
t1.shift(25);  
t2 = t1;
```





**60 Chapter 2 / Java Classes and Information Hiding**

12. Suppose that `t1` and `t2` are throttle variables and that `t1` is `null` but `t2` refers to a valid throttle. Which of these statements will cause an error?  
`t1 = t2;    t2 = t1;    t1.shift(2);`
13. This question tests your understanding of assignment statements with reference variables. What is the output of this code?  

```
Throttle t1;
Throttle t2;
t1 = new Throttle(100);
t2 = t1;
t1.shift(40);
t2.shift(2);
System.out.println(t1.getFlow( ));
```
14. Consider the code from the preceding question. At the end of the computation, is `(t1 == t2)` true or false?
15. Write some code that will make `t1` and `t2` refer to two different throttles with 100 positions each. Both throttles are shifted up to position 42. At the end of your code, is `(t1 == t2)` true or false?
16. Suppose you have a class called `Thermometer`. The class has a no-arguments constructor that sets the thermometer's temperature to zero Celsius. There are two modification methods (`setCelsius` and `setFahrenheit`) to change the temperature by a specified amount. (One method has a parameter in Celsius degrees, and the other has a parameter in Fahrenheit.) There are two accessor methods to get the current temperature (`getCelsius` and `getFahrenheit`). A thermometer keeps track of only one temperature, but it can be accessed in either Celsius or Fahrenheit. Write code to create a thermometer, add 10 degrees Celsius to it, and then print the Fahrenheit temperature.

## 2.3 PACKAGES

You now know enough to write a Java application program that uses a throttle. The `Throttle` class would be in one file (`Throttle.java` from Figure 2.1 on page 50), and the program that uses the `Throttle` class would be in a separate file. However, there's one more level of organization called a Java **package**, which we discuss in this section. The package makes it easy to group together related classes, with each class remaining in a separate `.java` file.

### Declaring a Package

The first step in declaring a package of related classes is to decide on a name for the package. For example, perhaps we are declaring a bunch of Java classes to simulate various real-world devices, such as a throttle. A good short name for the package would be the `simulations` package. But there's a problem with good short names: Other programmers might decide to use the same good short name for their packages, resulting in the same name for two or more different packages.

The solution is to include your Internet domain name as part of the package name. For example, at the University of Colorado, the Internet domain name is `colorado.edu`. Therefore, instead of using the package name `simulations`, I will use the longer package name `edu.colorado.simulations`. (Package names may include a “dot” as part of the name.) Many programmers follow this convention, using the Internet domain name in reverse. The only likely conflicts are with other programmers at your own Internet domain, and those conflicts can be prevented by internal cooperation.

use your Internet domain name

Once you have decided on a package name, a *package declaration* must be made at the top of each `.java` file of the package. The **package declaration** consists of the keyword `package` followed by the package name and a semicolon. For example, the start of `Throttle.java` must include this package declaration:

```
package edu.colorado.simulations;
```

The revised `Throttle.java`, with a package declaration, is shown in Figure 2.2. Some Java development environments require you to create a directory structure for your classes to match the structure of package names. For example, suppose you are doing your code development in your own directory called `classes`, and you want to use the `edu.colorado.simulations` package. You would follow these steps:

- Make sure your Java development environment can find and run any classes in your `classes` directory. The exact method of setting this up varies from one environment to another, but a typical approach is to define a system `CLASSPATH` variable to include your own `classes` directory.
- Underneath the `classes` directory, create a subdirectory called `edu`.
- Underneath `edu`, create a subdirectory called `colorado`.
- Underneath `colorado`, create a subdirectory called `simulations`.
- All the `.java` and `.class` files for the package must be placed in the `simulations` subdirectory.

If the `edu.colorado.simulations` package has other classes, then their files are also placed in the `simulations` subdirectory, and the package declaration is placed at the start of each `.java` file.

**FIGURE 2.2** Defining `Throttle.java` as Part of the `edu.colorado.simulations` Package

### Implementation

```
// File: Throttle.java from the package edu.colorado.simulations  
// Documentation is in Figure 2.1 on page 49 or from the Throttle link in  
// http://www.cs.colorado.edu/~main/docs/.
```

```
package edu.colorado.simulations; ← the package declaration
```

(continued)

**62 Chapter 2 / Java Classes and Information Hiding**

(FIGURE 2.2 continued)

```
public class Throttle
{
    private int top;      // The topmost position of the throttle
    private int position; // The current position of the throttle

    public Throttle(int size)
    {
        if (size <= 0)
            throw new IllegalArgumentException("Size <= 0: " + size);
        top = size;
        // No assignment needed for position -- it gets the default value of zero.
    }

    public double getFlow( )
    {
        return (double) position / (double) top;
    }

    public boolean isOn( )
    {
        return (getFlow( ) > 0);
    }

    public void shift(int amount)
    {
        if (amount > top - position)
            // Adding amount would put the position above the top.
            position = top;
        else if (position + amount < 0)
            // Adding amount would put the position below zero.
            position = 0;
        else
            // Adding amount puts position in the range [0 ... top].
            position += amount;
    }

    public void shutOff( )
    {
        position = 0;
    }
}
```

---

## The Import Statement to Use a Package

Once a package is set up and in the correct directory, the package's .java files can be compiled to create the various .class files. Then any other code you write can use part or all of the package. To use another package, a .java file places an import statement after its own package statement but before anything else. An **import statement for an entire package** has the keyword `import` followed by the package name plus “`*`” and a semicolon. For example, we can import the entire `edu.colorado.simulations` package with the import statement:

```
import edu.colorado.simulations.*;
```

*a program can use an entire package or just parts of a package*

If only a few classes from a package are needed, then each class can be imported separately. For example, this statement imports only the `Throttle` class from the `edu.colorado.simulations` package:

```
import edu.colorado.simulations.Throttle;
```

After this import statement, the `Throttle` class can be used. For example, a program can declare a variable:

```
Throttle control;
```

A sample program using our throttle appears in Figure 2.3. The program creates a new throttle, shifts the throttle fully on, and then steps the throttle back down to the shutoff position.

## The JCL Packages

The Java language comes with many useful packages called the **Java Class Libraries (JCL)**. Any programmer can use various parts of the JCL by including an appropriate import statement. In fact, one of the packages, `java.lang`, is so useful that it is automatically imported into every Java program. Some parts of the JCL are described in Section 2.5 and Appendix D.

## More about Public, Private, and Package Access

As you have seen, the `Throttle` class uses private instance variables (to keep track of the current status of a throttle) and public methods (to access and manipulate a throttle). The keywords `public` and `private` are called the **access modifiers** because they control access to the class members.

What happens if you declare a member with no access modifier—neither `public` nor `private`? In this case, the member can be accessed only by other classes in the same package. This kind of access is called **default access** (because there is no explicit access modifier); some programmers call it **package access**,

**64 Chapter 2 / Java Classes and Information Hiding****FIGURE 2.3** Implementation of the Throttle Demonstration Program with an Import StatementJava Application Program

```
// FILE: ThrottleDemonstration.java
// This small demonstration program shows how to use the Throttle class
// from the edu.colorado.simulations package.

import edu.colorado.simulations.Throttle; ← the import
public class ThrottleDemonstration
{
    public static void main(String[ ] args)
    {
        final int SIZE = 8; // The size of the demonstration Throttle

        Throttle small = new Throttle(SIZE);

        System.out.println("I am now shifting a Throttle fully on, and then I");
        System.out.println("will shift it back to the shutoff position.");

        small.shift(SIZE);
        while (small.isOn( ))
        {
            System.out.printf("The flow is now %5.3f\n", small.getFlow( )); ← the printf method from page 14
            small.shift(-1);
        }

        System.out.println("The flow is now off");
    }
}
```

Output from the Application

```
I am now shifting a Throttle fully on, and then I
will shift it back to the shutoff position.
The flow is now 1.000
The flow is now 0.875
The flow is now 0.750
The flow is now 0.625
The flow is now 0.500
The flow is now 0.375
The flow is now 0.250
The flow is now 0.125
The flow is now off
```

which is a nice descriptive name. We won't use package access much, because we prefer the pattern of private instance variables with public methods.

One other kind of access—protected access—will be discussed later when we cover derived classes and inheritance.

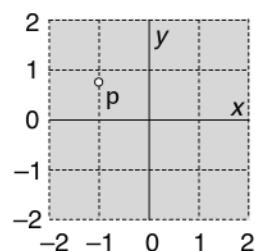
### Self-Test Exercises for Section 2.3

17. Suppose you are writing a package of classes for a company that has the Internet domain knafn.com. The classes in the package perform various statistical functions. Select a good name for the package.
18. Describe the directory structure that must be set up for the files of the package in the preceding question.
19. Write the package declaration that will appear at the top of every Java file for the package of the previous two questions.
20. Write the import statement that must be present to use the package from the previous questions.
21. Suppose you need only one class (called *Averager*) from the package of the previous questions. Rewrite the import statement to import only this one package.
22. What import statement is needed to use the `java.lang` package?
23. Describe public access, private access, and package access. What keywords are needed to obtain each kind of access for a method?

## 2.4 PARAMETERS, EQUALS METHODS, AND CLONES

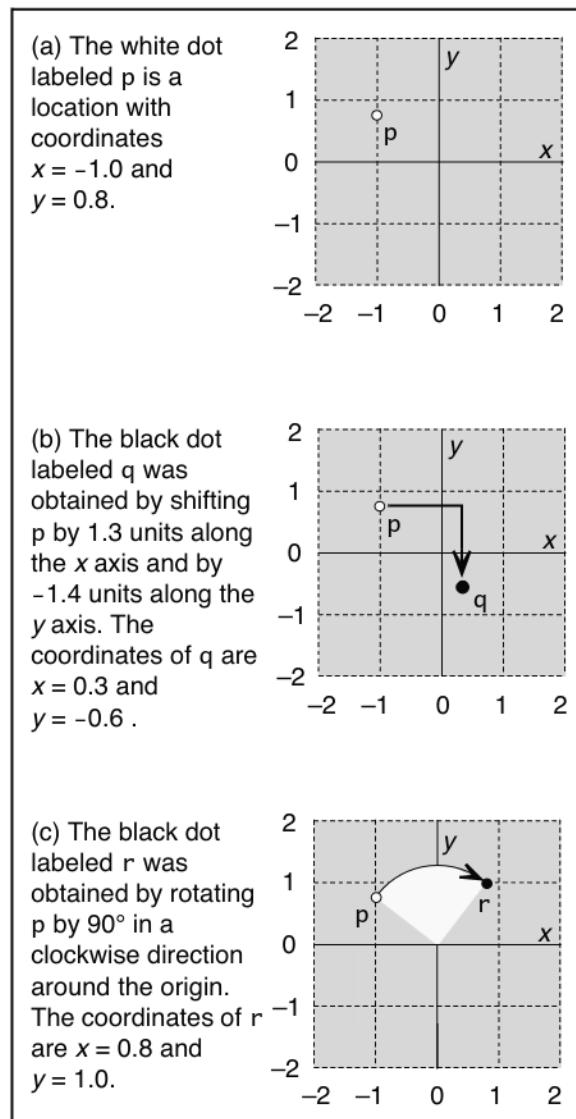
Every programmer requires an unshakable understanding of methods and their parameters. This section illustrates these issues and other issues that arise in Java, such as how to test whether two objects are equal to each other and how to make a copy of an object. The examples use a new class called `Location`, which will be placed in a package called `edu.colorado.geometry`.

The purpose of a `Location` object is to store the coordinates of a single point on a plane, as in the picture shown here. The location `p` in the picture lies at coordinates  $x = -1.0$  and  $y = 0.8$ . For future reference, you should know that Java has a similar class called `Point` in the `java.awt` package. But Java's `Point` class is limited to integer coordinates and used primarily to describe points on a computer's screen. I thought about using the same name, `Point`, for the example class of this section, but I decided against it because a program might want to use both classes. It's not legal to import two different classes with the same names (though you can use a full type name such as `java.awt.Point` without an import statement).



## 66 Chapter 2 / Java Classes and Information Hiding

**FIGURE 2.4** Three Locations in a Plane



### The Location Class

Figure 2.4 shows several sample locations. We'll use these sample locations to describe the `Location` constructor and methods.

- There is a constructor to initialize a location. The constructor's parameters provide the initial coordinates. For example, the location `p` in Figure 2.4(a) can be constructed with this statement:  
`Location p = new Location(-1, 0.8);`
- There is a modification method to shift a location by given amounts along the  $x$  and  $y$  axes, as shown in Figure 2.4(b).
- There is a modification method to rotate a location by  $90^\circ$  in a clockwise direction around the origin, as shown in Figure 2.4(c).
- There are two accessor methods that allow us to retrieve the current  $x$  and  $y$  coordinates of a location.
- There are a couple of methods to perform computations such as the distance between two locations. These are *static* methods—we'll discuss the importance of the *static* property in a moment.
- There are three methods called `clone`, `equals`, and `toString`. These methods have special importance for Java classes. The `clone` method allows a programmer to make an exact copy of an object. The `equals` method tests whether two different objects are identical. The `toString` method generates a string that represents an object. Special considerations for implementing these three methods are discussed next.

The `Location` class is small, yet it forms the basis for an actual data type that is used in graphics programs and other applications. All the methods and the constructor are listed in the specification of Figure 2.5. The figure also shows one way to implement the class. After you've looked through the figure, we'll discuss that implementation.

**FIGURE 2.5** Specification and Implementation for the Location Class

### Class Location

- ❖ **public class Location from the package edu.colorado.geometry**  
A Location object keeps track of a location on a two-dimensional plane.

#### Specification

◆ **Constructor for the Location**

```
public Location(double xInitial, double yInitial)
```

Construct a Location with specified coordinates.

**Parameters:**

xInitial – the initial x coordinate of this Location  
yInitial – the initial y coordinate of this Location

**Postcondition:**

This Location has been initialized at the given coordinates.

◆ **clone**

```
public Location clone()
```

Generate a copy of this Location.

**Returns:**

The return value is a copy of this Location. Subsequent changes to the copy will not affect the original, nor vice versa. Note that the return value must be typecast to a Location before it can be used.

◆ **distance**

```
public static double distance(Location p1, Location p2)
```

Compute the distance between two Locations.

**Parameters:**

p1 – the first Location  
p2 – the second Location

**Returns:**

the distance between p1 and p2

**Note:**

The answer is Double.POSITIVE\_INFINITY if the distance calculation overflows. The answer is Double.NaN if either Location is null.

(continued)

**68 Chapter 2 / Java Classes and Information Hiding**

(FIGURE 2.5 continued)

**◆ equals**

```
public boolean equals(Object obj)  
Compare this Location to another object for equality.
```

**Parameter:**

obj – an object with which this Location is compared

**Returns:**

A return value of true indicates that obj refers to a Location object with the same value as this Location. Otherwise, the return value is false.

**Note:**

If obj is null or is not a Location object, then the answer is false.

**◆ getX and getY**

```
public double getX()      –and–      public double getY()  
Get the x or y coordinate of this Location.
```

**Returns:**

the x or y coordinate of this Location

**◆ midpoint**

```
public static Location midpoint(Location p1, Location p2)  
Generates and returns a Location halfway between two others.
```

**Parameters:**

p1 – the first Location  
p2 – the second Location

**Returns:**

a Location that is halfway between p1 and p2

**Note:**

The answer is null if either p1 or p2 is null.

**◆ rotate90**

```
public void rotate90()  
Rotate this Location 90° in a clockwise direction.
```

**Postcondition:**

This Location has been rotated clockwise 90° around the origin.

**◆ shift**

```
public void shift(double xAmount, double yAmount)  
Move this Location by given amounts along the x and y axes.
```

**Postcondition:**

This Location has been moved by the given amounts along the two axes.

**Note:**

The shift may cause a coordinate to go above Double.MAX\_VALUE or below -Double.MAX\_VALUE. In these cases, subsequent calls of getX or getY will return Double.POSITIVE\_INFINITY or Double.NEGATIVE\_INFINITY.

(continued)

(FIGURE 2.5 continued)

♦ **toString**

```
public String toString( )
Generate a string representation of this Location.  
Returns:  
    a string representation of this Location
```

### Implementation

```
// File: Location.java from the package edu.colorado.geometry
// Documentation is available on pages 67-68 or from the Location link in
// http://www.cs.colorado.edu/~main/docs/.
```

```
package edu.colorado.geometry;
public class Location implements Cloneable
{
    private double x; // The x coordinate of the Location
    private double y; // The y coordinate of the Location

    public Location(double xInitial, double yInitial)
    {
        x = xInitial;
        y = yInitial;
    }

    public Location clone( )
    { // Clone a Location object.
        Location answer;

        try
        {
            answer = (Location) super.clone( );
        }
        catch (CloneNotSupportedException e)
        { // This exception should not occur. But if it does, it would indicate a programming
           // error that made super.clone unavailable. The most common cause would be
           // forgetting the "implements Cloneable" clause at the start of the class.
            throw new RuntimeException
                ("This class does not implement Cloneable.");
        }

        return answer;
    }
}
```

*the meaning of  
“implements Cloneable”  
and the clone method are  
discussed on page 81*

(continued)

**70 Chapter 2 / Java Classes and Information Hiding**

(FIGURE 2.5 continued)

```
public static double distance(Location p1, Location p2)
{
    double a, b, c_squared;
    // Check whether one of the Locations is null.
    if ((p1 == null) || (p2 == null))
        return Double.NaN; // the Java constant Double.NaN is discussed on page 74
    // Calculate differences in x and y coordinates.
    a = p1.x - p2.x;
    b = p1.y - p2.y;

    // Use Pythagorean Theorem to calculate the square of the distance
    // between the Locations.
    c_squared = a*a + b*b;

    return Math.sqrt(c_squared);
}

public boolean equals(Object obj)
{
    if (obj instanceof Location)
    {
        Location candidate = (Location) obj;
        return (candidate.x == x) && (candidate.y == y);
    }
    else
        return false;
}

public double getX()
{
    return x;
}

public double getY()
{
    return y;
}
```

*the meaning of a static method is discussed on page 72*

*the equals method is discussed on page 77*

(continued)

(FIGURE 2.5 continued)

```
public static Location midpoint(Location p1, Location p2)
{
    double xMid, yMid;

    // Check whether one of the Locations is null.
    if ((p1 == null) || (p2 == null))
        return null;

    // Compute the x and y midpoints.
    xMid = (p1.x/2) + (p2.x/2);
    yMid = (p1.y/2) + (p2.y/2);

    // Create a new Location and return it.
    Location answer = new Location(xMid, yMid);
    return answer;
}

public void rotate90( )
{
    double xNew;
    double yNew;

    // For a 90-degree clockwise rotation, the new x is the original y
    // and the new y is -1 times the original x.
    xNew = y;
    yNew = -x;
    x = xNew;
    y = yNew;
}

public void shift(double xAmount, double yAmount)
{
    x += xAmount;
    y += yAmount;
}

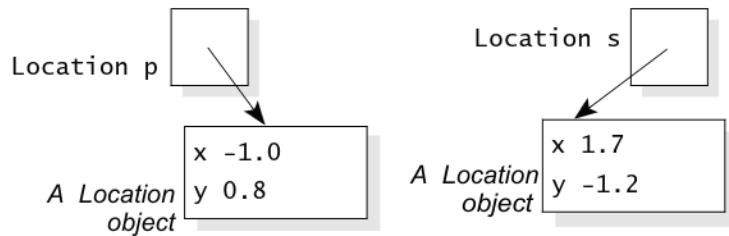
public String toString( )
{
    return "(x=" + x + " y=" + y + ")";
}
```

---



## Parameters That Are Objects

What happens when `Location.distance(p, s)` is activated? For example, suppose we have the two declarations shown previously for `p` and `s`. After these declarations, we have these two separate locations:



Now we can activate the method `Location.distance(p, s)`, which has an implementation that starts like this:

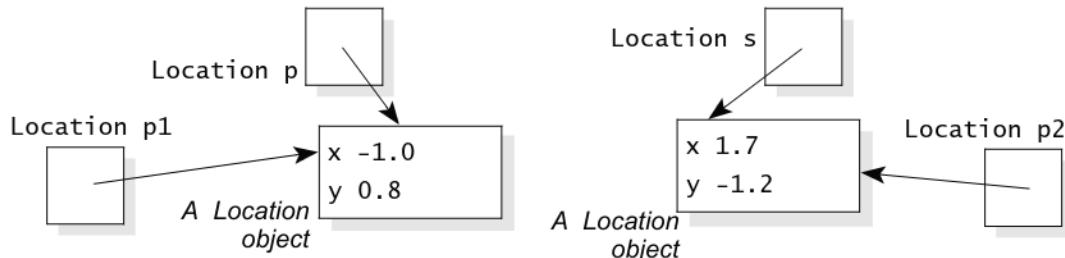
```
public static distance(Location p1, Location p2)
{
    ...
}
```

The names used within the method (`p1` and `p2`) are usually called **parameters** to distinguish them from the values that are passed in (`p` and `s`). On the other hand, the values that are passed in (`p` and `s`) are called the **arguments**. The first step of any method activation is to use the *arguments* to provide initial values for the *parameters*. Here's the important fact you need to know about parameters that are objects:

"parameters" versus "arguments"

When a parameter is an object, such as a `Location`, then the parameter is initialized so that it refers to the same object that the actual argument refers to.

In our example, `Location.distance(p, s)`, the parameters `p1` and `p2` are initialized to refer to the two locations we created, like this:



## 74 Chapter 2 / Java Classes and Information Hiding

*be careful about changing the value of a parameter*

Within the body of the `distance` method we can access `p1` and `p2`. For example, we can access `p1.x` to obtain the `x` coordinate of the first parameter. This kind of access is okay in a static method. The only forbidden expression is a direct `x` or `y` (without a qualifier such as `p1`).

Some care is needed in accessing a parameter that is an object. For instance, any change to `p1.x` will affect the actual argument `p.x`. We don't want the `distance` method to make changes to its arguments; it should just compute the distance between the two locations and return the answer. This computation occurs in the implementation of `distance` on page 70.

The implementation also handles a couple of special cases. One special case is when an argument is null. In this case, the corresponding parameter will be initialized as null, and the `distance` method executes this code:

```
// Check whether one of the Locations is null.  
if ((p1 == null) || (p2 == null))  
    return Double.NaN;
```

*the “not-a-number” constant*

If either parameter is null, then the method returns a Java constant named `Double.NaN`. This is a constant that a program uses to indicate that a double value is “not a number.”

*the “infinity” constant*

Another special case for the `distance` method is the possibility of a numerical overflow. The numbers obtained during a computation may go above the largest double number or below the smallest double number. These numbers are pretty large, but the possibility of overflow still exists. When an arithmetic expression with double numbers goes beyond the legal range, Java assigns a special constant to the answer. The constant is named `Double.POSITIVE_INFINITY` if it is too large (above about  $1.7^{308}$ ), and it is named `Double.NEGATIVE_INFINITY` if it is too small (below about  $-1.7^{308}$ ). Of course, these constants are not really “infinity.” They are merely indications to the programmer that a computation has overflowed. In the `distance` method, we indicate the possibility of overflow with the following comment:

**Note:**

The answer is `Double.POSITIVE_INFINITY` if the distance calculation overflows. The answer is `Double.NaN` if either Location is null.

### Methods May Access Private Instance Variables of Objects in Their Own Class

You may have noticed that the `distance` method used the `x` and `y` instance variables of `p1` and `p2` directly, for example:

```
a = p1.x - p2.x;
```

Is this allowed? After all, `x` and `y` are private instance variables. The answer is yes: A method may access private instance variables of an object as long as the method is declared as part of the same class as the object. In this example, `distance` is a member function of the `Location` class, and both `p1` and `p2` are `Location` objects. Therefore, the `distance` method may access the private instance variables of `p1` and `p2`.

### The Return Value of a Method May Be an Object

The return value of a method may also be an object, such as a `Location` object. For example, the `Location` class has this static method that creates and returns a new location that is halfway between two other locations. The method's specification and implementation are:

◆ **midpoint**

```
public static Location midpoint(Location p1, Location p2)  
Generates and returns a Location halfway between two others.
```

**Parameters:**

`p1` – the first `Location`  
`p2` – the second `Location`

**Returns:**

a `Location` that is halfway between `p1` and `p2`

**Note:**

The answer is null if either `Location` is null.

```
public static Location midpoint(Location p1, Location p2)  
{  
    double xMid, yMid;  
  
    // Check whether one of the Locations is null.  
    if ((p1 == null) || (p2 == null))  
        return null;  
  
    // Compute the x and y midpoints.  
    xMid = (p1.x/2) + (p2.x/2);  
    yMid = (p1.y/2) + (p2.y/2);  
  
    // Create a new Location and return it.  
    Location answer = new Location(xMid, yMid);  
    return answer;  
}
```

The method creates a new location using the local variable `answer` and then returns this location. Often the return value is stored in a local variable such as `answer`, but not always. For example, we could have eliminated `answer` by combining the last two statements in our implementation to a single statement:

## 76 Chapter 2 / Java Classes and Information Hiding

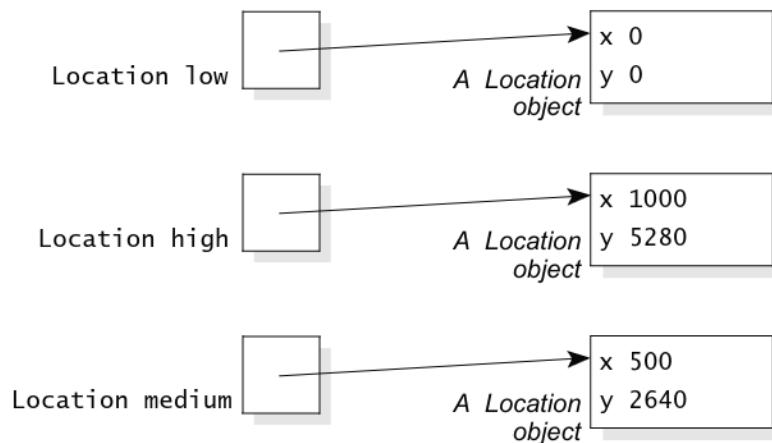
```
return new Location(xMid, yMid);
```

Either way—with or without the local variable—is fine.

Here's an example to show how the static `midpoint` method is used. The method creates two locations and then computes their midpoint:

```
Location low = new Location(0, 0);
Location high = new Location(1000, 5280);
Location medium = Location.midpoint(low, high);
```

In this example, the answer from the `midpoint` method is stored in a variable called `medium`. After the three statements, we have three locations:



### PROGRAMMING TIP

#### How to Choose the Names of Methods

**Accessor methods:** The name of a boolean accessor method will usually begin with "is" followed by an adjective (such as "isOn"). Methods that convert to another kind of data start with "to" (such as "toString"). Other accessor methods start with "get" or some other verb followed by a noun that describes the return value (such as "getFlow").

**Modification methods:** A modification method can be named by a descriptive verb (such as "shift") or a short verb phrase (such as "shutOff").

**Static methods that return a value:** Try to use a noun that describes the return object (such as "distance" or "midpoint").

Rules like these make it easier to determine the purpose of a method.

## Java's Object Type

One of the `Location` methods is an accessor method called `equals` with this heading:

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

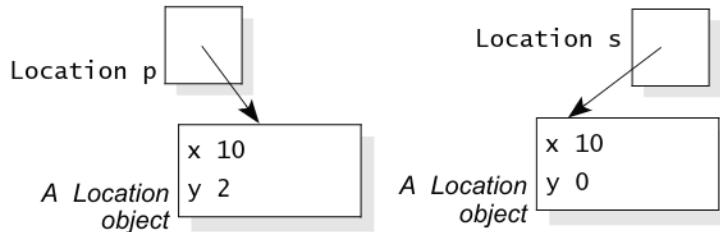
An accessor method with this name has a special meaning in Java. Before we discuss that meaning, you need to know a bit about the parameter type `Object`. In Java, `Object` is a kind of “super data type” that encompasses all data except the eight primitive types. So a primitive variable (`byte`, `short`, `int`, `long`, `char`, `float`, `double`, or `boolean`) is *not* an `Object`, but everything else is. A `String` is an `Object`, a `Location` is an `Object`, and even an array is an `Object`.

## Using and Implementing an equals Method

As your programming expertise progresses, you'll learn a lot about Java's `Object` type, but to start, you need just a few common patterns that use `Object`. For example, many classes implement an `equals` method with the heading we have seen. An `equals` method has one argument: an `Object` called `obj`. The method should return `true` if `obj` has the same value as the object that activated the method. Otherwise, the method returns `false`. Here is an example to show how the `equals` method works for the `Location` class:

```
Location p = new Location(10, 2); // Declare p at coordinates (10,2)
Location s = new Location(10, 0); // Declare s at coordinates (10,0)
```

After these two declarations, we have two separate locations:



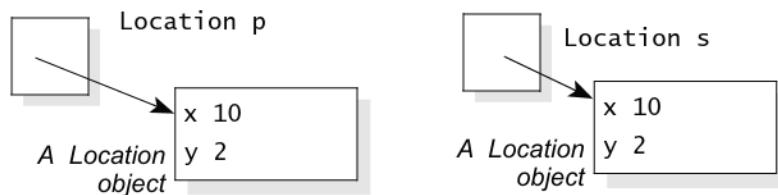
In this example, `p` and `s` refer to two separate objects with different values (their `y` coordinates are different), so both `p.equals(s)` and `s.equals(p)` are `false`.

Here's a slightly different example:

```
Location p = new Location(10, 2); // Declare p at coordinates (10,2)
Location s = new Location(10, 0); // Declare s at coordinates (10,0)
s.shift(0, 2); // Move s to (10,2)
```

## 78 Chapter 2 / Java Classes and Information Hiding

We have the same two declarations, but afterward we shift the *y* coordinate of *s* so that the two separate locations have identical values, like this:



Now *p* and *s* refer to identical locations, so both *p.equals(s)* and *s.equals(p)* are true. However, the test (*p == s*) is still false. Remember that (*p == s*) returns true only if *p* and *s* refer to the exact same location (as opposed to two separate locations that happen to contain identical values).

The argument to the *equals* method can be any object, not just a location. For example, we can try to compare a location with a string, like this:

```
Location p = new Location(10, 2);
System.out.println(p.equals("10, 2")); // Prints false.
```

This example prints false; a *Location* object is not equal to the string "10, 2", even if they are similar. You can also test to see whether a location is equal to null:

```
Location p = new Location(10, 2);
System.out.println(p.equals(null)); // Prints false.
```

The location is not null, so the result of *p.equals(null)* is false. Be careful with the last example: The argument to *p.equals* may be null and the answer will be false. However, when *p* itself is null, it is a programming error to activate any method of *p*. Trying to activate *p.equals* when *p* is null results in a *NullPointerException* (see page 55).

implementing an equals method

Now you know how to use an *equals* method. How do you write an *equals* method so that it returns true when its argument has the same value as the object that activates the method? A typical implementation follows an outline that is used for the *equals* method of the *Location* class, as shown here:

```
public boolean equals(Object obj)
{
    if (obj is actually a Location)
    {
        Figure out whether the location that obj refers to has the same
        value as the location that activated this method. Return true if
        they are the same; otherwise, return false.
    }
    else
        return false;
}
```

The method starts by determining whether `obj` actually refers to a `Location` object. In pseudocode, we wrote this as “`obj` is actually a `Location`.” In Java, this is accomplished with the test `(obj instanceof Location)`. This test uses the keyword `instanceof`, which is a boolean operator. On the left of the operator is a variable such as `obj`. On the right of the operator is a class name such as `Location`.

*the instanceof operator*

The test returns `true` if it is valid to convert the object (`obj`) to the given data type (`Location`). In our example, suppose that `obj` does not refer to a valid `Location`. It might be some other type of object, or perhaps it is simply null. In either case, we go to the `else`-statement and return `false`.

On the other hand, suppose that `(obj instanceof Location)` is true, so the code enters the first part of the `if`-statement. Then `obj` does refer to a `Location` object. We need to determine whether the `x` and `y` coordinates of `obj` are the same as the location that activated the method. Unfortunately, we can’t just look at `obj.x` and `obj.y`, because the compiler thinks of `obj` as a bare object with no `x` and `y` instance variables. The solution is an expression `(Location) obj`. This expression is called a *typecast*, as if we were pouring `obj` into a casting mold that creates a `Location` object. The expression can be used to initialize a `Location` reference variable, like this:

```
Location candidate = (Location) obj;
```

The **typecast**, on the right side of the declaration, consists of the new data type (`Location`) in parentheses, followed by the reference variable that is being cast. After this declaration, `candidate` is a reference variable that refers to the same object that `obj` refers to. However, the compiler *does* know that `candidate` refers to a `Location` object, so we can look at `candidate.x` and `candidate.y` to see if they are the same as the `x` and `y` coordinates of the object that activated the `equals` method. The complete implementation looks like this:

```
public boolean equals(Object obj)
{
    if (obj instanceof Location)
    {
        Location candidate = (Location) obj;
        return (candidate.x == x) && (candidate.y == y);
    }
    else
        return false;
}
```

*typecasts*

The implementation has the return statement:

```
return (candidate.x == x) && (candidate.y == y);
```

The boolean expression in this return statement is `true` if `candidate.x` and `candidate.y` are the same as the instance variables `x` and `y`. As with any method, these instance variables come from the object that activated the method. For future reference, the details of using a typecast are given in Figure 2.7.

## PITFALL

### CLASSCASTEXCEPTION

Suppose you have a variable such as `obj`, which is an `Object`. You can try a typecast to use the object as if it were another type. For example, we used the typecast `Location candidate = (Location) obj`.

What happens when `obj` doesn't actually refer to a `Location` object? The result is a runtime exception called `ClassCastException`. To avoid this, you must ensure that a typecast is valid before trying to execute the cast. For example, the `instanceof` operator can validate the actual type of an object before a typecast.

### Every Class Has an equals Method

You may write a class without an `equals` method, but Java automatically provides an `equals` method anyway. The `equals` method that Java provides is actually taken from the `Object` class, and it works exactly like the `==` operator. In other words, it returns `true` only when the two objects are the exact same.

FIGURE 2.7 Typecasts

#### A Simple Pattern for Typecasting an Object

A common situation in Java programming is a variable or other expression that is an `Object`, but the program needs to treat the `Object` as a specific data type such as `Location`. The problem is that when a variable is declared as an `Object`, that variable cannot be used immediately as if it were a `Location` (or some other type). For example, consider the parameter `obj` in the `equals` method of the `Location` class:

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

Within the implementation of the `equals` method, we need to treat `obj` as a `Location` rather than a mere `Object`. The solution has two parts: (1) Check that `obj` does indeed refer to a valid `Location`, and (2) declare a new variable of type `Location` and initialize this new variable to refer to the same object that `obj` refers to, like this:

```
public boolean equals(Object obj)           ← The parameter, obj, is an Object.  
{                                         ← Use the instanceof operator to check that  
    if (obj instanceof Location)          ← obj is a valid Location.  
    {  
        Location candidate = (Location) obj;  
        ...                                ← After this declaration, candidate refers to  
                                         ← the Location object that obj also refers to.
```

The expression `(Location) obj`, used in the declaration of `candidate`, is a typecast to tell the compiler that `obj` may be used as a `Location`.

object—but it returns `false` for two separate objects that happen to have the same values for their instance variables.

## Using and Implementing a `clone` Method

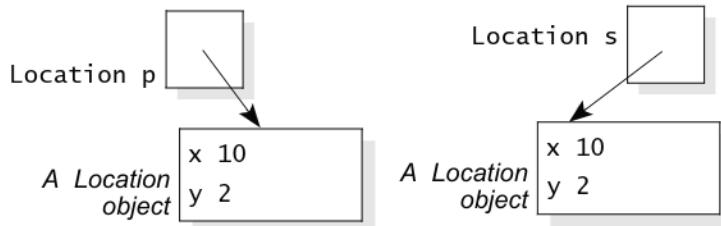
Another feature of our `Location` class is a method with this heading:

```
public Location clone()
```

The purpose of a `clone` method is to create a copy of an object. The copy is separate from the original so that subsequent changes to the copy won't alter the original, nor will subsequent changes to the original alter the copy. Here's an example of using the `clone` method for the `Location` class:

```
Location p = new Location(10, 2); // Declare p at (10,2)
Location s = p.clone();           // Initialize as a copy of p
```

The expression `p.clone()` activates the `clone` method for `p`. The method creates and returns an exact copy of `p`, which we use to initialize the new location `s`. After these two declarations, we have two separate locations, as shown in this picture:



As you can see, `s` and `p` have the same values for their instance variables, but the two objects are separate. Changes to `p` will not affect `s`, nor will changes to `s` affect `p`.

### PITFALL



#### OLDER JAVA CODE REQUIRES A TYPECAST FOR CLONES

Prior to Java 5.0, the data type of the return value of the `clone` method was always an `Object` and not a specific type such as `Location`. Because of this requirement, the `clone` return value could not be used directly in older versions of Java. For example, we could not write a declaration:

```
Location s = p.clone();
```

Instead, we must apply a typecast to the `clone` return value, converting it to a `Location` before we use it to initialize the new variable `s`, like this:

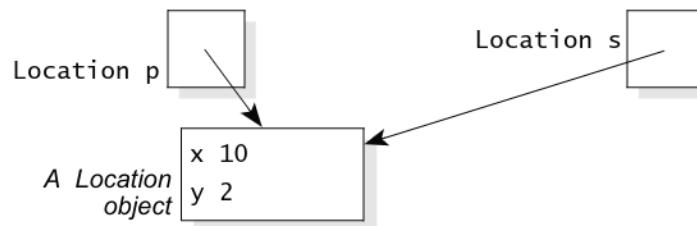
## 82 Chapter 2 / Java Classes and Information Hiding

```
// Typecast required for older Java compilers:  
|     Location s = (Location) p.clone();
```

Cloning is considerably different from using an assignment statement. For example, consider this code that does not make a clone:

```
Location p = new Location(10, 2); // Declare p at coordinates (10,2).  
Location s = p; // Declare s and make it refer  
// to the same object that p  
// refers to.
```

After these two declarations, we have just one location, and both variables refer to this location:



This is the situation with an ordinary assignment. Subsequent changes to the object that p refers to will affect the object that s refers to because there is only one object.

implementing a  
clone method

You now know how to use a `clone` method. How do you implement such a method? You should follow a three-step pattern:

**1. Modify the Class Head.** You must add the words “`implements Cloneable`” in the class head, as shown here for the `Location` class:

```
public class Location implements Cloneable
```

The modification informs the Java compiler that you plan to implement certain features that are specified elsewhere in a format called an *interface*. The full meaning of interfaces will be discussed in Chapter 5. At the moment, it is enough to know that `implements Cloneable` is necessary when you implement a `clone` method.

**2. Use `super.clone` to Make a Copy.** The implementation of a `clone` method should begin by making a copy of the object that activated the method. The best way to make the copy is to start with this pattern from the `Location` class:

```
public Location clone()
{ // Clone a Location object.
    Location answer;
    try
    {
        answer = (Location) super.clone();
    }
    catch (CloneNotSupportedException e)
    {
        throw new RuntimeException
            ("This class does not implement Cloneable.");
    }
    ...
}
```

In an actual implementation, you would use the name of your own class (rather than `Location`), but otherwise you should follow this pattern exactly.

It's useful to know what's happening in this pattern. The pattern starts by declaring a local `Location` variable called `answer`. We then have this block:

```
try
{
    answer = (Location) super.clone();
}
```

This is an example of a *try block*. If you plan on extensive use of Java exceptions, you should read all about try blocks in Appendix C. But for your first try block, all you need to know is that the code in the try block is executed, and the try block will be able to handle some of the possible exceptions that may arise in the code. In this example, the try block has just one assignment statement: `answer = (Location) super.clone()`. The right side of the assignment activates a method called `super.clone()`. This is actually the `clone` method from Java's `Object` type. It checks that the `Location` class specifies that it "implements `Cloneable`" and then correctly makes a copy of the location, assigning the result to the local variable `answer`.

After the try block is a sequence of one or more *catch blocks*. Each catch block can catch and handle an exception that may arise in the try block. Our example has just one catch block:

```
catch (CloneNotSupportedException e)
{
    throw new RuntimeException
        ("This class does not implement Cloneable.");
}
```

This catch block will handle a `CloneNotSupportedException`. This exception is thrown by the `clone` method from Java's `Object` type when a programmer tries to call `super.clone()` without including the `implements Cloneable`



**PROGRAMMING TIP****ALWAYS USE SUPER.CLONE FOR YOUR CLONE METHODS**

Perhaps you thought of a simpler way to create a clone. Instead of using `super.clone` and the try/catch blocks, you could write this code:

```
Location answer = new Location(x, y);
return answer;
```

You could combine these into one statement: `return new Location(x, y)`. This creates and returns a new location, using the instance variables `x` and `y` to initialize the new location. These instance variables come from the location that activated the `clone` method, so `answer` will indeed be a copy of that location. This is a nice direct approach, but the direct approach will encounter problems when we start building new classes that are based on existing classes (see Chapter 13). Therefore, it is better to use the pattern with `super.clone` and a try/catch block.

**PROGRAMMING TIP****WHEN TO THROW A RUNTIME EXCEPTION**

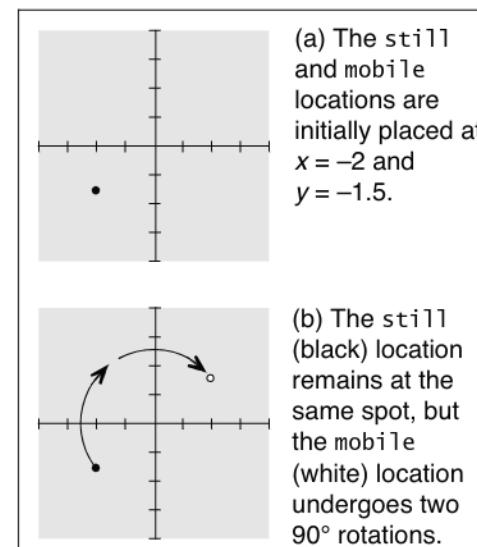
A `RuntimeException` is thrown to indicate a programming error. For example, the `clone` method from Java's `Object` type is not supposed to be called by an object unless that object's class has implemented the `Cloneable` interface. If we detect that the exception has been thrown by the `Object clone` method, then the programmer probably forgot to include the "implements `Cloneable`" clause.

When you throw a `RuntimeException`, include a message with your best guess about the programming error.

**A Demonstration Program for the Location Class**

As one last example, let's look at a program that creates two locations called `still` and `mobile` (see Figure 2.8). Both are initially placed at  $x = -2$  and  $y = -1.5$ , as shown in Figure 2.8(a). To be more precise, the `still` location is placed at this spot, and then `mobile` is initialized as a clone of the `still` location. Because the `mobile` location is a clone, later changes to one location will not affect the other.

The program prints some information about both locations, and then the `mobile` location undergoes two  $90^\circ$  rotations, as shown in Figure 2.8(b). The information about the locations is then printed a second time.

**FIGURE 2.8** Two  $90^\circ$  Rotations

## 86 Chapter 2 / Java Classes and Information Hiding

The complete program is shown in Figure 2.9 on page 87. Pay particular attention to the `specifiedRotation` method, which illustrates some important principles about what happens when a parameter is changed within a method. We'll look at those principles in a moment, but first let's look at the complete output from the program:

```
The still location is at: (x=-2.0 y=-1.5)
The mobile location is at: (x=-2.0 y=-1.5)
Distance between them: 0.0
These two locations have equal coordinates.
```

```
I will rotate one location by two 90-degree turns.
The still location is at: (x=-2.0 y=-1.5)
The mobile location is at: (x=2.0 y=1.5)
Distance between them: 5.0
These two locations have different coordinates.
```

### What Happens When a Parameter Is Changed Within a Method?

Let's examine the program's `specifiedRotation` method to see exactly what happens when a parameter is changed within a method. Here is the method's implementation:

```
// Rotate a Location p by a number of 90-degree clockwise rotations.
public static void specifiedRotation(Location p, int n)
{
    while (n > 0)
    {
        p.rotate90();
        n--;
    }
}
```

The method rotates the location `p` by `n` 90° clockwise rotations.

In Java, a parameter that is a reference variable (such as the `Location p`) has different behavior from a parameter that is one of the eight primitive types (such as `int n`). Here is the difference:

- When a parameter is one of the eight primitive types, the actual argument provides an initial value for that parameter. To be more precise, the parameter is implemented as a local variable of the method, and the argument is used to initialize this variable. Changes that are made to the parameter *do not* affect the actual argument.
- When a parameter is a reference variable, the parameter is initialized so that it refers to the same object as the actual argument. Subsequent changes to this object *do* affect the actual argument's object.

**FIGURE 2.9** A Demonstration Program for the Location ClassJava Application Program

```
// FILE: LocationDemonstration.java
// This small demonstration program shows how to use the Location class
// from the edu.colorado.geometry package.

import edu.colorado.geometry.Location;

public class LocationDemonstration
{
    public static void main(String[ ] args)
    {
        final double STILL_X = -2.0;
        final double STILL_Y = -1.5;
        final int ROTATIONS = 2;

        Location still = new Location(STILL_X, STILL_Y);
        Location mobile = still.clone( );
        printData(still, mobile);

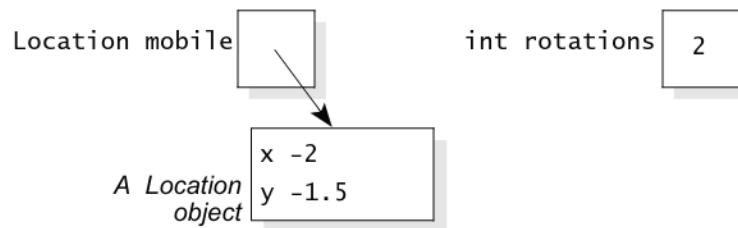
        System.out.println("I will rotate one location by two 90-degree turns.");
        specifiedRotation(mobile, ROTATIONS);
        printData(still, mobile);
    }

    // Rotate a Location p by a specified number of 90-degree clockwise rotations.
    public static void specifiedRotation(Location p, int n)
    {
        while (n > 0)
        {
            p.rotate90( );
            n--;
        }
    }

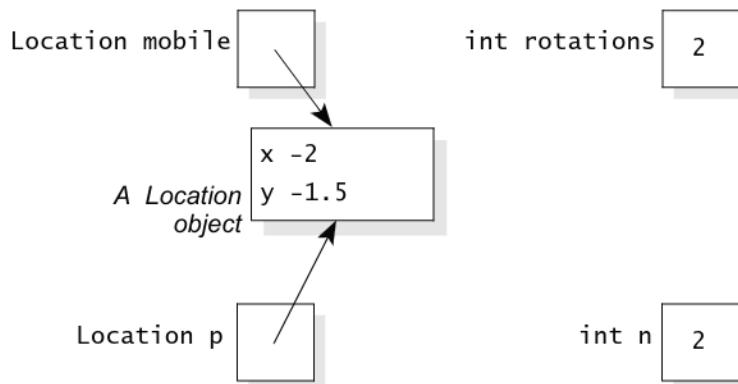
    // Print some information about two locations: s (a "still" location) and m (a "mobile" location).
    public static void printData(Location s, Location m)
    {
        System.out.println("The still location is at: " + s.toString( ));
        System.out.println("The mobile location is at: " + m.toString( ));
        System.out.println("Distance between them: " + Location.distance(s, m));
        if (s.equals(m))
            System.out.println("These two locations have equal coordinates.");
        else
            System.out.println("These two locations have different coordinates.");
        System.out.println( );
    }
}
```

**88** Chapter 2 / Java Classes and Information Hiding

For example, suppose that we have initialized a location called `mobile` at the coordinates  $x = -2$  and  $y = -1.5$ . Suppose we also have an integer variable called `rotations` with a value of 2:



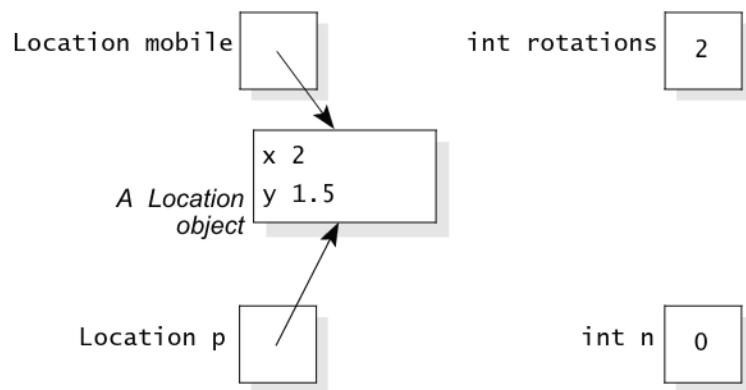
Now suppose the program activates `specifiedRotation(mobile, rotations)`. The method's first parameter, `p`, is initialized to refer to the same location that `mobile` refers to. The method's second parameter, `n`, is initialized with the value 2 (from the `rotations` argument). So, when the method begins its work, the situation looks like this:



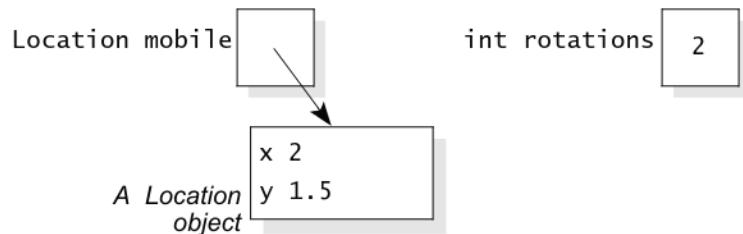
The method now executes its loop:

```
while (n > 0)
{
    p.rotate90( );
    n--;
}
```

The first iteration of the loop rotates the location by 90° and decreases `n` to 1. The second iteration does another rotation of the location and decreases `n` to 0. Now the loop ends with these variable values:



Notice the difference between the two kinds of parameters. The integer parameter `n` has changed to zero without affecting the actual argument `rotations`. On the other hand, rotating the location `p` has changed the object that `mobile` refers to. When the method returns, the parameters `p` and `n` disappear, leaving this situation:



#### Java Parameters

**The eight primitive types** (`byte`, `short`, `int`, `long`, `char`, `float`, `double`, or `boolean`): The parameter is initialized with the value of the argument. Subsequent changes to the parameter do not affect the argument.

**Reference variables:** When a parameter is a reference variable, the parameter is initialized so that it refers to the same object as the actual argument. Subsequent changes to this object do affect the actual argument's object.

#### Self-Test Exercises for Section 2.4

24. Write some code that declares two locations: one at the origin and the other at the coordinates  $x = 1$  and  $y = 1$ . Print the distance between the two locations, and then create a third location that is at the midpoint between the first two locations.

**90 Chapter 2 / Java Classes and Information Hiding**

25. The location's `distance` method is a static method. What effect does this have on how the method is used? What effect does this have on how the method is implemented?
26. What is the purpose of the Java constant `Double.NaN`?
27. What is the result when you add two `double` numbers and the answer is larger than the largest possible `double` number?
28. In the `midpoint` method, we used the expression `(p1.x/2) + (p2.x/2)`. Can you think of a reason why this expression is better than `(p1.x + p2.x)/2`?
29. Implement an `equals` method for the `Throttle` class from Section 2.1.
30. If you don't implement an `equals` method for a class, then Java automatically provides one. What does the automatic `equals` method do?
31. Implement a `clone` method for the `Throttle` class from Section 2.1.
32. When should a program throw a `RuntimeException`?
33. Suppose a method has an `int` parameter called `x`, and the body of the method changes `x` to zero. When the method is activated, what happens to the argument that corresponds to `x`?
34. Suppose a method has a `Location` parameter called `x`, and the body of the method activates `x.rotate90()`. When the method is activated, what happens to the argument that corresponds to `x`?

## 2.5 THE JAVA CLASS LIBRARIES

As a computer scientist, it's important for you to understand how to build and test your own classes, but frequently you'll find that a suitable class has already been built for you to use in an application. There's no need for you to write everything from scratch! In Java, a variety of container classes called the **Java Class Libraries** are available for all programs. This section provides an introduction to one of these classes: the `BigInteger` class.

Each `BigInteger` object holds information about an integer, but unlike an ordinary Java `int`, there is no limit to the size of a `BigInteger`. A `BigInteger` object could be a small integer (perhaps 0, 1, 2, or even a negative number), but it could be much larger than any Java `int`, such as the large number in this output:

```
The value of 42 factorial is  
14050061177528798985431426062445115699363840000000000.
```

The `BigInteger` data type is a class that has some interesting private instance variables to hold the information about a potentially large integer. But in order for you to *use* a `BigInteger` object in your program, you don't need to know those private details. All you need to know is the information that the designers of the `BigInteger` class provide about the public member functions, part of which is shown in Figure 2.10.

**FIGURE 2.10** Part of the Specification for the BigInteger Class

### Class BigInteger

❖ **public class BigInteger from the package java.math**

A BigInteger object provides an immutable arbitrary precision integer.

#### Partial Specification

(taken partly from <http://download.oracle.com/javase/6/docs/api/java/math/BigInteger.html>)

◆ **One of the Constructors for the BigInteger**

`public Location(String val)`

Translate the decimal String representation of a base 10 integer into a BigInteger.

**Parameter:**

`xInitial` – a string that contains a base 10 integer of any size

**Postcondition:**

This BigInteger has been initialized to the specified value from `val`.

◆ **add**

`public BigInteger add(BigInteger val)`

Add this BigInteger to another.

**Returns:**

The return value is this BigInteger + `val`.

◆ **equals**

`public boolean equals(Object obj)`

Compare this BigInteger to another object for equality.

**Parameter:**

`obj` – an object with which this BigInteger is compared

**Returns:**

A return value of `true` indicates that `obj` refers to a BigInteger object with the same value as this BigInteger. Otherwise, the return value is `false`.

**Note:**

If `obj` is null or is not a BigInteger object, then the return value is `false`.

◆ **multiply**

`public BigInteger multiply(BigInteger val)`

Multiply this BigInteger times another.

**Returns:**

The return value is this BigInteger \* `val`.

## 92 Chapter 2 / Java Classes and Information Hiding

Here's a short example that uses three `BigInteger` objects to compute and print the value of 42 factorial (i.e.,  $1 \times 2 \times 3 \times 4 \times \dots$  up to 42):

```
import java.math.BigInteger;

public class BigIntegerDemonstration
{
    public static void main(String[ ] args)
    {
        // Note: Sample must be >= 2 for this demonstration.
        BigInteger sample = new BigInteger("42");
        BigInteger answer = new BigInteger("1");
        BigInteger factor = new BigInteger("1");

        // Compute the factorial of sample:
        do
        {
            factor = factor.add(BigInteger.ONE);
            answer = answer.multiply(factor);
        } while (!factor.equals(sample));

        System.out.println("The value of " + sample);
        System.out.println(" factorial is " + answer + ".");
    }
}
```

In addition to the methods that were documented in Figure 2.10, the sample program also uses a constant, `BigInteger.ONE`, that is provided as part of the class.

## CHAPTER SUMMARY

- In Java, object-oriented programming (OOP) is supported by implementing *classes*. Each class defines a collection of data, called its *instance variables*. In addition, a class has the ability to include two other items: *constructors* and *methods*. Constructors are designed to provide initial values to the class's data; methods are designed to manipulate the data. Taken together, the instance variables, constructors, and methods of a class are called the *class members*.
- We generally use *private instance variables* and *public methods*. This approach supports information hiding by forbidding data components of a class to be directly accessed outside of the class.
- A new class can be implemented in a Java package that is provided to other programmers to use. The package includes documentation to tell programmers what the new class does without revealing the details of how the new class is implemented.

- A program uses a class by creating new objects of that class and activating these objects' methods through *reference variables*.
- When a method is activated, each of its parameters is initialized. If a parameter is one of the eight primitive types, then the parameter is initialized by the value of the argument, and subsequent changes to the parameter do not affect the actual argument. On the other hand, when a parameter is a reference variable, the parameter is initialized so that it refers to the same object as the actual argument. Subsequent changes to this object do affect the actual argument's object.
- Java programmers must understand how these items work for classes:
  - the assignment operator ( $x = y$ )
  - the equality test ( $x == y$ )
  - a `clone` method to create a copy of an object
  - an `equals` method to test whether two separate objects are equal to each other
- Java provides many prebuilt classes—the Java Class Libraries—for all programmers to use.

## Solutions to Self-Test Exercises



1. We have used *private* instance variables, *public* constructors, and *public* methods.
2. Accessor methods are not usually void, because they use the return value to provide some information about the current state of the object. Modification methods are often void because they change the object but do not return information.
3. In this solution, the assignment to `position` is not really needed, since `position` will be given its default value of zero before the constructor executes. However, including the assignment makes it clear that we intended for `position` to start at zero:

```
public Throttle( )
{
    top = 1;
    position = 0;
}
```
4. Notice that our solution (shown at the top of the next column) has the preconditions that `0 < size` and `0 <= initial <= size`.

```
public Throttle(int size, int initial)
{
    if (size <= 0)
        throw new
            IllegalArgumentException
            ("Size <= 0:" + size);
    if (initial < 0)
        throw new
            IllegalArgumentException
            ("Initial < 0:" + initial);
    if (initial > size)
        throw new
            IllegalArgumentException
            ("Initial too big:" + initial);
    top = size;
    position = initial;
}
```

5. An accessor method makes no changes to the object's instance variables.
6. The method implementation is:

```
public boolean isAboveHalf( )
{
    return (getFlow( ) > 0.5);
}
```

**94 Chapter 2 / Java Classes and Information Hiding**

7. You'll find part of a solution in Figure 13.1 on page 677.

8. Java usually provides a no-arguments constructor that sets each instance variable to its initialization value (if there is one) or to its default value (if there is no initialization value).

9. The program should include the following statements:

```
Throttle exercise = new Throttle(6);
exercise.shift(3);
System.out.println
(exercise.getFlow());
```

10. The control should be assigned the value of null. By the way, if it is an instance variable of a class, then it is initialized to null.

11. A NullPointerException is thrown.

12. t1.shift(2) will cause an error because you cannot activate methods when t1 is null. The other two statements are fine.

13. Both t1 and t2 refer to the same throttle, which has been shifted up 42 positions. So the output is 0.42.

14. At the end of the code, (t1 == t2) is true since there is only one throttle that both variables refer to.

15. Here is the code (and at the end, t1 == t2 is false since there are two separate throttles):

```
Throttle t1;
Throttle t2;
t1 = new Throttle(100);
t2 = new Throttle(100);
t1.shift(42);
t2.shift(42);
```

16. Thermometer t = new Thermometer();
t.addCelsius(10);
System.out.println
(t.getFahrenheit());

17. com.knafn.statistics

18. Underneath your classes directory, create a subdirectory com. Underneath com create a subdirectory knafn. Underneath knafn create a subdirectory statistics. Your package is placed in the statistics subdirectory.

19. package com.knafn.statistics;

20. import com.knafn.statistics.\*;

21. import
com.knafn.statistics.Averager;

22. Java automatically imports java.lang; no explicit import statement is needed.

23. Public access is obtained with the keyword public, and it allows access by any program. Private access is obtained with the keyword private, and it allows access only by the methods of the class. Package access is obtained with no keyword, and it allows access within the package but not elsewhere.

24. Here is the code:

```
Location p1 = new Location(0, 0);
Location p2 = new Location(1, 1);
System.out.println
(Location.distance(p1, p2));
Location p3 =
Location.midpoint(p1, p2);
```

25. A static method is not activated by any one object. Instead, the class name is placed in front of the method to activate it. For example, the distance between two locations p1 and p2 is computed by:

```
Location.distance(p1, p2);
```

Within the implementation of a static method, we cannot directly refer to the instance variables.

26. The constant Double.NaN is used when there is no valid number to store in a double variable ("not a number").

27. The result is the constant
Double.POSITIVE\_INFINITY.

28. The alternative  $(p1.x + p2.x)/2$  has a subexpression  $p1.x + p2.x$  that could result in an overflow.
29. Here is the implementation for the throttle:
- ```
public boolean equals(Object obj)
{
    if (obj instanceof Throttle)
    {
        Throttle candidate = (Throttle) obj;
        return
            (candidate.top==top)
            &&
            (candidate.position==position);
    }
    else
        return false;
}
```
30. The automatic `equals` method returns `true` only when the two objects are the exact same object (as opposed to two separate objects that have the same value).
31. The solution is the same as the `Location` clone on page 69, but change the `Location` type to `Throttle`.
32. A `RuntimeException` indicates a programming error. When you throw a `RuntimeException`, you should provide an indication of the most likely cause of the error.
33. The argument remains unchanged.
34. The object that the argument refers to has been rotated 90°.

## PROGRAMMING PROJECTS



**1** Specify, design, and implement a class that can be used in a program that simulates a combination lock. The lock has a circular knob with the numbers 0 through 39 marked on the edge, and it has a three-number combination, which we'll call  $x$ ,  $y$ ,  $z$ . To open the lock, you must turn the knob clockwise at least one entire revolution, stopping with  $x$  at the top; then you turn the knob counterclockwise, stopping the *second* time that  $y$  appears at the top; finally, you turn the knob clockwise again, stopping the next time that  $z$  appears at the top. At this point, you may open the lock.

Your `Lock` class should have a constructor that initializes the three-number combination. Also provide methods:

- (a) To alter the lock's combination to a new three-number combination
- (b) To turn the knob in a given direction until a specified number appears at the top
- (c) To close the lock
- (d) To attempt to open the lock
- (e) To inquire about the status of the lock (open or shut)
- (f) To tell what number is currently at the top

**2** Specify, design, and implement a class called `Statistician`. After a statistician is initialized, it can be given a sequence of double numbers. Each number in the sequence is given to the statistician by activating a method called `nextNumber`. For example, we can declare a statistician called `s` and then give it the sequence of numbers 1.1, -2.4, 0.8, as shown here:

```
Statistician s = new Statistician();
s.nextNumber(1.1);
s.nextNumber(-2.4);
s.nextNumber(0.8);
```

After a sequence has been given to a statistician, there are various methods to obtain information about the sequence. Include methods that will provide the length of the sequence, the last number of the sequence, the sum of all the numbers in the sequence, the arithmetic mean of the numbers (i.e., the sum of the numbers divided by the length of the sequence), the smallest number in the sequence, and the largest number in the sequence. Notice that the length and sum methods can be called at any time, even if there are no numbers in the sequence. In this

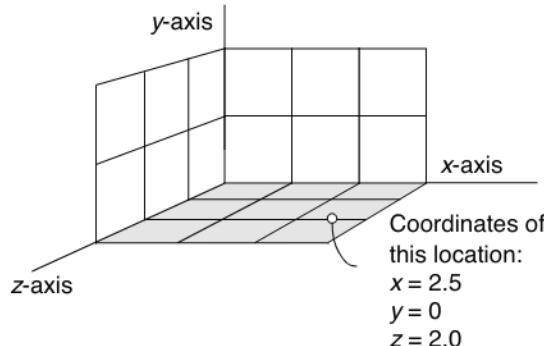
## 96 Chapter 2 / Java Classes and Information Hiding

case of an “empty” sequence, both length and sum will be zero. The other methods should return `Double.NaN` if they are called for an empty sequence.

Notes: Do not try to store the entire sequence (because you don’t know how long this sequence will be). Instead, just store the necessary information about the sequence: What is the sequence length; what is the sum of the numbers in the sequence; and what are the last, smallest, and largest numbers? Each of these pieces of information can be stored in a private instance variable that is updated whenever `nextNumber` is activated.

**3** Write a new static method to allow you to “add” two statisticians from the previous project. If `s1` and `s2` are two statisticians, then the result of adding them should be a new statistician that behaves as if it had all of the numbers of `s1` followed by all of the numbers of `s2`.

**4** Specify, design, and implement a class that can be used to keep track of the position of a location in three-dimensional space. For example, consider this location:



The location shown in the picture has three coordinates:  $x = 2.5$ ,  $y = 0$ , and  $z = 2.0$ . Include methods to set a location to a specified point, to shift a location a given amount along one of the axes, and to retrieve the coordinates of a location. Also provide methods that will rotate the location by a specified angle around a specified axis.

To compute these rotations, you will need a bit of trigonometry. Suppose you have a location with co-

ordinates  $x$ ,  $y$ , and  $z$ . After rotating this location by an angle  $\theta$ , the location will have new coordinates, which we’ll call  $x'$ ,  $y'$ , and  $z'$ . The equations for the new coordinates use the `java.lang` methods `Math.sin` and `Math.cos`, as shown here:

**After a  $\theta$  rotation around the x-axis:**

$$\begin{aligned}x' &= x \\y' &= y \cos(\theta) - z \sin(\theta) \\z' &= y \sin(\theta) + z \cos(\theta)\end{aligned}$$

**After a  $\theta$  rotation around the y-axis:**

$$\begin{aligned}x' &= x \cos(\theta) + z \sin(\theta) \\y' &= y \\z' &= -x \sin(\theta) + z \cos(\theta)\end{aligned}$$

**After a  $\theta$  rotation around the z-axis:**

$$\begin{aligned}x' &= x \cos(\theta) - y \sin(\theta) \\y' &= x \sin(\theta) + y \cos(\theta) \\z' &= z\end{aligned}$$

**5** In three-dimensional space, a line segment is defined by its two endpoints. Specify, design, and implement a class for a line segment. The class should have two private instance variables that are 3D locations from the previous project.

**6** Specify, design, and implement a class for a card in a deck of playing cards. The class should contain methods for setting and retrieving the suit and rank of a card.

**7** Specify, design, and implement a class that can be used to hold information about a musical note. A programmer should be able to set and retrieve the length of the note and the value of the note. The length of a note may be a sixteenth note, eighth note, quarter note, half note, or whole note. A value is specified by indicating how far the note lies above or below the A note that orchestras use in tuning. In counting “how far,” you should include both the white and black notes on a piano. For example, the note numbers for the octave beginning at middle C are shown at the top of the next page.



## 98 Chapter 2 / Java Classes and Information Hiding

- (5) If  $a$  is nonzero and  $b^2 = 4ac$ , then there is one real root,  $x = -b/2a$ .
- (6) If  $a$  is nonzero and  $b^2 > 4ac$ , then there are two real roots:

$$x = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Write a new method that returns the number of real roots of a quadratic expression. This answer could be 0, 1, 2, or infinity. In the case of an infinite number of real roots, have the method return 3. (Yes, we know that 3 is not infinity, but for this purpose it is close enough!) Write two other methods that calculate and return the real roots of a quadratic expression. The precondition for both methods is that the expression has at least one real root. If there are two real roots, then one of the methods returns the smaller of the two roots, and the other method returns the larger of the two roots. If every value of  $x$  is a real root, then both methods should return zero.

**10**

Specify, design, and implement a class that can be used to simulate a lunar lander, which is a small spaceship that transports astronauts from lunar orbit to the surface of the moon. When a lunar lander is constructed, the following items should be initialized as follows:

- (1) Current fuel flow rate as a fraction of the maximum fuel flow (initially zero)
- (2) Vertical speed of the lander (initially zero meters/sec)
- (3) Altitude of the lander (specified as a parameter of the constructor)
- (4) Amount of fuel (specified as a parameter of the constructor)
- (5) Mass of the lander when it has no fuel (specified as a parameter of the constructor)
- (6) Maximum fuel consumption rate (specified as a parameter of the constructor)
- (7) Maximum thrust of the lander's engine (specified as a parameter of the constructor)

Don't worry about other properties (such as horizontal speed).

The lander has accessor methods that allow a program to retrieve the current values of any of the preceding seven items. There are only two modification methods, described next.

The first modification method changes the current fuel flow rate to a new value ranging from 0.0 to 1.0. This value is expressed as a fraction of the maximum fuel flow.

The second modification method simulates the passage of a small amount of time. This time, called  $t$ , is expressed in seconds and will typically be a small value such as 0.1 seconds. The method will update the first four values in the preceding list to reflect the passage of  $t$  seconds. To implement this method, you will require a few physics formulas, listed below. These formulas are only approximate because some of the lander's values are changing during the simulated time period. But if the time span is kept short, these formulas will suffice.

**Fuel flow rate:** Normally, the fuel flow rate does not change during the passage of a small amount of time. But there is one exception: If the fuel flow rate is greater than zero and the amount of fuel left is zero, then you should reset the fuel flow rate to zero (because there is no fuel to flow).

**Velocity change:** During  $t$  seconds, the velocity of the lander changes by approximately this amount (measured in meters/sec):

$$t \times \left( \frac{f}{m} - 1.62 \right)$$

The value  $m$  is the total mass of the lander, measured in kilograms (i.e., the mass of a lander with no fuel plus the mass of any remaining fuel). The value  $f$  is the thrust of the lander's engine, measured in newtons. You can calculate  $f$  as the current fuel flow rate times the maximum thrust of the lander. The number  $-1.62$  is the downward acceleration from gravity on the moon.

**Altitude change:** During  $t$  seconds, the altitude of the lander changes by  $t \times v$  meters, where  $v$  is the vertical velocity of the lander (measured in meters/sec with negative values downward).

**Change in remaining fuel:** During  $t$  seconds, the amount of remaining fuel is reduced by  $t \times r \times c$  kilograms. The value of  $r$  is the current fuel flow rate, and  $c$  is the maximum fuel consumption (measured in kilograms per second).

We suggest that you calculate the changes to the four items in the order just listed. After all the changes have been made, there are two further adjustments. First, if the altitude has dropped below zero, then reset both altitude and velocity to zero (indicating that the ship has landed). Second, if the total amount of remaining fuel drops below zero, then reset this amount to zero (indicating that we have run out of fuel).

**11** In this project, you will design and implement a class that can generate a sequence of **pseudorandom** integers, which is a sequence that appears random in many ways. The approach uses the **linear congruence method**, explained below. The linear congruence method starts with a number called the **seed**. In addition to the seed, three other numbers are used in the linear congruence method: the **multiplier**, the **increment**, and the **modulus**. The formula for generating a sequence of pseudorandom numbers is quite simple. The first number is:

```
(multiplier * seed + increment) % modulus
```

This formula uses the Java % operator, which computes the remainder from an integer division.

Each time a new random number is computed, the value of the seed is changed to that new number. For example, we could implement a pseudorandom number generator with `multiplier = 40`, `increment = 3641`, and `modulus = 729`. If we choose the seed to be 1, then the sequence of numbers will proceed this way:

First number  
= `(multiplier * seed + increment) % modulus`  
= `(40 * 1 + 3641) % 729`  
= 36

and 36 becomes the new seed.

Next number  
= `(multiplier * seed + increment) % modulus`  
= `(40 * 36 + 3641) % 729`  
= 707

and 707 becomes the new seed.

These particular values for multiplier, increment, and modulus happen to be good choices. The pattern

generated will not repeat until 729 different numbers have been produced. Other choices for the constants might not be so good.

For this project, design and implement a class that can generate a pseudorandom sequence in the manner described. The initial seed, multiplier, increment, and modulus should all be parameters of the constructor. There should also be a method to permit the seed to be changed and a method to generate and return the next number in the pseudorandom sequence.

**12** Add a new method to the random number class of the previous project. The new method generates the next pseudorandom number but does not return the number directly. Instead, the method returns this number divided by the modulus. (You will have to cast the modulus to a double number before carrying out the division; otherwise, the division will be an integer division, throwing away the remainder.)

The return value from this new member function is a pseudorandom double number in the range [0..1]. (The square bracket, “[”, indicates that the range does include 0, but the rounded parenthesis, “)”, indicates that the range goes up to 1, without actually including 1.)

**13** Run some experiments to determine the distribution of numbers returned by the new pseudorandom method from the previous project. Recall that this method returns a double number in the range [0..1]. Divide this range into 10 intervals and call the method one million times, producing a table such as this:

| Range       | Number of Occurrences |
|-------------|-----------------------|
| [0...0.1)   | 99889                 |
| [0.1...0.2) | 100309                |
| [0.2...0.3) | 100070                |
| [0.3...0.4) | 99940                 |
| [0.4...0.5) | 99584                 |
| [0.5...0.6) | 100028                |
| [0.6...0.7) | 99669                 |
| [0.7...0.8) | 100100                |
| [0.8...0.9) | 100107                |
| [0.9...1.0) | 100304                |

**100** Chapter 2 / Java Classes and Information Hiding

Run your experiment for different values of the multiplier, increment, and modulus. With good choices for the constants, you will end up with about 10% of the numbers in each interval. A pseudorandom number generator with this equal-interval behavior is called **uniformly distributed**.

**14** This project is a continuation of the previous project. Many applications require pseudorandom number sequences that are *not* uniformly distributed. For example, a program that simulates the birth of babies can use random numbers for the birth weights of the newborns. But these birth weights should have a **Gaussian distribution**. In a Gaussian distribution, numbers form a bell-shaped curve in which values are more likely to fall in intervals near the center of the overall distribution. The exact probabilities of falling in a particular interval can be computed by knowing two numbers: (1) a number called the *variance*, which indicates how widely spread the distribution appears, and (2) the center of the overall distribution, called the *median*. For this kind of distribution, the median is equal to the arithmetic average (the *mean*) and equal to the most frequent value (the *mode*).

Generating a pseudorandom number sequence with an exact Gaussian distribution can be difficult, but there is a good way to approximate a Gaussian distribution using uniformly distributed random numbers in the range [0..1). The approach is to generate three pseudorandom numbers  $r_1$ ,  $r_2$ , and  $r_3$ , each of which is in the range [0..1). These numbers are then combined to produce the next number in the Gaussian sequence. The formula to combine the numbers is:

Next number in the Gaussian sequence

$$= \text{median} + (2 \times (r_1 + r_2 + r_3) - 3) \times \text{variance}$$

Add a new method to the random number class, which can be used to produce a sequence of pseudorandom numbers with a Gaussian distribution.

**15** Implement the Thermometer class from Self-Test Exercise 16 on page 60. Make sure that the methods to alter the temperature do

not allow the temperature to drop below absolute zero ( $-273.16^{\circ}\text{C}$ ). Also include these extra methods:

1. Two accessor methods that return the maximum temperature that the thermometer has ever recorded (with the return value in either Celsius or Fahrenheit degrees)
2. Two accessor methods that return the minimum temperature ever recorded
3. A modification method to reset the maximum and minimum counters
4. A boolean method that returns true if the temperature is at or below  $0^{\circ}\text{C}$

**16** Write a class for rational numbers. Each object in the class should have two integer values that define the rational number: the numerator and the denominator. For example, the fraction  $\frac{5}{6}$  would have a numerator of 5 and a denominator of 6. Include a constructor with two arguments that can be used to set the numerator and denominator (forbidding zero in the denominator). Also provide a no-arguments constructor that has zero for the numerator and 1 for the denominator.

Include a method that prints a rational number to `System.out` in a normal form (so that the denominator is as small as possible). Note that the numerator or denominator (or both) may contain a minus sign, but when a rational number is printed, the denominator should never include a minus sign. So if the numerator is 1 and the denominator is -2, then the printing method should print  $-1/2$ .

Include a function to normalize the values stored so that, after normalization, the denominator is positive and as small as possible. For example, after normalization,  $4/-8$  would be represented as  $-1/2$ .

Write static methods for the usual arithmetic operators to provide addition, subtraction, multiplication, and division of two rational numbers. Write static boolean methods to provide the usual comparison operations to allow comparison of two rational numbers.

Hints: Two rational numbers  $a/b$  and  $c/d$  are equal if  $a*d$  equals  $c*b$ . For positive rational numbers,  $a/b$  is less than  $c/d$  provided that  $a*d$  is less than  $c*b$ .

**17** Write a class to keep track of a balance in a bank account with a varying annual interest rate. The constructor will set both the balance and the annual interest rate to some initial values (and you should also implement a no-arguments constructor that sets both the balance and the interest rate to zero).

The class should have methods to change or retrieve the current balance or interest rate. There should also be methods to make a deposit (add to the balance) or a withdrawal (subtract from the balance).

Finally, there should be a method that adds interest to the balance at the current interest rate. This function should have a parameter indicating how many years' worth of interest are to be added. (For example, 0.5 years indicates that the account should have six months' interest added.)

Use the class as part of an interactive program that allows the user to determine how long an initial balance will take to grow to a given value. The program should allow the user to specify the initial balance, the interest rate, and whether there are additional yearly deposits.

**18** This project requires a little understanding of velocity and gravity, but don't let that scare you away! It's actually an easy project. The assignment is to write a class in which each object represents a satellite in orbit around the Earth's equator. Each object has four instance variables.

Two variables store the current  $x$  and  $y$  positions of the satellite in a coordinate system with the origin at the center of the Earth. The plane formed by the  $x$ - and  $y$ -axes comes out through the equator, with the positive  $x$ -axis passing through the equator at the Greenwich prime meridian. The positive  $y$ -axis is 90 degrees away from that, at the  $90^{\circ}\text{W}$  meridian, and all measurements are in meters. The drawing in the next column shows a possible location of the satellite as viewed from far above the South Pole.

Two other variables ( $vx$  and  $vy$ ) store the current velocity of the satellite in the  $x$  and  $y$  directions. These measurements are in meters per second. The values can be positive (which means that the satellite is moving toward the positive direction of the axis) or negative (which means that the satellite is moving toward the negative axis).

Your class should have methods to set and retrieve all four instance variables. Also include a modification method that will change all four instance variables to reflect the passage of a small amount of time  $t$  (measured in seconds). When  $t$  is small, the equations for the new values of  $x$ ,  $y$ ,  $vx$ , and  $vy$  are given by:

$$\text{new value of } x = x + vx*t$$

$$\text{new value of } y = y + vy*t$$

$$\text{new value of } vx = vx + ax*t$$

$$\text{new value of } vy = vy + ay*t.$$

The numbers  $ax$  and  $ay$  are the current accelerations from gravity along the  $x$ - and  $y$ -axes, determined by:

$$ax = -G*M*x/d^3$$

$$ay = -G*M*y/d^3$$

$$G = \text{gravitational constant in N-m}^2/\text{sec}^2$$

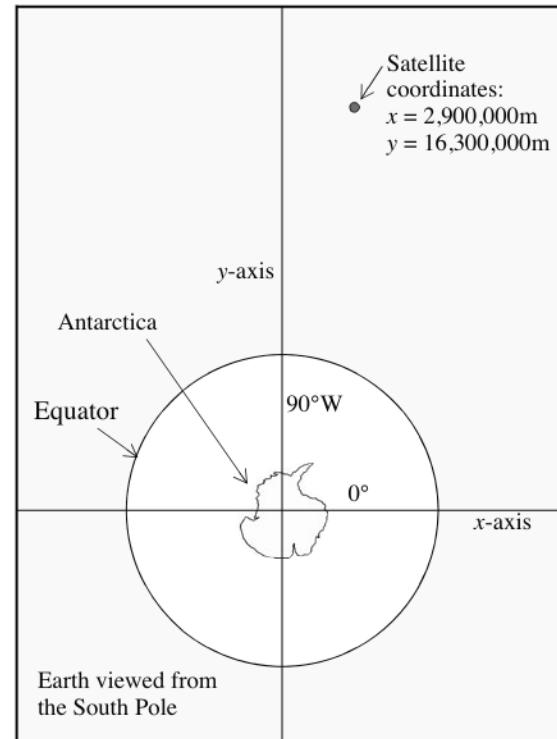
$$= 6.67 \times 10^{-11}$$

$$M = \text{mass of the Earth in kilograms}$$

$$= 5.97 \times 10^{24}$$

$$d = \text{distance of the satellite from center of Earth}$$

$$= \sqrt{x^2 + y^2}$$



**102** Chapter 2 / Java Classes and Information Hiding

Include accessor methods to retrieve the current values of  $ax$ ,  $ay$ ,  $d$ , and the satellite's current altitude above the surface of the Earth (the value of  $d$  minus the equatorial radius of the Earth, which is about 12,756,300 meters).

Include a boolean method that determines whether the satellite has crashed into the Earth. For the method that simulates the passage of a small amount of time, there should be a check to see whether the satellite has crashed into the surface of the Earth. If so, the method should set both velocities to zero and not change the  $x$  and  $y$  locations. Of course, all of this ignores many factors (such as air resistance when the satellite approaches the Earth's atmosphere).

Use your satellite class in a small application program that allows the user to set the initial values of  $x$ ,  $y$ ,  $vx$ , and  $vy$ . The program then simulates the satellite's flight by repeatedly calling the simulation method with  $t = 1$ . The program's output consists of a list of the  $x$ ,  $y$ ,  $vx$ , and  $vy$  values printed once every 60 simulated seconds (with the total amount of time simulated being specified by the user).

You can run your program using some actual values that are similar to Canada's Alouette I satellite, launched in 1962:

Initial  $x = 7,392,000$  meters

Initial  $y = 0$

Initial  $vx = 0$

Initial  $vy = 7,349$  meters/sec

Each complete orbit should take a bit more than 106 minutes.

**19** Specify, design, and implement a class called `Date`. Use integers to represent a date's month, day, and year. Write a method to increment the date to the next day.

Include methods to display a date in both number and word format.

**20** Specify, design, and implement a class called `Employee`. The class has instance variables for the employee's name, ID number, and salary based on an hourly wage. Methods can compute the yearly salary and increase the salary by a certain percentage. Add additional members to store the paycheck amount (calculated every two weeks) and calculate overtime (for over 40 hours per week) for each paycheck.

**21** Write a class for complex numbers. A complex number has the form  $a + bi$ , where  $a$  and  $b$  are real numbers and  $i$  is the square root of  $-1$ . We refer to  $a$  as the real part and  $b$  as the imaginary part of the number. The class should have two instance variables to represent the real and imaginary numbers; the constructor takes two arguments to set these members. Discuss and implement other appropriate methods for this class.

**22** Write a class called `fueler` that can keep track of the fuel and mileage of a vehicle. Include private instance variables to track the amount of fuel that the vehicle has consumed and the distance that the vehicle has traveled. You may choose whatever units you like (for example, fuel could be measured in U.S. gallons, Imperial gallons, or liters), but be sure to document your choices at the point where you declare the variables.

The class should have a constructor that initializes these variables to zero. Include a method that can later reset both variables to zero. There are two different modification methods to add a given amount to the total distance driven (one has a miles parameter, and the other has a kilometers parameter); similarly, there are three methods to add a given amount to the total fuel consumed (with different units for the amount of fuel).

The class has two accessor methods to retrieve the total distance driven (in miles or km), three methods for the fuel consumed (in U.S. gallons, Imperial gallons, or liters), and four for the fuel mileage (in U.S. mpg, Imperial mpg, km per liters, or liters per 100 km).



# CHAPTER 3

## Collection Classes

### LEARNING OBJECTIVES

---

When you complete Chapter 3, you will be able to ...

- use arrays in programs, including storing and retrieving elements, obtaining array lengths, iterating over an array with Java's new form of the for-loop, and using array assignments, array clones, `System.arraycopy`, and array parameters.
- design and implement collection classes that use arrays to store a collection of elements, generally using linear algorithms to access, insert, and remove elements.
- correctly maintain the capacity of an array, including increasing the capacity when needed because of the addition of a new element.
- implement and use methods with a variable number of arguments.
- write and maintain an accurate invariant for each class that you implement.
- write programs that use the `HashSet` class and its iterators (part of the Java Class Libraries).

### CHAPTER CONTENTS

---

- 3.1 A Review of Java Arrays
- 3.2 An ADT for a Bag of Integers
- 3.3 Programming Project: The Sequence ADT
- 3.4 Programming Project: The Polynomial
- 3.5 The Java `HashSet` and Iterators
  - Chapter Summary
  - Solutions to Self-Test Exercises
  - Programming Projects

# CHAPTER 3

*(I am large, I contain multitudes.)*

## Collection Classes

WALT WHITMAN  
“Song of Myself”

*an ADT in which each object contains a collection of elements*

The Throttle and Location classes in Chapter 2 are good examples of abstract data types (ADTs), but their applicability is limited to a few specialized programs. This chapter begins the presentation of several ADTs with broad applicability to programs large and small. The ADTs in this chapter—bags and sequences—are small, but they provide the basis for more complex ADTs such as Java’s `ArrayList` class.

The ADTs in this chapter are examples of **collection classes**. Intuitively, a collection class is a class in which each object contains a collection of elements. For example, one program might keep track of a collection of integers, perhaps the collection of test scores for a group of students. Another program, perhaps a cryptography program, can use a collection of characters.

There are many different ways to implement a collection class; the simplest approach utilizes an array, so this chapter begins with a quick review of Java arrays before approaching actual collection classes.

### 3.1 A REVIEW OF JAVA ARRAYS

An array is a sequence with a certain number of components. We draw arrays with each component in a separate box. For example, here’s an array of the four integers 7, 22, 19, and 56:

|   |    |    |    |
|---|----|----|----|
| 7 | 22 | 19 | 56 |
|---|----|----|----|

Each component of an array can be accessed through an index. In Java, the indexes are written with square brackets, beginning with [0], [1], ... . The array shown has four components, so the indexes are [0] through [3]:

|     |     |     |     |
|-----|-----|-----|-----|
| 7   | 22  | 19  | 56  |
| [0] | [1] | [2] | [3] |

In these examples, each component is an integer, but arrays can be built for any fixed data type: arrays of double numbers, arrays of boolean values, and even arrays in which the components are objects from a new class that you write yourself.

An array is declared like any other variable, except that a pair of square brackets is placed after the name of the data type. For example, a program can declare an array of integers like this:

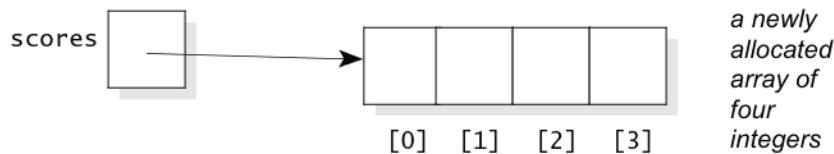
```
int[ ] scores;
```

The name of this array is `scores`. The components of this array are integers, but as we have mentioned, the components can be any fixed type. For example, an array of double numbers would use `double[ ]` instead of `int[ ]`.

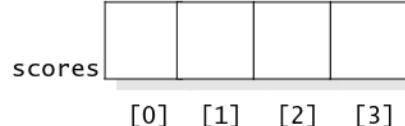
An array variable, such as `scores`, is capable of referring to an array of any size. In fact, an array variable is a reference variable, just like the reference variables we have used for other objects, and arrays are created with the same new operator that allocates other objects. For example, we can write these statements:

```
int[ ] scores;
scores = new int[4];
```

The number `[4]`, occurring with the `new` operator, indicates that we want a new array with four components. Once both statements finish, `scores` refers to an array with four integer components:



This is an accurate picture, showing how `scores` refers to a new array of four integers, but the picture has some clutter that we can usually omit. Here is a simpler picture that we'll usually use to show that `scores` refers to an array of four integers:



Both pictures mean the same thing. The first picture is a more accurate depiction of what Java actually does; the second is a kind of shorthand that is typical of what programmers draw to illustrate an array.

An array can also be declared and allocated with a single statement, such as:

```
int [ ] scores = new int[4];
```

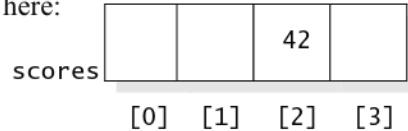
An array declared like this will initially be filled with Java's default values (for example, zeros for any number type). Instead of specifying an actual size with the `new` operator, another alternative is to provide a list of initial values. For example, these statements will create an array called `b` that contains the three values 10, 20, and 30:

```
int[ ] b = new int[ ] { 10, 20, 30 };
```

*providing initial values for an array*

**106 Chapter 3 / Collection Classes**

Once an array has been allocated, individual components can be selected using the square bracket notation with an index. For example, with scores allocated as shown, we can set the array's [2] component to 42 with the assignment `scores[2] = 42`. The result is shown here:

**PITFALL****EXCEPTIONS THAT ARISE FROM ARRAYS**

Two kinds of exceptions commonly arise from programming errors with arrays. One problem is trying to use an array variable before the array has been allocated. For example, suppose we declare `int[] scores`, but we forget to use the new operator to create an array for scores to refer to. At this point, scores is actually a reference variable, just like the reference variables we discussed for other kinds of objects on page 54. But merely declaring a reference variable doesn't allocate an array, and it is a programming error to try to access a component such as `scores[2]`. A program that tries to access a component of a nonexistent array may throw a `NullPointerException` (if the reference is null), or there may be a compile-time error (if the variable is an uninitialized local variable).

Another common programming error is trying to access an array outside of its bounds. For example, suppose that scores refers to an array with four components. The indexes are [0] through [3], so it is an error to use an index that is too small (such as `scores[-1]`) or too large (such as `scores[4]`). A program that tries to use these indexes will throw an `ArrayIndexOutOfBoundsException`.

**The Length of an Array**

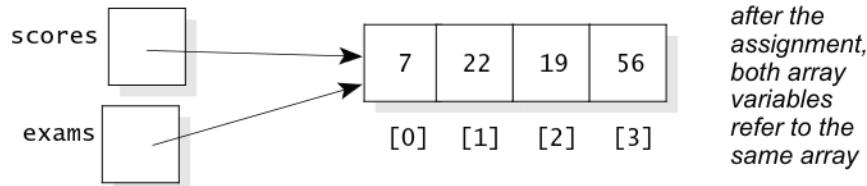
Every array has an instance variable called `length`, which tells the number of components in the array. For example, consider `scores = new int[4]`. After this allocation, `scores.length` is 4. Notice that `length` is not a method, so the syntax is merely `scores.length` (with no argument list). By the way, if an array variable is the null reference, you cannot ask for its length. (Trying to do so results in a `NullPointerException` or a compile-time error.)

**Assignment Statements with Arrays**

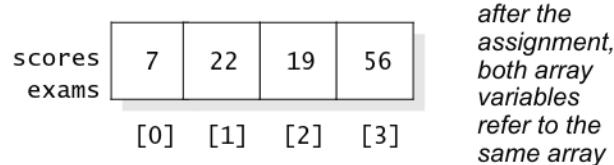
A program can use an assignment statement to make two array variables refer to the same array. Here is some example code:

```
int[] scores = new int[] {7, 22, 19, 56};  
int[] exams;  
exams = scores;
```

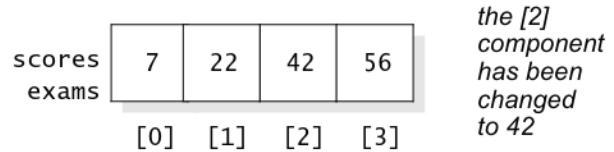
After these statements, `scores` refers to an array containing the four integers: 7, 22, 19, and 56. The assignment statement, `exams = scores`, causes `exams` to refer to the exact same array. Here is an accurate drawing of the situation:



Here's a shorthand drawing of the same situation to show that `scores` and `exams` refer to the same array:



In this example, there is only one array, and both array variables refer to this one array. Any change to the array will affect both `scores` and `exams`. For example, after the preceding statements, we might assign `exams[2] = 42`. The situation after the assignment to `exams[2]` is shown here:



At this point, both `exams[2]` and `scores[2]` are 42.

## Clones of Arrays

In Chapter 2, you saw how to use a `clone` method to create a completely separate copy of an object. Every Java array comes equipped with a `clone` method to create a copy of the array. Just like the other clones you've seen, changes to the original array don't affect the clone, and changes to the clone don't affect the original array. Here's an example:

```
int[ ] scores = new int[] {7, 22, 19, 56};  
int[ ] exams;  
exams = scores.clone();
```

The final statement in this example uses `scores.clone()` to create a copy of the `scores` array.

**108 Chapter 3 / Collection Classes**

Remember that in older versions of Java, the data type of the return value of any `clone` method was always Java's `Object` data type. Because of this, older code could not use the `clone` return value directly. For example, we could not write an assignment:

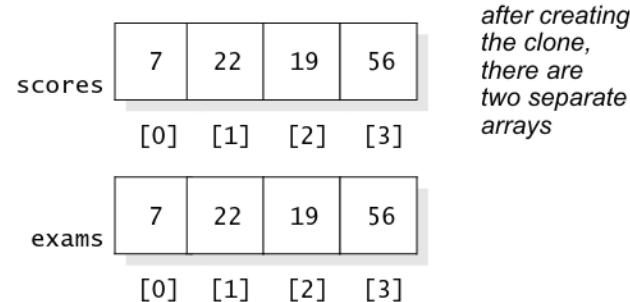
```
exams = scores.clone();
```

Instead, older code must apply a typecast to the `clone` return value, converting it to an integer array before we assign it to `exams`, like this:

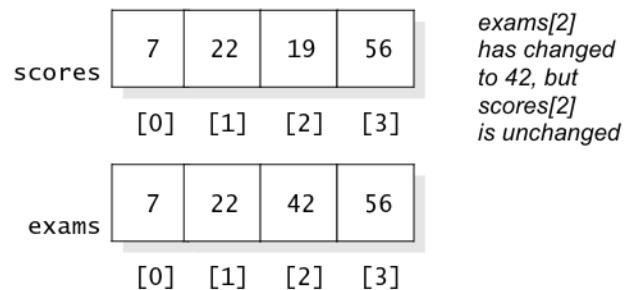
```
exams = (int[]) scores.clone();
```

The expression `(int[])` tells the compiler to treat the return value of the `clone` method as an integer array.

After the assignment statement, `exams` refers to a new array that is an exact copy of the `scores` array:



There are now two separate arrays. Changes to one array do not affect the other. For example, after the preceding statements, we might assign `exams[2] = 42`. The situation after the assignment to `exams[2]` is shown here:



At this point, `exams[2]` is 42, but `scores[2]` is unchanged.

### The Arrays Utility Class

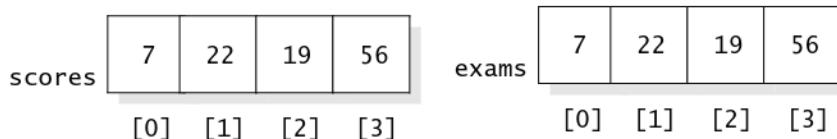
In Java, a **utility class** is a class that provides methods that are intended to help you manipulate objects from some other class. Generally, these utility methods

are static methods of the utility class, meaning that they can be used without being activated by any one particular object (see page 72). One of Java's utility classes is the `Arrays` utility from `java.util.Arrays`. It contains several dozen useful static methods for manipulating arrays. We'll look at three of these methods now, and others will be examined in later chapters.

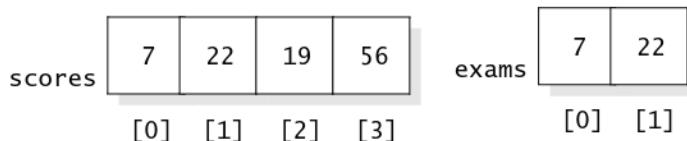
The first method, called `Arrays.copyOf`, provides an alternative way to make a clone of an array. Here is an example:

```
int[ ] scores = new int[] {7, 22, 19, 56};  
int[ ] exams = Array.copyOf(scores, 4);
```

The `copyOf` method has two arguments: an array to copy (such as `scores`) and the number of elements to copy (which is 4 in our example). So, after these statements, `exams` is a copy of the `scores` array:



We are not required to copy all of the array. For example, consider the assignment `exams = Arrays.copyOf(scores, 2)`, which results in this situation:



The number of items to copy may also be longer than the original array, such as `exams = Arrays.copyOf(scores, 5)`. In this case, the extra elements of the new array are filled with Java default values (which are zeros for numbers).

Notice that we always call `Arrays.copyOf` rather than `scores.copyOf`. That's because `copyOf` is a static method of the `Arrays` class. It is not activated by any single object; instead, the object to copy (such as `scores`) is passed as an argument to the `copyOf` method.

A second static method, called `Arrays.copyOfRange`, allows you to copy a section of one array into a newly created array. For example:

```
int[ ] scores = new int[] {7, 22, 19, 56};  
int[ ] exams = Array.copyOfRange(scores, 1, 3);
```

The argument 1 indicates the first index of `scores` to copy into `exams`; the argument 3 indicates the index that is just *after* the last element to copy. So, in this case, the elements `scores[1]` and `scores[2]` are copied into the new array (but `scores[3]` is not copied), as shown on the top of the next page.



**Array Parameters**

When a parameter is an array, then the parameter is initialized to refer to the same array that the actual argument refers to. Therefore, if the method changes the components of the array, the changes do affect the actual argument.

**PROGRAMMING TIP** **ENHANCED FOR-LOOPS FOR ARRAYS**

There are several new versions of for-loops that you might not have seen before. One version allows a program to step through every element of an array in order. Of course, we can do this with an integer variable that simply steps through each index, such as this method that computes the sum of all elements in an integer array (from Self-Test Exercise 11 on page 26):

```
public static int sum(int[ ] a)
{
    int answer, i;

    answer = 0;
    for (i = 0; i < a.length; i++)
        answer += a[i];
    return answer;
}
```

The new version of the for-loop uses a control variable that is declared in the parentheses following the keyword `for`. In the following example, we rewrite the `sum` method to access the elements of the array `a`, and we use the name `item` for the control variable. The body of the loop will be executed once with `item` set equal to `a[0]`, then with `item` set to `a[1]`, and so on, through all the items of the array:

```
public static int sum(int[ ] a)
{
    int answer;

    answer = 0;
    for (int item : a)
        answer += item;
    return answer;
}
```

**112 Chapter 3 / Collection Classes**

As with any loop, the work of the loop might be cut short by a return statement that is executed inside the loop. For example, here is another example of a method to determine whether a particular element appears in a double array:

```
public static boolean search(double[ ] data, double target)
{
    for (double item : data)
    { // Check whether item is the target.
        if (item == target)
            return true;
    }

    // The loop finished without finding the target.
    return false;
}
```

This form of the for-loop is called **iterating over an array**. The general format is given here, although you can use any variable name instead of item:

```
for (data type of the array elements item : name of the array)
{
    // Do something with item.
    // The first time through the loop, item will be set equal to the
    // [0] element of the array; the second time through the loop, item will
    // be set equal to the [1] element of the array, and so on.
    ...
}
```

**Self-Test Exercises for Section 3.1**

1. Suppose I have `int b = new int[42]`. What are the highest and lowest legal array indexes for b?
2. Write code that follows these steps: (1) Declare an integer array variable called b; (2) allocate a new array of 1000 integers for b to refer to; and (3) place the numbers 1 through 1000 in the array.
3. Redo the preceding exercise with only five elements instead of 1000. Your solution should be just a single Java statement.
4. Write a Java expression that will indicate how many elements are in the array b (from the previous exercise).
5. Consider the following statements:

```
int[ ] p = new int[100];
int[ ] s = p;
```

Which of the following statements will change the last value of p to 75? (There may be more than one correct answer.)

```
p[99] = 75;          p[100] = 75;
s[99] = 75;          s[100] = 75;
```

6. Repeat the previous exercise using these two initial statements:

```
int[ ] p = new int[100];
int[ ] s = p.clone( );
```

7. What is the output from this code?

```
int[ ] a, b;
a = new int[10];
a[5] = 0;
b = a;
a[5] = 42;
System.out(b[5]);
```

8. What is the output from this code?

```
int[ ] a, b;
a = new int[10];
a[5] = 0;
b = a.clone( );
a[5] = 42;
System.out(b[5]);
```

9. What kind of exception will be thrown by these statements?

```
int[ ] b;
b[0] = 12;
```

10. Suppose an array is passed as a parameter to a method, and the method changes the first component of the array to 42. What effect does this have on the actual argument back in the calling program?
11. Write a method that copies  $n$  elements from the front of one integer array to the front of another. The two arrays and the number  $n$  are all arguments to the method. Include a precondition/postcondition contract as part of your implementation.
12. Write a Java method called `zero_some` with one parameter  $x$  that is an array of double numbers. The method changes  $x[0], x[2], x[4], \dots$ , all to zero. If you activate `zero_some(a)` for an actual array  $a$  of length 6, which components of  $a$  will be changed to zero?
13. Use the new form of the for-loop to write a method that counts how many times a certain target appears in an integer array.

## 3.2 AN ADT FOR A BAG OF INTEGERS

This section provides an example of the design and implementation of a collection class. In this first example, the collection class will use an array to store its collection of elements (but later we will see other ways to store collections).

The example collection class is called a *bag of integers*. To describe the bag data type, think about an actual bag—a grocery bag or a garbage bag—and imagine writing integers on slips of paper and putting them in the bag. A **bag of integers** is similar to this imaginary bag: It's a container that holds a collection

## 114 Chapter 3 / Collection Classes

of integers that we place into it. A bag of integers can be used by any program that needs to store a collection of integers for its own use. For example, later we will write a program that keeps track of the ages of your family's members. If you have a large family with 10 people, the program keeps track of 10 ages—and these ages are kept in a bag of integers.

### The Bag ADT—Specification

We've given an intuitive description of a bag of integers. We will implement this bag as a class called `IntArrayBag`, in which the integers are stored in an array. We'll use a three-part name for a collection: "Int" specifies the type of the elements in the bag; "Array" indicates the mechanism for storing the elements; and "Bag" indicates the kind of collection. For a precise specification of the `IntArrayBag` class, we must describe each of the public methods to manipulate an `IntArrayBag` object.

In Chapter 5, we'll see how to implement more general collections that can be used for any kind of data (not just integers), but the `IntArrayBag` is a good place to start.

**The Constructors.** The `IntArrayBag` class has two constructors to initialize a new, empty bag. One constructor has a parameter, as shown in this heading:

```
public IntArrayBag(initialCapacity)
```

The parameter, `initialCapacity`, is the initial capacity of the bag—the number of elements that the bag can hold. Once this capacity is reached, more elements can still be added and the capacity will automatically increase in a manner that you'll see in a moment.

The other constructor has no parameters, and it constructs a bag with an initial capacity of 10.

**The add Method.** This is a modification method that places a new integer, called `element`, into a bag. Here is the heading:

```
public void add(int element)
```

As an example, here are some statements for a bag called `firstBag`:

```
IntArrayBag firstBag = new IntArrayBag( );
firstBag.add(8);
firstBag.add(4);           After these statements, firstBag
firstBag.add(8);          contains two 8s and a 4.
```



After these statements are executed, `firstBag` contains three integers: the number 4 and two copies of the number 8. Notice that a bag can contain many copies of the same integer, such as in this example, which has two copies of 8.

**The addMany Method.** This method is similar to the add method, except it has a strange new syntax in the heading:

```
public void addMany(int... elements)
```

The ellipsis after the data type (`int...`) means that the method can be called with any number of integer arguments. For example, any of these statements are permitted with a bag called `firstBag`:

```
firstBag.addMany(8,4,4); // Add one 8 and two 4s to the bag.  
firstBag.addMany(1,2,3,4,5,6,7,8,9,10); // Add 1 through 10.  
firstBag.addMany(5); // Just like firstBag.add(5);
```

The method adds each of its arguments to the bag. This kind of method has a **variable number of parameters**. Mathematicians use the term **arity** for the number of arguments, so programmers say this is a **variable arity method**. A variable arity may be used only with the final (rightmost) parameter of a method. We'll also need to see how the implementation accesses the many parameters, but that can wait until the implementation stage.

*variable arity*

**The addAll Method.** This method allows us to insert the contents of one bag into the existing contents of another bag. The method has this heading:

```
public void addAll(IntArrayBag addend)
```

We use the name `addend` for the parameter, meaning “something to be added.” As an example, suppose we create two bags called `helter` and `skelter`, and we then want to add all the contents of `skelter` to `helter`:

```
IntArrayBag helter = new IntArrayBag();  
IntArrayBag skelter = new IntArrayBag();  
helter.add(8);  
skelter.add(4);  
skelter.add(8);  
helter.addAll(skelter);
```

*This adds the contents of  
skelter to what's  
already in helter.*

After these statements, `helter` contains one 4 and two 8s.

**The remove Method.** The heading for this modification method is:

```
public boolean remove(int target)
```

Provided that `target` is actually in the bag, the method removes one copy of `target` and returns `true` to indicate that something has been removed. If `target` isn't in the bag, then the method just returns `false`.

**The size Method.** This accessor method returns the count of how many integers are in a bag. The heading is:

```
public int size()
```

For example, suppose `firstBag` contains one copy of the number 4 and two copies of the number 8. Then `firstBag.size()` returns 3.

**116 Chapter 3 / Collection Classes**

**The countOccurrences Method.** This is an accessor method that determines how many copies of a *particular* number are in a bag. The heading is:

```
public int countOccurrences(int target)
```

The return value of `countOccurrences(n)` is the number of occurrences of `n` in a bag. For example, if `firstBag` contains the number 4 and two copies of the number 8, then we will have these values:

```
System.out.println(firstBag.countOccurrences(1)); Prints 0
System.out.println(firstBag.countOccurrences(4)); Prints 1
System.out.println(firstBag.countOccurrences(8)); Prints 2
```

**The union Method.** The **union** of two bags is a new, larger bag that contains all the numbers in the first bag and all the numbers in the second bag:



We will implement `union` with a static method that has two parameters:

```
public static IntArrayBag union(IntArrayBag b1, IntArrayBag b2)
```

The `union` method computes the union of `b1` and `b2`. For example:

```
IntArrayBag part1 = new IntArrayBag();
IntArrayBag part2 = new IntArrayBag();

part1.add(8);
part1.add(9);
part2.add(4);
part2.add(8); This computes the union of
the two bags, putting the result
in a third bag.

IntArrayBag total = IntArrayBag.union(part1, part2);
```

After these statements, `total` contains one 4, two 8s, and one 9.

The `union` method is similar to `addAll`, but the usage is different. The `addAll` method is an ordinary method that is activated by a bag (for example, `helter.addAll(skelter)`, which adds the contents of `skelter` to `helter`). On the other hand, `union` is a static method with two arguments. As a static method, `union` is not activated by any one bag. Instead, the activation of `IntArrayBag.union(part1, part2)` creates and returns a new bag that includes the contents of both `part1` and `part2`.

**The `clone` Method.** As part of our specification, we require that bag objects can be copied with a `clone` method. For example:

```
IntArrayBag b = new IntArrayBag();
b.add(42);
IntArrayBag c = b.clone();
```

*b now contains a 42*

*c is initialized  
as a clone of b*

At this point, because we are only specifying which operations can manipulate a bag, we don't need to say anything more about the `clone` method.

**Three Methods That Deal with Capacity.** Each bag has a current **capacity**, which is the number of elements the bag can hold without having to request more memory. Once the capacity is reached, more elements can still be added by using the `add` method. In this case, the `add` method itself will increase the capacity as needed. In fact, our implementation of `add` will double the capacity whenever the bag becomes full.

With this in mind, you might wonder why a programmer needs to worry about the capacity at all. For example, why does the constructor require the programmer to specify an initial capacity? Couldn't we always use the constructor that has an initial capacity of 10 and have the `add` method increase capacity as more and more elements are added? Yes, this approach will always work correctly. But if there are many elements, then many of the activations of `add` would need to increase the capacity. This could be inefficient. To avoid repeatedly increasing the capacity, a programmer provides an initial guess at the needed capacity for the constructor.

For example, suppose a programmer expects no more than 1000 elements for a bag named `kilosack`. The bag is declared this way, with an initial capacity of 1000:

```
IntArrayBag kilosack = new IntArrayBag(1000);
```

After this declaration, the programmer can place 1000 elements in the bag without worrying about the capacity. Later, the programmer can add more elements to the bag, maybe even more than 1000. If there are more than 1000 elements, then `add` increases the capacity as needed.

There are three methods that allow a programmer to manipulate a bag's capacity after the bag is in use. The methods have these headers:

```
public int getCapacity()
public void ensureCapacity(int minimumCapacity)
public void trimToSize()
```

The first method, `getCapacity`, just returns the current capacity of the bag. The second method, `ensureCapacity`, increases the capacity to a specified minimum amount. For example, to ensure that a bag called `bigboy` has a capacity of at least 10,000, we would activate `bigboy.ensureCapacity(10000)`.

## 118 Chapter 3 / Collection Classes

The third method, `trimToSize`, reduces the capacity of a bag to its current size. For example, suppose that `bigboy` has a current capacity of 10,000, but it contains only 42 elements, and we are not planning to add any more. Then we can reduce the current capacity to 42 with the activation `bigboy.trimToSize()`. Trimming the capacity is never required, but doing so can reduce the memory used by a program.

That's all the methods, and we're almost ready to write the methods' specifications. But first there are some limitations to discuss.

### OutOfMemoryError and Other Limitations for Collection Classes

Our plan is to store a bag's elements in an array and to increase the capacity of the array as needed. The memory for any array comes from a location called the program's **heap** (also called the **free store**). In fact, the memory for all Java objects comes from the heap.

*what happens  
when the heap  
runs out of  
memory?*

If a heap has insufficient memory for a new object or array, then the result is a Java exception called `OutOfMemoryError`. This exception is thrown automatically by an unsuccessful “new” operation. For example, if there is insufficient memory for a new `Throttle` object, then `Throttle t = new Throttle()` throws an `OutOfMemoryError`. Experienced programmers may monitor the size of the heap and the amount that is still unused. Our programs won't attempt such monitoring, but our specification for any collection class will always mention that the maximum capacity is limited by the amount of free memory. To aid more experienced programmers, the specification will also indicate precisely which methods have the possibility of throwing an `OutOfMemoryError`. (Any method that uses the “new” operation could throw this exception.)

*collection  
classes may be  
limited by the  
maximum value  
of an integer*

Many collection classes have another limitation that is tied to the maximum value of an integer. In particular, our bag stores the elements in an array, and every array has integers for its indexes. Java integers are limited to no more than 2,147,483,647, which is also written as `Integer.MAX_VALUE`. An attempt to create an array with a size beyond `Integer.MAX_VALUE` results in an arithmetic overflow during the calculation of the size of the array. Such an overflow usually produces an array size that Java's runtime system sees as negative. This is because Java represents integers so that the “next” number after `Integer.MAX_VALUE` is actually the smallest negative number.

Programmers often ignore the array-size overflow problem (since today's machines generally have an `OutOfMemoryError` before `Integer.MAX_VALUE` is approached). We won't provide special code to handle this problem, but we won't totally ignore the problem either. Instead, our documentation will indicate precisely which methods have the potential for an array-size overflow. We'll also add a note to advise that large bags should probably use a different implementation method anyway because many of the array-based algorithms are slow for large bags ( $O(n)$ , where  $n$  is the number of elements in the bag).

### The IntArrayBag Class—Specification

We now know enough about the `IntArrayBag` to write a specification, as shown in Figure 3.1. It is part of a package named `edu.colorado.collections`.

**FIGURE 3.1** Specification for the `IntArrayBag` Class

### Class `IntArrayBag`

❖ **public class IntArrayBag from the package edu.colorado.collections**

An `IntArrayBag` is a collection of `int` numbers.

**Limitations:**

- (1) The capacity of one of these bags can change after it's created, but the maximum capacity is limited by the amount of free memory on the machine. The constructor, `add`, `clone`, and `union` will result in an `OutOfMemoryError` when free memory is exhausted.
- (2) A bag's capacity cannot exceed the largest integer, 2,147,483,647 (`Integer.MAX_VALUE`). Any attempt to create a larger capacity results in failure due to an arithmetic overflow.
- (3) Because of the slow linear algorithms of this class, large bags will have poor performance.

### Specification

❖ **Constructor for the `IntArrayBag`**

`public IntArrayBag()`

Initialize an empty bag with an initial capacity of 10. Note that the `add` method works efficiently (without needing more memory) until this capacity is reached.

**Postcondition:**

This bag is empty and has an initial capacity of 10.

**Throws:** `OutOfMemoryError`

Indicates insufficient memory for `new int[10]`.

❖ **Second Constructor for the `IntArrayBag`**

`public IntArrayBag(int initialCapacity)`

Initialize an empty bag with a specified initial capacity.

**Parameter:**

`initialCapacity` – the initial capacity of this bag

**Precondition:**

`initialCapacity` is non-negative.

**Postcondition:**

This bag is empty and has the specified initial capacity.

**Throws:** `IllegalArgumentException`

Indicates that `initialCapacity` is negative.

**Throws:** `OutOfMemoryError`

Indicates insufficient memory for allocating the bag.

(continued)

**120 Chapter 3 / Collection Classes***(FIGURE 3.1 continued)***◆ add**

```
public void add(int element)
```

Add a new element to this bag. If this new element would take this bag beyond its current capacity, then the capacity is increased before adding the new element.

**Parameter:**

element – the new element that is being added

**Postcondition:**

A new copy of the element has been added to this bag.

**Throws: OutOfMemoryError**

Indicates insufficient memory for increasing the capacity.

**Note:**

Creating a bag with capacity beyond Integer.MAX\_VALUE causes arithmetic overflow.

**◆ addAll**

```
public void addAll(IntArrayBag addend)
```

Add the contents of another bag to this bag.

**Parameter:**

addend – a bag whose contents will be added to this bag

**Precondition:**

The parameter, addend, is not null.

**Postcondition:**

The elements from addend have been added to this bag.

**Throws: NullPointerException**

Indicates that addend is null.

**Throws: OutOfMemoryError**

Indicates insufficient memory to increase the size of this bag.

**Note:**

Creating a bag with capacity beyond Integer.MAX\_VALUE causes arithmetic overflow.

**◆ addMany**

```
public void addMany(int... elements)
```

Add a variable number of new elements to this bag. If these new elements would take this bag beyond its current capacity, then the capacity is increased before adding the new elements.

**Parameter:**

elements – a variable number of new elements that are all being added

**Postcondition:**

New copies of all the elements have been added to this bag.

**Throws: OutOfMemoryError**

Indicates insufficient memory for increasing the capacity.

**Note:**

Creating a bag with capacity beyond Integer.MAX\_VALUE causes arithmetic overflow.

(continued)

(FIGURE 3.1 continued)

◆ **clone**

```
public IntArrayBag clone( )
```

Generate a copy of this bag.

**Returns:**

The return value is a copy of this bag. Subsequent changes to the copy will not affect the original, nor vice versa. The return value must be typecast to an `IntArrayBag` before it is used.

**Throws:** `OutOfMemoryError`

Indicates insufficient memory for creating the clone.

◆ **countOccurrences**

```
public int countOccurrences(int target)
```

Accessor method to count the number of occurrences of a particular element in this bag.

**Parameter:**

`target` – the element that needs to be counted

**Returns:**

the number of times that `target` occurs in this bag

◆ **ensureCapacity**

```
public void ensureCapacity(int minimumCapacity)
```

Change the current capacity of this bag.

**Parameter:**

`minimumCapacity` – the new capacity for this bag

**Postcondition:**

This bag's capacity has been changed to at least `minimumCapacity`. If the capacity was already at or greater than `minimumCapacity`, then the capacity is left unchanged.

**Throws:** `OutOfMemoryError`

Indicates insufficient memory for new `int[minimumCapacity]`.

◆ **getCapacity**

```
public int getCapacity( )
```

Accessor method to determine the current capacity of this bag. The `add` method works efficiently (without needing more memory) until this capacity is reached.

**Returns:**

the current capacity of this bag

◆ **remove**

```
public boolean remove(int target)
```

Remove one copy of a specified element from this bag.

**Parameter:**

`target` – the element to remove from this bag

**Postcondition:**

If `target` was found in this bag, then one copy of `target` has been removed and the method returns `true`. Otherwise, this bag remains unchanged, and the method returns `false`.

(continued)

**122 Chapter 3 / Collection Classes***(FIGURE 3.1 continued)***◆ size**

```
public int size()
```

Accessor method to determine the number of elements in this bag.

**Returns:**

the number of elements in this bag

**◆ trimToSize**

```
public void trimToSize()
```

Reduce the current capacity of this bag to its actual size (i.e., the number of elements it contains).

**Postcondition:**

This bag's capacity has been changed to its current size.

**Throws: OutOfMemoryError**

Indicates insufficient memory for altering the capacity.

**◆ union**

```
public static IntArrayBag union(IntArrayBag b1, IntArrayBag b2)
```

Create a new bag that contains all the elements from two other bags.

**Parameters:**

b1 – the first of two bags

b2 – the second of two bags

**Precondition:**

Neither b1 nor b2 is null.

**Returns:**

a new bag that is the union of b1 and b2

**Throws: NullPointerException**

Indicates that one of the arguments is null.

**Throws: OutOfMemoryError**

Indicates insufficient memory for the new bag.

**Note:**

Creating a bag with capacity beyond Integer.MAX\_VALUE causes arithmetic overflow.

### The IntArrayBag Class—Demonstration Program

With the specification in hand, we can write a program that uses a bag. We don't need to know what the instance variables of a bag are, and we don't need to know how the methods are implemented. As an example, a demonstration program appears in Figure 3.2. The program asks a user about the ages of family members. The user enters the ages, followed by a negative number to indicate the end of the input. (Using a special value to end a list is a common technique; this value is called a **sentinel value**.) A typical dialogue with the program looks like this:



**124** Chapter 3 / Collection Classes

(FIGURE 3.2 continued)

```
public static void getAges(IntArrayBag ages)
// The getAges method prompts the user to type in the ages of family members. These
// ages are read and placed in the ages bag, stopping when the user types a negative
// number. This demonstration does not worry about the possibility of running out
// of memory (therefore, an OutOfMemoryError is possible).
{
    int userInput; // An age from the user's family

    System.out.println("Type the ages of your family members.");
    System.out.println("Type a negative number at the end and press return.");
    userInput = stdin.nextInt();
    while (userInput >= 0)
    {
        ages.add(userInput);
        userInput = stdin.nextInt();
    }
}

public static void checkAges(IntArrayBag ages)
// The checkAges method prompts the user to type in the ages of family members once
// again. Each age is removed from the ages bag when it is typed, stopping when the bag
// is empty.
public static void checkAges(IntArrayBag ages)
{
    int userInput; // An age from the user's family

    System.out.print("Type those ages again. ");
    System.out.println("Press return after each age.");
    while (ages.size() > 0)
    {
        System.print("Next age: ");
        userInput = stdin.nextInt();
        if (ages.countOccurrences(userInput) == 0)
            System.out.println("No, that age does not occur!");
        else
        {
            System.out.println("Yes, I've got that age and will remove it.");
            ages.remove(userInput);
        }
    }
}
```

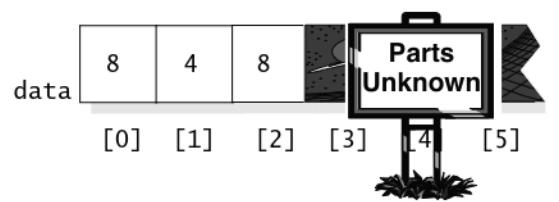
---

## The IntArrayBag Class—Design

There are several ways to design the IntArrayBag class. For now, we'll keep things simple and design a somewhat inefficient data structure using an array. Later, the data structure will be redesigned several times for more efficiency.

We start the design by thinking about the data structure—the actual configuration of private instance variables used to implement the class. The primary structure for our design is an array that stores the elements of a bag. Or, to be more precise, we use the *beginning part* of a large array. Such an array is called a **partially filled array**. For example, if the bag contains the integer 4 and two copies of 8, then the first part of the array could look this way:

Components of  
the partially filled  
array contain the  
elements of the bag.



use the  
beginning part  
of an array

This array, called `data`, will be one of the private instance variables of the IntArrayBag class. The length of the array will be determined by the current capacity, but as the picture indicates, when we are using the array to store a bag with just three elements, we don't care what appears beyond the first three components. Starting at index 3, the array might contain all zeros, or it might contain garbage or our favorite number—it really doesn't matter.

Because part of the array can contain garbage, the IntArrayBag class must keep track of one other item: *How much of the array is currently being used?* For example, in the preceding picture, we are using only the first three components of the array because the bag contains three elements. The amount of the array being used can be as small as zero (an empty bag) or as large as the current capacity. The amount increases as elements are added to the bag, and it decreases as elements are removed. In any case, we will keep track of the amount in a private instance variable called `manyItems`. With this approach, there are two instance variables for a bag:

```
public class IntArrayBag implements Cloneable
{
    private int[ ] data;    // An array to store elements
    private int manyItems; // How much of the array is used

    || The public methods will be given in a moment.
}
```

the bag's  
instance  
variables

Notice that we are planning to implement a `clone` method; therefore, we indicate “`implements Cloneable`” at the start of the class definition.

### The Invariant of an ADT

We've defined the bag data structure, and we have an intuitive idea of how the structure will be used to represent a bag of elements. But as an aid in implementing the class, we should also write down an explicit statement of how the data structure is used to represent a bag. In the case of the bag, we need to state how the instance variables of the class are used to represent a bag of elements. There are two rules for our bag implementation:

*rules that dictate  
how the instance  
variables are  
used to  
represent a  
value*

1. The number of elements in the bag is stored in the instance variable `manyItems`, which is no more than `data.length`.
2. For an empty bag, we do not care what is stored in any of `data`; for a non-empty bag, the elements of the bag are stored in `data[0]` through `data[manyItems-1]`, and we don't care what is stored in the rest of `data`.

The rules that dictate how the instance variables of a class represent a value (such as a bag of elements) are called the **invariant of the ADT**. The knowledge of these rules is essential to the correct implementation of the ADT's methods. With the exception of the constructors, each method depends on the invariant being valid when the method is activated. And each method, including the constructors, has the responsibility of ensuring that the invariant is valid when the method finishes. In some sense, the invariant of an ADT is a condition that is an *implicit* part of every method's postcondition. And (except for the constructors) it is also an implicit part of every method's precondition. The invariant is not usually written as an *explicit* part of the precondition and postcondition because the programmer who uses the ADT does not need to know about these conditions. But to the implementor of the ADT, the invariant is indispensable. In other words, the invariant is a critical part of the implementation of an ADT, but it has no effect on the way the ADT is used.

#### Key Design Concept

The invariant is a critical part of an ADT's implementation.

### The Invariant of an ADT

When you design a new class, always make an explicit statement of the rules that dictate how the instance variables are used. These rules are called the **invariant of the ADT**. All of the methods (except the constructors) can count on the invariant being valid when the method is called. Each method also has the responsibility of ensuring that the invariant is valid when the method finishes.

Once the invariant of an ADT is stated, the implementation of the methods is relatively simple because there is no interaction between the methods—except

for their cooperation in keeping the invariant valid. We'll look at these implementations one at a time, starting with the constructors.

### The IntArrayBag ADT—Implementation

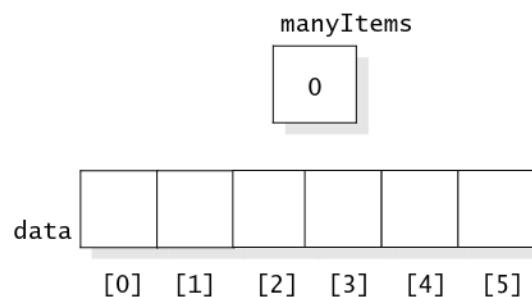
**The Constructor.** Every constructor has one primary job: to set up the instance variables correctly. In the case of the bag, the constructor must set up the instance variables so that they represent an empty bag with a current capacity given by the parameter `initialCapacity`. The bag has two instance variables, so its constructor will include two assignment statements, shown in this implementation of one of the constructors:

```
public IntArrayBag(int initialCapacity)
{
    if (initialCapacity < 0)
        throw new IllegalArgumentException
            ("initialCapacity is negative: " + initialCapacity);
    manyItems = 0;
    data = new int[initialCapacity];
}
```

The if-statement at the start checks the constructor's precondition. The first assignment statement, `manyItems = 0`, simply sets `manyItems` to zero, indicating that the bag does not yet have any elements. The second assignment statement, `data = new int[initialCapacity]`, is more interesting. This statement allocates an array of the right capacity (`initialCapacity`) and makes `data` refer to the new array. For example, suppose that `initialCapacity` is 6. After the two assignment statements, the instance variables look like this:

*The private instance variables of this bag include an array of six integers.*

*The bag does not yet contain any elements, so none of the array is being used.*



*implementing  
the constructor*

Later, the program could add many elements to this bag, maybe even more than six. If there are more than six elements, then the bag's methods will increase the array's capacity as needed.

The other constructor is similar, except it always provides an initial capacity of 10.

**128 Chapter 3 / Collection Classes**

**The add Method.** The add method checks whether there is room to add a new element. If not, the array capacity is increased before proceeding. (The new capacity is twice the old capacity plus 1. The extra +1 deals with the case in which the original size was zero.) The attempt to increase the array capacity may lead to an `OutOfMemoryError` or an arithmetic overflow, as discussed on page 118. But usually these errors do not occur, and we can place the new element in the next available location of the array. What is the index of the next available location? For example, if `manyItems` is 3, then `data[0]`, `data[1]`, and `data[2]` are already occupied, and the next location is `data[3]`. In general, the next available location will be `data[manyItems]`. We can place the new element in `data[manyItems]`, as shown in this implementation:

*implementing  
add*

```
public void add(int element)
{
    if (manyItems == data.length)
    {
        // Double the capacity and add 1; this works even if manyItems is 0.
        // However, in the case that manyItems*2 + 1 is beyond
        // Integer.MAX_VALUE, there will be an arithmetic overflow and
        // the bag will fail.
        ensureCapacity(manyItems*2 + 1);
    }

    data[manyItems] = element; ← See Self-Test Exercise 22
    manyItems++;
}
```

*for an alternative approach  
to these steps.*

Within a method we can activate other methods, such as the way that the add implementation activates `ensureCapacity` to increase the capacity of the array.

**The addAll Method.** The addAll method has this heading:

```
public void addAll(IntArrayBag addend)
```

*implementing  
addAll*

The bag that activates `addAll` is increased by adding all the elements from `addend`. Our implementation follows these steps:

1. Ensure that the capacity of the bag is large enough to contain its current elements plus the extra elements that will come from `addend`, as shown here:

```
ensureCapacity(manyItems + addend.manyItems);
```

By the way, what happens in this statement if `addend` is null? Of course, a null value violates the precondition of `addAll`, but a programmer could mistakenly provide null. In that case, a `NullPointerException` will be thrown, and this possibility is documented in the specification of `addAll` on page 120.

2. Copy the elements from addend.data to the next available positions in our own data array. In other words, we will copy addend.manyItems elements from the front of addend.data. These elements go into our own data array beginning at the next available spot, data[manyItems]. We could write a loop to copy these elements, but a quicker approach is to use Java's System.arraycopy method, which has these five arguments:

```
System.arraycopy(source, si, destination, di, n);
```

The arguments source and destination are two arrays, and the other arguments are integers. The method copies n elements from source (starting at source[si]) to the destination array (with the elements being placed at destination[di] through destination[di+n-1]). For our purposes, we call the **arraycopy** method:

```
System.arraycopy  
(addend.data, 0, data, manyItems, addend.manyItems);
```

*the arraycopy method*

3. Increase our own manyItems by addend.manyItems:

```
manyItems += addend.manyItems;
```

These three steps are shown in the addAll implementation of Figure 3.3.

**The addMany Method.** This method has a variable number of parameters, as indicated in its heading:

```
public void addMany(int... elements)
```

---

**FIGURE 3.3** Implementation of the Bag's addAll Method

### Implementation

```
public void addAll(IntArrayBag addend)  
{  
    // If addend is null, then a NullPointerException is thrown.  
    // In the case that the total number of items is beyond  
    // Integer.MAX_VALUE, there will be  
    // arithmetic overflow and the bag will fail.  
    ensureCapacity(manyItems + addend.manyItems);  
  
    System.arraycopy(addend.data, 0, data, manyItems, addend.manyItems);  
    manyItems += addend.manyItems;  
}
```

---

**FIGURE 3.4** Implementation of the Bag's addMany Method

### Implementation

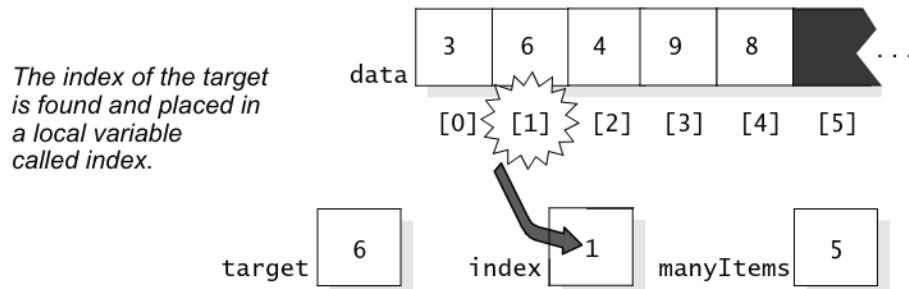
```
public void addMany(int... elements)
{
    if (manyItems + elements.length > data.length)
    { // Ensure twice as much space as we need.
        ensureCapacity((manyItems + elements.length)*2);
    }

    System.arraycopy(elements, 0, data, manyItems, elements.length);
    manyItems += elements.length;
}
```

#### *implementing addMany*

When the method is activated, the Java runtime system will take the actual arguments and put them in an array that the implementation can access with the parameter name (`elements`). For example, if we activate `b.addMany(4, 8, 4)`, then `elements` will be an array containing the three integers 4, 8, and 4. One possible implementation is to activate the ordinary `add` method once for each integer in the `elements` array. This would work fine, but a more efficient approach uses `System.arraycopy`, as shown in Figure 3.4.

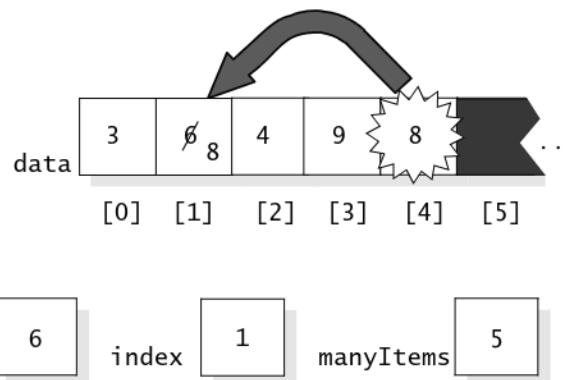
**The remove Method.** The `remove` method takes several steps to remove an element named `target` from a bag. In the first step, we find the index of `target` in the bag's array and store this index in a local variable named `index`. For example, suppose that `target` is the number 6 in this bag:



In this example, `target` is a parameter to the `remove` method, `index` is a local variable in the `remove` method, and `manyItems` is the bag instance variable. As you can see in the drawing, the first step of `remove` is to locate the target (6) and place the index of the target in the local variable called `index`.

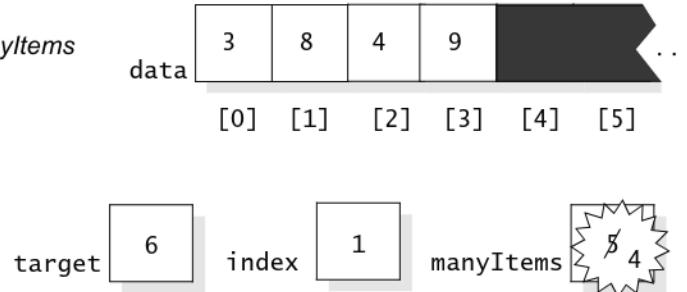
Once the index of the target is found, the second step is to take the *final* element in the bag and copy it to `data[index]`. The reason for this copying is so that all the bag's elements stay together at the front of the partially filled array, with no “holes.” In our example, the number 8 is copied to `data[index]`:

*The final element is copied onto the element that we are removing.*



The third step is to reduce `manyItems` by one—in effect reducing the used part of the array by one. In our example, `manyItems` is reduced from 5 to 4:

*The value of manyItems is reduced by one to indicate that one element has been removed.*



The code for the `remove` method, shown in Figure 3.5, follows these three steps. There is also a check that the target is actually in the bag. If we discover that the target is not in the bag, then we do not need to remove anything. Also note that our method works correctly for the boundary values of removing the first or last element in the array.

Before we continue, we want to point out a programming technique. Look at the following loop from Figure 3.5:

```
index = 0;
while ((index < manyItems) && (target != data[index]))
    index++;
```

*implementing  
remove*

**132 Chapter 3 / Collection Classes****FIGURE 3.5** Implementation of the Bag's Method to Remove an ElementImplementation

```
public boolean remove(int target)
{
    int index; // The location of target in the data array

    // First, set index to the location of target in the data array,
    // which could be as small as 0 or as large as manyItems-1.
    // If target is not in the array, then index will be set equal to manyItems.
    index = 0;
    while ((index < manyItems) && (target != data[index]))
        index++;

    if (index == manyItems)
        // The target was not found, so nothing is removed.
        return false;
    else
    { // The target was found at data[index].
        manyItems--;
        data[index] = data[manyItems]; ← See Self-Test Exercise 22 for an
        return true;
    }
}
```

The boolean expression indicates that the loop continues as long as `index` is still a location in the used part of the array (i.e., `index < manyItems`) and we have not yet found the target (i.e., `target != data[index]`). Each time through the loop, the `index` is incremented by one. No other work is needed in the loop. But take a careful look at the expression `data[index]` in the boolean test of the loop. The valid indexes for `data` range from 0 to `manyItems-1`. But if the target is not in the array, then `index` will eventually reach `manyItems`, which could be an invalid index. At that point, with `index` equal to `manyItems`, we must not evaluate the expression `data[index]`. Trying to evaluate `data[index]` with an invalid `index` will cause an `ArrayIndexOutOfBoundsException`.

**Using an Array Index in a Boolean Expression**

Never use an invalid index, even in a simple test.

Avoiding the invalid index is the reason for the first part of the boolean test (i.e., `index < manyItems`). Moreover, the test for (`index < manyItems`) must appear *before* the other part of the test. Placing (`index < manyItems`) first ensures that only valid indexes are used. The insurance comes from a technique called *short-circuit evaluation*, which Java uses to evaluate boolean expressions. In **short-circuit evaluation**, a boolean expression is evaluated from left to right, and the evaluation stops as soon as there is enough information to determine the value of the expression. In our example, if `index` equals `manyItems`, then the first part of the boolean expression (`index < manyItems`) is `false`, so the entire `&&` expression *must* be `false`. It doesn't matter whether the second part of the `&&` expression is `true` or `false`. Therefore, Java doesn't bother to evaluate the second part of the expression, and the potential error of an invalid index is avoided.

*short-circuit  
evaluation of  
boolean  
expressions*

**The countOccurrences Method.** To count the number of occurrences of a particular element in a bag, we step through the used portion of the partially filled array. Remember that we are using locations `data[0]` through `data[manyItems-1]`, so the correct loop is shown in this implementation:

```
public int countOccurrences(int target)
{
    int answer;
    int index;

    answer = 0;
    for (index = 0; index < manyItems; index++)
        if (target == data[index])
            answer++;
    return answer;
}
```

*implementing  
the  
countOccurrences  
method*

**The union Method.** The `union` method is different from our other methods. It is a *static* method, which means it is not activated by any one bag object. Instead, the method must take its two parameters (bags `b1` and `b2`), combine these two bags together into a third bag, and return this third bag. The third bag is declared as a local variable called `answer` in the implementation of Figure 3.6. The capacity of the `answer` bag must be the sum of the capacities of `b1` and `b2`, so the actual `answer` bag is allocated by the statement:

*implementing  
the  
union  
method*

```
answer = new IntArrayBag(b1.getCapacity() + b2.getCapacity());
```

This calls the `IntArrayBag` constructor to create a new bag with an initial capacity of `b1.getCapacity() + b2.getCapacity()`.

The `union` implementation also makes use of the `System.arraycopy` method to copy elements from `b1.data` and `b2.data` into `answer.data`.

**FIGURE 3.6** Implementation of the Bag's union Method

### Implementation

```
public static IntArrayBag union(IntArrayBag b1, IntArrayBag b2)
{
    // If either b1 or b2 is null, then a NullPointerException is thrown.
    // In the case that the total number of items is beyond Integer.MAX_VALUE,
    // there will be an arithmetic overflow and the bag will fail.

    IntArrayBag answer =
        new IntArrayBag(b1.getCapacity() + b2.getCapacity());

    System.arraycopy(b1.data, 0, answer.data, 0, b1.manyItems);
    System.arraycopy(b2.data, 0, answer.data, b1.manyItems, b2.manyItems);
    answer.manyItems = b1.manyItems + b2.manyItems;

    return answer;
}
```

---

**The `clone` Method.** The `clone` method of a class allows a programmer to make a copy of an object. For example, the `IntArrayBag` class has a `clone` method to allow a programmer to make a copy of an existing bag. The copy is separate from the original so that subsequent changes to the copy won't change the original, nor will subsequent changes to the original change the copy.

The `IntArrayBag` `clone` method will follow the pattern introduced in Chapter 2 on page 82. Therefore, the start of the `clone` method is:

```
public IntArrayBag clone()
{ // Clone an IntArrayBag object.
    IntArrayBag answer;

    try
    {
        answer = (IntArrayBag) super.clone();
    }
    catch (CloneNotSupportedException e)
    {
        throw new RuntimeException
            ("This class does not implement Cloneable.");
    }
    ...
}
```

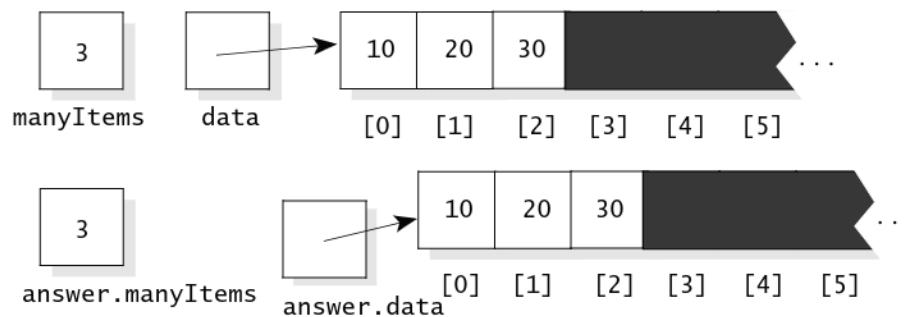


**136 Chapter 3 / Collection Classes**

As you can see, `answer.manyItems` has a copy of the number 3, and that is fine. But `answer.data` merely refers to the original's array. Subsequent changes to `answer.data` will affect the original and vice versa. This is incorrect behavior for a clone. To fix the problem, we need an additional statement before the return of the `clone` method. The purpose of the statement is to create a new array for the clone's `data` instance variable to refer to. Here's the statement:

```
answer.data = data.clone( );
```

After this statement, `answer.data` refers to a separate array:



The new `answer.data` array was created by creating a clone of the original array (as described on page 107). Subsequent changes to `answer` will not affect the original, nor will changes to the original affect `answer`. The complete `clone` method, including the extra statement at the end, is shown as part of the full implementation of Figure 3.7.

**PROGRAMMING TIP****CLONING A CLASS THAT CONTAINS AN ARRAY**

If a class has an instance variable that is an array, then the `clone` method needs extra work before it returns. The extra work creates a new array for the clone's instance variable to refer to.

The class may have other instance variables that are references to objects. In such a case, the `clone` method also carries out extra work. The extra work creates a new object for each such instance variable to refer to.

**The `ensureCapacity` Method.** This method ensures that a bag's array has at least a certain minimum length. Here is the method's heading:

```
public void ensureCapacity(int minimumCapacity)
```

The method determines whether the bag's array has a length below `minimumCapacity`. If so, the method allocates a new larger array with a length of `minimumCapacity`. The elements are copied into the larger array, and the `data` instance variable is then made to refer to the larger array. Part of Figure 3.7 shows our implementation, which follows the steps we have outlined.

### The Bag ADT—Putting the Pieces Together

Three bag methods remain to be implemented: `size` (which returns the number of elements currently in the bag), `getCapacity` (which returns the current length of the bag's array, including the part that's not currently being used), and `trimToSize` (which reduces the capacity of the bag's array to equal exactly the current number of elements in the bag).

The `size` and `getCapacity` methods are implemented in one line each, and `trimToSize` is similar to `ensureCapacity`, so we won't discuss these methods. But you should examine these methods in the complete implementation file of Figure 3.7. Also notice that the `IntArrayBag` class is placed in a package called `edu.colorado.collections`. Throughout the rest of this book, we will add other collection classes to this package.

---

**FIGURE 3.7** Implementation File for the `IntArrayBag` Class

#### Implementation

```
// File: IntArrayBag.java from the package edu.colorado.collections
// Complete documentation is in Figure 3.1 on page 119 or from the IntArrayBag link at
// http://www.cs.colorado.edu/~main/docs/.

package edu.colorado.collections;

public class IntArrayBag implements Cloneable
{
    // Invariant of the IntArrayBag class:
    // 1. The number of elements in the Bag is in the instance variable manyItems.
    // 2. For an empty Bag, we do not care what is stored in any of data;
    //     for a non-empty Bag, the elements in the Bag are stored in data[0]
    //     through data[manyItems-1], and we don't care what's in the rest of data.
    private int[] data;
    private int manyItems;

    public IntArrayBag()
    {
        final int INITIAL_CAPACITY = 10;
        manyItems = 0;
        data = new int[INITIAL_CAPACITY];
    }
}
```

(continued)

## 138 Chapter 3 / Collection Classes

(FIGURE 3.7 continued)

```
public IntArrayBag(int initialCapacity)
{
    if (initialCapacity < 0)
        throw new IllegalArgumentException
            ("initialCapacity is negative: " + initialCapacity);
    manyItems = 0;
    data = new int[initialCapacity];
}

public void add(int element)
{
    if (manyItems == data.length)
    {
        // Double the capacity and add 1; this works even if manyItems is 0. However, in
        // the case that manyItems*2 + 1 is beyond Integer.MAX_VALUE, there will be an
        // arithmetic overflow and the bag will fail.
        ensureCapacity(manyItems*2 + 1);
    }
    data[manyItems] = element;
    manyItems++;
}

public void addAll(IntArrayBag addend)
{
    // If addend is null, then a NullPointerException is thrown.
    // In the case that the total number of items is beyond Integer.MAX_VALUE, there will
    // be an arithmetic overflow and the bag will fail.
    ensureCapacity(manyItems + addend.manyItems);

    System.arraycopy(addend.data, 0, data, manyItems, addend.manyItems);
    manyItems += addend.manyItems;
}

public void addMany(int... elements)
{
    if (manyItems + elements.length > data.length)
    { // Ensure twice as much space as we need.
        ensureCapacity((manyItems + elements.length)*2);
    }

    System.arraycopy(elements, 0, data, manyItems, elements.length);
    manyItems += elements.length;
}
```

(continued)

(FIGURE 3.7 continued)

```
public IntArrayBag clone( )
{ // Clone an IntArrayBag object.
    IntArrayBag answer;

    try
    {
        answer = (IntArrayBag) super.clone( );
    }
    catch (CloneNotSupportedException e)
    {
        // This exception should not occur. But if it does, it would probably indicate a
        // programming error that made super.clone unavailable. The most common
        // error would be forgetting the "Implements Cloneable" clause at the start of
        // this class.
        throw new RuntimeException
            ("This class does not implement Cloneable.");
    }

    answer.data = data.clone( );

    return answer;
}

public int countOccurrences(int target)
{
    int answer;
    int index;

    answer = 0;
    for (index = 0; index < manyItems; index++)
        if (target == data[index])
            answer++;
    return answer;
}

public void ensureCapacity(int minimumCapacity)
{
    int[ ] biggerArray;

    if (data.length < minimumCapacity)
    {
        biggerArray = new int[minimumCapacity];
        System.arraycopy(data, 0, biggerArray, 0, manyItems);
        data = biggerArray;
    }
}
```

(continued)

**140 Chapter 3 / Collection Classes**

(FIGURE 3.7 continued)

```
public int getCapacity( )
{
    return data.length;
}

public boolean remove(int target)
{
    int index; // The location of target in the data array

    // First, set index to the location of target in the data array,
    // which could be as small as 0 or as large as manyItems-1.
    // If target is not in the array, then index will be set equal to manyItems.
    index = 0;
    while ((index < manyItems) && (target != data[index]))
        index++;

    if (index == manyItems)
        // The target was not found, so nothing is removed.
        return false;
    else
        { // The target was found at data[index].
            manyItems--;
            data[index] = data[manyItems]; ← See Self-Test Exercise 22 for an
            return true;
        }
    }

public int size( )
{
    return manyItems;
}

public void trimToSize( )
{
    int[ ] trimmedArray;

    if (data.length != manyItems)
    {
        trimmedArray = new int[manyItems];
        System.arraycopy(data, 0, trimmedArray, 0, manyItems);
        data = trimmedArray;
    }
}
```

See Self-Test Exercise 22 for an alternative approach to this step.

(continued)

(FIGURE 3.7 continued)

```
public static IntArrayBag union(IntArrayBag b1, IntArrayBag b2)
{
    // If either b1 or b2 is null, then a NullPointerException is thrown.
    // In the case that the total number of items is beyond Integer.MAX_VALUE, there will
    // be an arithmetic overflow and the bag will fail.
    IntArrayBag answer = new IntArrayBag(b1.getCapacity() + b2.getCapacity());

    System.arraycopy(b1.data, 0, answer.data, 0, b1.manyItems);
    System.arraycopy(b2.data, 0, answer.data, b1.manyItems, b2.manyItems);
    answer.manyItems = b1.manyItems + b2.manyItems;
    return answer;
}
```

---

### PROGRAMMING TIP

#### DOCUMENT THE ADT INVARIANT IN THE IMPLEMENTATION FILE

The invariant of an ADT describes the rules that dictate how the instance variables are used. This information is important to the programmer who implements the class. Therefore, you should write this information in the implementation file, just before the declarations of the private instance variables. For example, the invariant for the `IntArrayBag` class appears before the declarations of `manyItems` and `data` in the implementation file of Figure 3.7 on page 137.

This is the best place to document the ADT's invariant. In particular, do not write the invariant as part of the class's specification, because a programmer who uses the ADT does not need to know about private instance variables. But the programmer who implements the ADT does need to know about the invariant.

#### The Bag ADT—Testing

Thus far, we have focused on the design and implementation of new classes and their methods. But it's also important to continue practicing the other aspects of software development, particularly testing. Each of the bag's new methods must be tested. As shown in Chapter 1, it is important to concentrate the testing on boundary values. At this point, we will alert you to only one potential pitfall and will leave the complete testing to Programming Project 2 on page 169.

**FIGURE 3.8** Wrong Implementation of the Bag's addAll Method**A Wrong Implementation**

```
public void addAll(IntArrayBag addend)
{
    int i; // An array index
    ensureCapacity(manyItems + addend.manyItems);
    for (i = 0; i < addend.manyItems; i++)
        add(addend.data[i]);
}
```

**WARNING!**

*There is a bug in this implementation. See Self-Test Exercise 23.*

**PITFALL****AN OBJECT CAN BE AN ARGUMENT TO ITS OWN METHOD**

A class can have a method with a parameter that is the same data type as the class itself. For example, one of the IntArrayBag methods, addAll, has a parameter that is an IntArrayBag itself, as shown in this heading:

```
public void addAll(IntArrayBag addend)
```

An IntArrayBag can be created and activate its addAll method using itself as the argument. For example:

```
IntArrayBag b = new IntArrayBag();
b.add(5);           b now contains a 5 and a 2.
b.add(2);           ← Now b contains two 5s and two 2s.
b.addAll(b);       ←
```

The highlighted statement takes all the elements in b (the 5 and the 2) and adds them to what's already in b, so b ends up with two copies of each number.

In the highlighted statement, the bag b is activating the addAll method, but this same bag b is the actual argument to the method. This is a situation that must be carefully tested. As an example of the danger, consider the incorrect implementation of addAll in Figure 3.8. Do you see what goes wrong with `b.addAll(b)`? (See the answer to Self-Test Exercise 23.)

**The Bag ADT—Analysis**

We'll finish this section with a time analysis of the bag's methods. Generally, we'll use the number of elements in a bag as the input size. For example, if b is a bag containing  $n$  integers, then the number of operations required by `b.countOccurrences` is a formula involving  $n$ . To determine the operations, we'll see how many statements are executed by the method, although we won't need an exact determination since our answer will use big-O notation. Except for two declarations and two statements, all of the work in `countOccurrences`

happens in this loop:

```
for (index = 0; index < manyItems; index++)
    if (target == data[index])
        answer++;
```

We can see that the body of the loop will be executed exactly  $n$  times—once for each element in the bag. The body of the loop also has another important property: The body contains no other loops or calls to methods that contain loops. This is enough to conclude that the total number of statements executed by `countOccurrences` is no more than:

$$n \times (\text{number of statements in the loop}) + 4$$

The extra  $+4$  at the end is for the two declarations and two statements outside the loop. Regardless of how many statements are actually in the loop, the time expression is *always*  $O(n)$ —so the `countOccurrences` method is linear.

A similar analysis shows that `remove` is also linear, although `remove`'s loop sometimes executes fewer than  $n$  times. However, the fact that `remove` *sometimes* requires less than  $n \times (\text{number of statements in the loop})$  does not change the fact that the method is  $O(n)$ . In the worst case, the loop does execute a full  $n$  iterations; therefore, the correct time analysis is no better than  $O(n)$ .

**FIGURE 3.9** Time Analysis for the Bag Operations

| Operation                               | Time Analysis  |                                           | Operation          | Time Analysis  |                                          |
|-----------------------------------------|----------------|-------------------------------------------|--------------------|----------------|------------------------------------------|
| Constructor                             | $O(c)$         | $c$ is the initial capacity               | countOccurrences   | $O(n)$         | Linear time                              |
| add without capacity increase           | $O(1)$         | Constant time                             | ensureCapacity     | $O(c)$         | $c$ is the specified minimum capacity    |
| add with capacity increase              | $O(n)$         | Linear time                               | getCapacity        | $O(1)$         | Constant time                            |
| b1.addAll(b2) without capacity increase | $O(n_2)$       | Linear in the size of the added bag       | remove             | $O(n)$         | Linear time                              |
| b1.addAll(b2) with capacity increase    | $O(n_1 + n_2)$ | $n_1$ and $n_2$ are the sizes of the bags | size               | $O(1)$         | Constant time                            |
| clone                                   | $O(c)$         | $c$ is the bag's capacity                 | trimToSize         | $O(n)$         | Linear time                              |
|                                         |                |                                           | Union of b1 and b2 | $O(c_1 + c_2)$ | $c_1$ and $c_2$ are the bags' capacities |

**144 Chapter 3 / Collection Classes**

constant time  
 $O(1)$

The analysis of the constructor is a special case. The constructor allocates an array of `initialCapacity` integers, and in Java all array components are initialized (integers are set to zero). The initialization time is proportional to the capacity of the array, so an accurate time analysis is  $O(\text{initialCapacity})$ .

Several of the other bag methods do not contain any loops or array allocations. This is a pleasant situation because the time required for any of these methods does not depend on the number of elements in the bag. For example, when an element is added to a bag that does not need to grow, the new element is placed at the end of the array, and the `add` method never looks at the elements that were already in the bag. When the time required by a method does not depend on the size of the input, the procedure is called **constant time**, which is written  $O(1)$ .

The `add` method has two distinct cases. If the current capacity is adequate for a new element, then the time is  $O(1)$ . But if the capacity needs to be increased, then the time increases to  $O(n)$  because of the array allocation and copying of elements from the old array to the new array.

The time analyses of all methods are summarized in Figure 3.9.

**Self-Test Exercises for Section 3.2**

14. Draw a picture of `mybag.data` after these statements:

```
IntArrayBag mybag = new IntArrayBag(10);
mybag.add(1);
mybag.add(2);
mybag.addMany(1, 3, 4);
mybag.remove(1);
```
15. The bag in the preceding question has a capacity of 10. What happens if you try to add more than 10 elements to the bag?
16. Write the invariant of the bag ADT.
17. Which bag methods do not have the bag invariant as part of their implicit precondition?
18. Why don't we list the bag invariant as part of the explicit precondition and postcondition of the bag methods?
19. The `add` method uses `ensureCapacity(manyItems*2 + 1)`. What would go wrong if we forgot the `+1`?
20. Would the `add` method work correctly if we used `ensureCapacity(manyItems + 1)` without the `*2`?
21. What is the meaning of a *static* method? How is activation different from that of an ordinary method?
22. Use the expression `--manyItems` (with the `--` before `manyItems`) to rewrite the last two statements of `remove` (Figure 3.5 on page 132) as a single statement. If you are unsure of the difference between `manyItems--` and `--manyItems`, then go ahead and peek at our answer

at the back of this chapter. Use `manyItems++` to make a similar alteration to the `add` method.

23. Suppose we implement `addAll` as shown in Figure 3.8 on page 142. What goes wrong with `b.addAll(b)`?
24. Describe the extra work that must be done at the end of the `clone` method. Draw pictures to show what goes wrong if this step is omitted.
25. Use the `arraycopy` method to copy 10 elements from the front of an array `x` to the front of an array `y`.
26. Suppose `x` and `y` are arrays with 100 elements each. Use the `arraycopy` method to copy `x[10]...x[25]` to `y[33]...y[48]`.
27. Write a new bag method that removes all copies of a specified target from a bag (rather than removing just one copy of the target). The return value should be the number of copies of the target that were removed from the bag.
28. Write a variable-arity version of the `remove` method. The return value is the total number of elements removed from the bag. Your implementation should use the array version of a for-loop from page 111.
29. Write a static bag method called `intersection` that creates a new bag from two other bags `b1` and `b2`. The number of copies of an integer `x` in the new bag will always be the minimum of `b1.countOccurrences(x)` and `b2.countOccurrences(x)`.

### 3.3 PROGRAMMING PROJECT: THE SEQUENCE ADT

You are ready to tackle a collection class implementation on your own. The data type is called a **sequence**. A sequence is similar to a bag—both contain a bunch of elements. Unlike a bag, however, the elements in a sequence are arranged one after another.

How does this differ from a bag? After all, aren't the bag elements arranged one after another in the partially filled array that implements the bag? Yes, but that's a quirk of our particular bag implementation, and the order is just happenstance. Moreover, there is no way that a program using the bag can refer to the bag elements by their position in the array.

In contrast, the elements of a sequence are kept one after another, and the sequence's methods allow a program to step through the sequence one element at a time, using the order in which the elements are stored. Methods also permit a program to control precisely where elements are inserted and removed within the sequence.

*how a sequence  
differs from a  
bag*

### The Sequence ADT—Specification

Our bag happened to be a bag of *integers*. We could have had a different underlying element type, such as a bag of *double* numbers or a bag of *characters*. In fact, in Chapter 5, we'll see how to construct a collection that can simultaneously handle many different types of elements rather than being restricted to one type of element. But for now, our collection classes will have just one kind of element for each collection. In particular, for our sequence class, each element will be a *double* number, and the class itself is called `DoubleArrayListSeq`. We could have chosen some other type for the elements, but double numbers are as good as anything for your first implementation of a collection class.

As with the bag, each sequence will have a current capacity, which is the number of elements the sequence can hold without having to request more memory. The initial capacity will be set by the constructor. The capacity can be increased in several different manners, which we'll see as we specify the various methods of the new class.

**Constructor.** The `DoubleArrayListSeq` has two constructors—one that constructs an empty sequence with an initial capacity of 10 and another that constructs an empty sequence with some specified initial capacity.

**The `size` Method.** The `size` method returns the number of elements in the sequence. Here is the heading:

```
public int size()
```

10.1  
40.2  
1.1

For example, if `scores` is a sequence containing the values 10.1, 40.2, and 1.1, then `scores.size()` returns 3. Throughout our examples, we will draw sequences vertically with the first element on top, as shown in the picture in the margin (where the first element is 10.1).

**Methods to Examine a Sequence.** We will have methods to build a sequence, but it will be easier to explain first the methods to examine a sequence that has already been built. The elements of a sequence can be examined one after another, but the examination must be in order from the first to the last. Three methods work together to enforce the in-order retrieval rule. The methods' headings are:

```
public void start()
public double getCurrent()
public void advance()
```

When we want to retrieve the elements of a sequence, we begin by activating `start`. After activating `start`, the `getCurrent` method returns the first element of the sequence. Each time we call `advance`, the `getCurrent` method changes

so that it returns the next element of the sequence. For example, if a sequence called `numbers` contains the four numbers 37, 10, 83, and 42, then we can write the following code to print the first three numbers of the sequence:

```
numbers.start();
System.out.println(numbers.getCurrent());
numbers.advance();
System.out.println(numbers.getCurrent());
numbers.advance();
System.out.println(numbers.getCurrent());
```

Prints 37  
Prints 10  
Prints 83

start,  
getCurrent,  
advance

isCurrent

One other method cooperates with `getCurrent`. The `isCurrent` method returns a boolean value to indicate whether there actually is a current element for `getCurrent` to provide or whether we have advanced right off the end of the sequence.

Using all four of the methods with a for-loop, we can print an entire sequence, as shown here for the `numbers` sequence:

```
for (numbers.start(); numbers.isCurrent(); numbers.advance())
    System.out.println(numbers.getCurrent());
```

**The `addBefore` and `addAfter` Methods.** There are two methods to add a new element to a sequence, with these headers:

```
public void addBefore(double element)
public void addAfter(double element)
```

The first method, `addBefore`, places a new element before the current element. For example, suppose that we have created the sequence shown in the margin and that the current element is 8.8. In this example, we want to add 10.0 to our sequence, immediately before the current element. When 10.0 is added before the current element, other elements in the sequence—such as 8.8 and 99.0—will move down in the sequence to make room for the new element. After the addition, the sequence has the four elements shown in the lower box.

If there is no current element, then `addBefore` places the new element at the front of the sequence. In any case, after the `addBefore` method returns, the new element will be the current element. In the example shown in the margin, 10.0 becomes the new current element.

The second method, `addAfter`, also adds a new element to a sequence, but the new element is added *after* the current element. If there is no current element, then the `addAfter` method places the new element at the end of the sequence (rather than at the front). In all cases, when the method finishes, the new element will be the current element.

Either `addBefore` or `addAfter` can be used on an empty sequence to add the first element.

|      |
|------|
| 42.1 |
| 8.8  |
| 99.0 |

The sequence grows by adding 10.0 before the current element.

|      |
|------|
| 42.1 |
| 10.0 |
| 8.8  |
| 99.0 |



A concatenation is somewhat similar to the union of two bags. For example:

```
DoubleArrayList part1 = new DoubleArrayList();
DoubleArrayList part2 = new DoubleArrayList();

part1.addAfter(3.7);
part1.addAfter(9.5);
part2.addAfter(4.0);
part2.addAfter(8.6);  
    This computes the concatenation  
    of the two sequences, putting the  
    result in a third sequence.

DoubleArrayList total = DoubleArrayList.concat(part1, part2);
```

After these statements, `total` is the sequence consisting of 3.7, 9.5, 4.0, and 8.6. The new sequence computed by `concat` has no current element. The original sequences, `part1` and `part2`, are unchanged.

Notice the effect of having a *static* method: `concat` is not activated by any one sequence. Instead, the activation of

```
DoubleArrayList.concat(part1, part2)
```

creates and returns a new sequence that includes the contents of `part1` followed by the contents of `part2`.

**The `clone` Method.** As part of our specification, we require that a sequence can be copied with a `clone` method. The clone contains the same elements as the original. If the original had a current element, then the clone has a current element in the corresponding place. For example:

```
DoubleArrayList s = new DoubleArrayList();
s.addAfter(4.2);
s.addAfter(1.5);
s.start();
IntArrayBag t = (DoubleArrayList) s.clone();
```

At the point when the clone is made, the sequence `s` has two elements (4.2 and 1.5), and the current element is the 4.2. Therefore, `t` will end up with the same two elements (4.2 and 1.5), and its current element will be the number 4.2. Subsequent changes to `s` will not affect `t`, nor vice versa.

**Three Methods That Deal with Capacity.** The sequence class has three methods for dealing with capacity—the same three methods that the bag has:

```
public int getCapacity()
public void ensureCapacity(int minimumCapacity)
public void trimToSize()
```

**150 Chapter 3 / Collection Classes**

As with the bag, the purpose of these methods is to allow a programmer to explicitly set the capacity of the collection. If a programmer does not explicitly set the capacity, then the class will still work correctly, but some operations will be less efficient because the capacity might be repeatedly increased.

**The Sequence ADT—Documentation**

The complete specification for this first version of our sequence class is shown in Figure 3.10. This specification is also available from the [DoubleArrayList](#) link at the following web address:

<http://www.cs.colorado.edu/~main/docs/>

When you read the specification, you'll see that the package name is `edu.colorado.collections`. So you should create a subdirectory called `edu/colorado/collections` for your implementation.

The specification also indicates some limitations—the same limitations that we saw for the bag class. For example, an `OutOfMemoryError` can occur in any method that increases the capacity. Several of the methods throw an `IllegalStateException` to indicate that they have been illegally activated (with no current element). Also, an attempt to move the capacity beyond the maximum integer causes the class to fail by an arithmetic overflow.

After you've looked through the specifications, we'll suggest a design that uses three private instance variables.

**The Sequence ADT—Design**

Our suggested design for the sequence ADT has three private instance variables. The first variable, `data`, is an array that stores the elements of the sequence. Just like the bag, `data` is a partially filled array, and a second instance variable, called `manyItems`, keeps track of how much of the `data` array is currently being used. Therefore, the used part of the array extends from `data[0]` to `data[manyItems-1]`. The third instance variable, `currentIndex`, gives the index of the current element in the array (if there is one). Sometimes a sequence has no current element, in which case `currentIndex` will be set to the same number as `manyItems` (since this is larger than any valid index). The complete invariant of our ADT is stated as three rules:

1. The number of elements in the sequence is stored in the instance variable `manyItems`.
2. For an empty sequence (with no elements), we do not care what is stored in any of `data`; for a nonempty sequence, the elements of the sequence are stored from the front to the end in `data[0]` to `data[manyItems-1]`, and we don't care what is stored in the rest of `data`.
3. If there is a current element, then it lies in `data[currentIndex]`; if there is no current element, then `currentIndex` equals `manyItems`.

**FIGURE 3.10** Specification for the DoubleArraySeq Class

### Class DoubleArraySeq

#### ◆ public class DoubleArraySeq from the package edu.colorado.collections

A DoubleArraySeq keeps track of a sequence of double numbers. The sequence can have a special “current element,” which is specified and accessed through four methods that are not available in the bag class (`start`, `getCurrent`, `advance`, and `isCurrent`).

#### **Limitations:**

- (1) The capacity of a sequence can change after it’s created, but the maximum capacity is limited by the amount of free memory on the machine. The constructor, `addAfter`, `addBefore`, `clone`, and `concatenation` will result in an `OutOfMemoryError` when free memory is exhausted.
- (2) A sequence’s capacity cannot exceed the largest integer, 2,147,483,647 (`Integer.MAX_VALUE`). Any attempt to create a larger capacity results in failure due to an arithmetic overflow.

### Specification

#### ◆ Constructor for the DoubleArraySeq

```
public DoubleArraySeq()
```

Initialize an empty sequence with an initial capacity of 10. Note that the `addAfter` and `addBefore` methods work efficiently (without needing more memory) until this capacity is reached.

#### **Postcondition:**

This sequence is empty and has an initial capacity of 10.

#### **Throws:** `OutOfMemoryError`

Indicates insufficient memory for new `double[10]`.

#### ◆ Second Constructor for the DoubleArraySeq

```
public DoubleArraySeq(int initialCapacity)
```

Initialize an empty sequence with a specified initial capacity. Note that the `addAfter` and `addBefore` methods work efficiently (without needing more memory) until this capacity is reached.

#### **Parameter:**

`initialCapacity` – the initial capacity of this sequence

#### **Precondition:**

`initialCapacity` is non-negative.

#### **Postcondition:**

This sequence is empty and has the given initial capacity.

#### **Throws:** `IllegalArgumentException`

Indicates that `initialCapacity` is negative.

#### **Throws:** `OutOfMemoryError`

Indicates insufficient memory for new `double[initialCapacity]`.

(continued)

**152 Chapter 3 / Collection Classes***(FIGURE 3.10 continued)***◆ addAfter and addBefore**

```
public void addAfter(double element)
public void addBefore(double element)
```

Adds a new element to this sequence, either before or after the current element. If this new element would take this sequence beyond its current capacity, then the capacity is increased before adding the new element.

**Parameter:**

element – the new element that is being added

**Postcondition:**

A new copy of the element has been added to this sequence. If there was a current element, then addAfter places the new element after the current element, and addBefore places the new element before the current element. If there was no current element, then addAfter places the new element at the end of this sequence, and addBefore places the new element at the front of this sequence. In all cases, the new element becomes the new current element of this sequence.

**Throws: OutOfMemoryError**

Indicates insufficient memory to increase the size of this sequence.

**Note:**

An attempt to increase the capacity beyond Integer.MAX\_VALUE will cause this sequence to fail with an arithmetic overflow.

**◆ addAll**

```
public void addAll(DoubleArrayList addend)
```

Place the contents of another sequence at the end of this sequence.

**Parameter:**

addend – a sequence whose contents will be placed at the end of this sequence

**Precondition:**

The parameter, addend, is not null.

**Postcondition:**

The elements from addend have been placed at the end of this sequence. The current element of this sequence remains where it was, and the addend is also unchanged.

**Throws: NullPointerException**

Indicates that addend is null.

**Throws: OutOfMemoryError**

Indicates insufficient memory to increase the capacity of this sequence.

**Note:**

An attempt to increase the capacity beyond Integer.MAX\_VALUE will cause this sequence to fail with an arithmetic overflow.

(continued)

(FIGURE 3.10 continued)

◆ **advance**

```
public void advance()
```

Move forward so that the current element is now the next element in this sequence.

**Precondition:**

`isCurrent()` returns `true`.

**Postcondition:**

If the current element was already the end element of this sequence (with nothing after it), then there is no longer any current element. Otherwise, the new element is the element immediately after the original current element.

**Throws:** `IllegalStateException`

Indicates that there is no current element, so `advance` may not be called.

◆ **clone**

```
public DoubleArrayList clone()
```

Generate a copy of this sequence.

**Returns:**

The return value is a copy of this sequence. Subsequent changes to the copy will not affect the original, nor vice versa. The return value must be typecast to a `DoubleArrayList` before it is used.

**Throws:** `OutOfMemoryError`

Indicates insufficient memory for creating the clone.

◆ **concatenation**

```
public static DoubleArrayList concatenation  
(DoubleArrayList s1, DoubleArrayList s2)
```

Create a new sequence that contains all the elements from one sequence followed by another.

**Parameters:**

`s1` – the first of two sequences

`s2` – the second of two sequences

**Precondition:**

Neither `s1` nor `s2` is null.

**Returns:**

a new sequence that has the elements of `s1` followed by the elements of `s2` (with no current element)

**Throws:** `NullPointerException`

Indicates that one of the arguments is null.

**Throws:** `OutOfMemoryError`

Indicates insufficient memory for the new sequence.

**Note:**

An attempt to increase the capacity beyond `Integer.MAX_VALUE` will cause this sequence to fail with an arithmetic overflow.

(continued)

**154 Chapter 3 / Collection Classes***(FIGURE 3.10 continued)***◆ ensureCapacity**

```
public void ensureCapacity(int minimumCapacity)
```

Change the current capacity of this sequence.

**Parameter:**

minimumCapacity – the new capacity for this sequence

**Postcondition:**

This sequence's capacity has been changed to at least minimumCapacity.

**Throws:** OutOfMemoryError

Indicates insufficient memory for new double[minimumCapacity].

**◆ getCapacity**

```
public int getCapacity()
```

Accessor method to determine the current capacity of this sequence. The addBefore and addAfter methods work efficiently (without needing more memory) until this capacity is reached.

**Returns:**

the current capacity of this sequence

**◆ getCurrent**

```
public double getCurrent()
```

Accessor method to determine the current element of this sequence.

**Precondition:**

isCurrent( ) returns true.

**Returns:**

the current element of this sequence

**Throws:** IllegalStateException

Indicates that there is no current element.

**◆ isCurrent**

```
public boolean isCurrent()
```

Accessor method to determine whether this sequence has a specified current element that can be retrieved with the getCurrent method.

**Returns:**

true (there is a current element) or false (there is no current element at the moment)

**◆ removeCurrent**

```
public void removeCurrent()
```

Remove the current element from this sequence.

**Precondition:**

isCurrent( ) returns true.

**Postcondition:**

The current element has been removed from this sequence, and the following element (if there is one) is now the new current element. If there was no following element, then there is now no current element.

**Throws:** IllegalStateException

Indicates that there is no current element, so removeCurrent may not be called. (continued)

(FIGURE 3.10 continued)

◆ **size**

```
public int size()
```

Accessor method to determine the number of elements in this sequence.

**Returns:**

the number of elements in this sequence

◆ **start**

```
public void start()
```

Set the current element at the front of this sequence.

**Postcondition:**

The front element of this sequence is now the current element (but if this sequence has no elements at all, then there is no current element).

◆ **trimToSize**

```
public void trimToSize()
```

Reduce the current capacity of this sequence to its actual size (i.e., the number of elements it contains).

**Postcondition:**

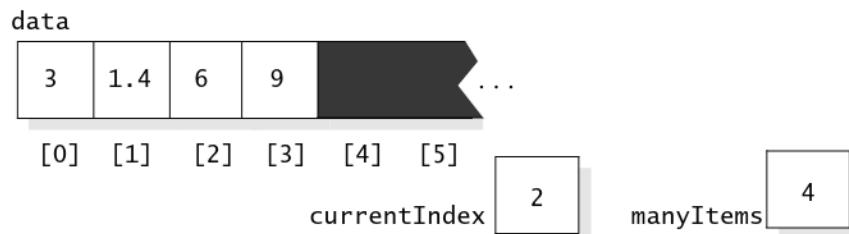
This sequence's capacity has been changed to its current size.

**Throws:** OutOfMemoryError

Indicates insufficient memory for altering the capacity.

---

As an example, suppose a sequence contains four numbers, with the current element at `data[2]`. The instance variables of the object might appear as shown here:



In this example, the current element is at `data[2]`, so the `getCurrent()` method would return the number 6. At this point, if we called `advance()`, then `currentIndex` would increase to 3, and `getCurrent()` would then return the 9.

Normally, a sequence has a current element, and the instance variable `currentIndex` contains the location of that current element. But if there is no current element, then `currentIndex` contains the same value as `manyItems`. In the preceding example, if `currentIndex` was 4, then that would indicate that there is no current element. Notice that this value (4) is beyond the used part of the array (which stretches from `data[0]` to `data[3]`).

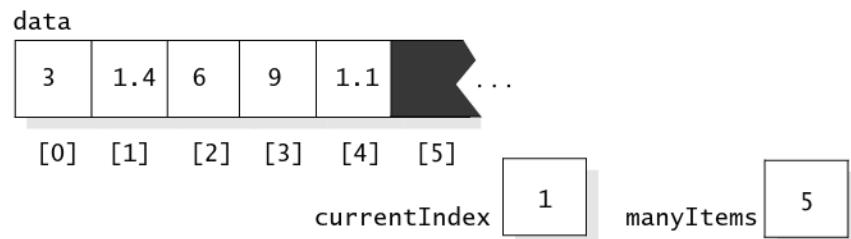
**156 Chapter 3 / Collection Classes**

*invariant of the ADT*

The stated requirements for the instance variables form the invariant of the sequence ADT. You should place this invariant at the top of your implementation file (`DoubleArrayList.java`). We will leave most of this implementation file up to you, but we will offer some hints and a bit of pseudocode.

### The Sequence ADT—Pseudocode for the Implementation

**The `removeCurrent` Method.** This method removes the current element from the sequence. First check that the precondition is valid (use `isCurrent()`). Then remove the current element by shifting each of the subsequent elements leftward one position. For example, suppose we are removing the current element from this sequence:



What is the current element in this picture? It is the 1.4 since `currentIndex` is 1 and `data[1]` contains 1.4.

In the case of the bag, we could remove an element such as 1.4 by copying the final element (1.1) onto the 1.4. But this approach won't work for the *sequence* because the elements would lose their sequence order. Instead, each element after the 1.4 must be moved leftward one position. The 6 moves from `data[2]` to `data[1]`; the 9 moves from `data[3]` to `data[2]`; and the 1.1 moves from `data[4]` to `data[3]`. This is a lot of movement, but a small for-loop suffices to carry out all the work. This is the pseudocode:

```
for (i = the index after the current element; i < manyItems; i++)
    Move an element from data[i] back to data[i-1];
```

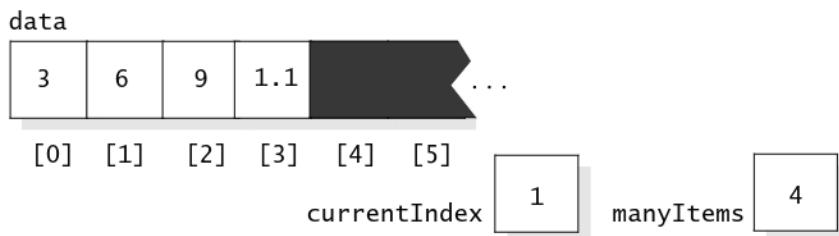
When the loop completes, you should reduce `manyItems` by one. The final result for our example is:



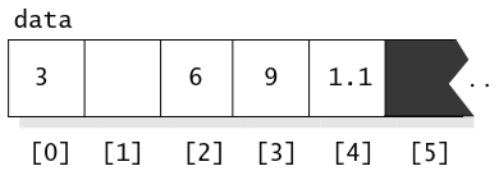
After the removal, the value in `currentIndex` is unchanged. In effect, this means that the element that was just after the removed element is now the current element. You must check that the method works correctly for boundary values—removing the first element and removing the end element. In fact, both these cases work fine. When the end element is removed, `currentIndex` will end up with the same value as `manyItems`, indicating that there is no longer a current element.

**The addBefore Method.** If there is a current element, then `addBefore` must take care to put the new element just before the current position. Elements that are already at or after the current position must be shifted rightward to make room for the new element. We suggest that you start by shifting elements at the end of the array rightward one position each until you reach the position for the new element.

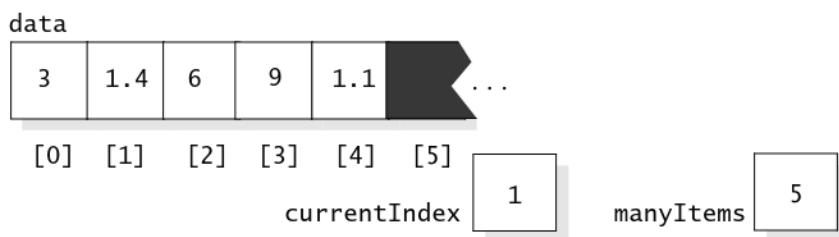
For example, suppose you are putting 1.4 at the location `data[1]` in this sequence:



You would begin by shifting the `1.1` rightward from `data[3]` to `data[4]`; then you'd move the `9` from `data[2]` to `data[3]`; then the `6` moves from `data[1]` rightward to `data[2]`. At this point, the array looks like this:



Of course, `data[1]` actually still contains a `6` because we just copied the `6` from `data[1]` to `data[2]`. But we have drawn `data[1]` as an empty box to indicate that `data[1]` is now available to hold the new element (the `1.4` that we are putting in the sequence). At this point, we can place the `1.4` in `data[1]` and add one to `manyItems`:



**158** Chapter 3 / Collection Classes

The pseudocode for shifting the elements rightward uses a for-loop. Each iteration of the loop shifts one element, as shown here:

```
for (i = manyItems; data[i] is the wrong spot for element ; i--)
    data[i] = data[i-1];
```

The key to the loop is the test `data[i]` is the wrong spot for `element`. How do we test whether a position is the wrong spot for the new element? A position is wrong when (`i > currentIndex`). Can you now write the entire method in Java? (See the solution to Self-Test Exercise 35 and don't forget to handle the special case when there is no current element.)

**Other Methods.** The other sequence methods are straightforward; for example, the `addAfter` method is similar to `addBefore`. Some additional useful methods are described in Programming Project 4 on page 169. You'll also need to be careful that you don't mindlessly copy the implementation of a bag method. For example, the `concatenation` method is similar to the bag's `union` method, but there is one extra step that concatenation must take. (It sets `currentIndex` to `manyItems`.)

### Self-Test Exercises for Section 3.3

30. What elements will be in the sequence `s` after these statements?

```
DoubleArrayListSeq s = new DoubleArrayListSeq();
s.addAfter(1);
s.addAfter(2);
s.start();
s.addAfter(3);
```
31. What are the instance variables for our sequence implementation?
32. Write some statements that declare a sequence and insert the numbers 1 through 100 into the sequence (in that order).
33. Suppose `x` is a double number (not an exact integer). Write some statements that insert `x` into the sequence from the previous exercise. The insertion should occur so that all the numbers of the sequence remain in order from smallest to largest.
34. Which of the sequence methods have implementations that could be identical to the bag methods?
35. Write the sequence's `addBefore` method.
36. Suppose a sequence has 24 elements, and there is no current element. According to the invariant of the ADT, what is `currentIndex`?
37. Suppose `g` is a sequence with 10 elements, and you activate `g.start()` and then activate `g.advance()` three times. What value is then in `g.currentIndex`?

38. What are good boundary values to test the `removeCurrent` method?
39. Write a demonstration program that asks the user for a list of family member ages and then prints the list in the same order it was given.
40. Write a new method to remove a specified element from a sequence. The method has one parameter (the element to remove).
41. For a sequence of numbers, suppose you insert 1, then 2, then 3, and so on, up to  $n$ . What is the big- $O$  time analysis for the combined time of inserting all  $n$  numbers with `addAfter`? How does the analysis change if you insert  $n$  first, then  $n-1$ , and so on, down to 1—always using `addBefore` instead of `addAfter`?
42. Which of the ADTs—the bag or the sequence—*must* be implemented by storing the elements in an array? (Hint: We are not beyond asking a trick question.)

### 3.4 PROGRAMMING PROJECT: THE POLYNOMIAL

A one-variable **polynomial** is an arithmetic expression of the form:

$$a_kx^k + \dots + a_2x^2 + a_1x^1 + a_0x^0$$

The highest exponent,  $k$ , is called the **degree** of the polynomial, and the constants  $a_0, a_1, \dots$  are the **coefficients**. For example, here is a polynomial with degree three:

$$0.3x^3 + 0.5x^2 + (-0.9)x^1 + 1.0x^0$$

Each individual **term** of a polynomial consists of a real number as a coefficient (such as 0.3), the variable  $x$ , and a non-negative integer as an **exponent**. The  $x^1$  term is usually written with just an  $x$  rather than  $x^1$ ; the  $x^0$  term is usually written with just the coefficient (since  $x^0$  is always defined to be 1); and a negative coefficient may also be written with a subtraction sign, so another way to write the same polynomial is:

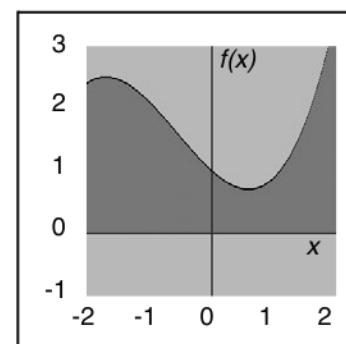
$$0.3x^3 + 0.5x^2 - 0.9x + 1.0$$

For any specific value of  $x$ , a polynomial can be evaluated by plugging the value of  $x$  into the expression. For example, the value of the sample polynomial at  $x = 2$  is:

$$0.3(2)^3 + 0.5(2)^2 - 0.9(2) + 1.0$$

A typical algebra exercise is to plot the graph of a polynomial for each value of  $x$  in a given range. For example, Figure 3.11 plots the value of a polynomial for each  $x$  in the range of  $-2$  to  $+2$ .

**FIGURE 3.11** A Polynomial



The graph of the function  $f(x)$  defined by the polynomial  $0.3x^3 + 0.5x^2 - 0.9x + 1.0$

**160 Chapter 3 / Collection Classes**

For this project, you should specify, design, and implement a class for polynomials. The coefficients are double numbers, and the exponents are non-negative integers. The coefficients should be stored in an array of double numbers, with the coefficient for the  $x^k$  term stored in location [k] of the array. The maximum index of the array needs to be at least as big as the degree of the polynomial so that the largest nonzero coefficient can be stored. For the example polynomial  $0.3x^3 + 0.5x^2 - 0.9x + 1.0$ , the start of the coefficient array contains these numbers:

|     |      |     |     |     |
|-----|------|-----|-----|-----|
| 1.0 | -0.9 | 0.5 | 0.3 | ... |
| [0] | [1]  | [2] | [3] | [4] |

In addition, the class should have an instance variable to keep track of the current degree of the polynomial. (You could manage without the degree variable, but having it around makes certain operations more efficient.)

The rest of this section lists some methods that you could provide to the class.

**Constructors.**

```
public Polynomial( )           // No-arguments constructor  
public Polynomial(double a0)   // Set the  $x^0$  coefficient only  
public Polynomial(Polynomial source) // Copy constructor
```

The no-arguments constructor creates a polynomial with all zero coefficients. The second constructor creates a polynomial with the specified parameter as the coefficient of the  $x^0$  term, and all other coefficients are zero. For example:

```
Polynomial p = new Polynomial(4.2);
```

After this declaration, p has only one nonzero term,  $4.2x^0$ , which is the same as the number 4.2 (since  $x^0$  is defined as equal to 1).

The third constructor will create a polynomial that is a copy of another polynomial (called the “source”).

**Clone Method.**

```
public Polynomial clone( )
```

This is the usual `clone` method that returns a copy of the object that activates it.

**Modification Methods.**

```
public void add_to_coef(double amount, int k)  
public void assign_coef(double new_coefficient, int k)  
public void clear( )  
public void reserve(int degree)
```



### Self-Test Exercises for Section 3.4

43. Suppose you implement the polynomial class using an array called `coef`. What values will be stored in `coef` for the polynomial  $0.3x^3 + 0.5x^2 - 0.9x^1 + 1.0$ ?
44. What is your guess for the running time required for the three arithmetic operators of addition, subtraction, and multiplication?

## 3.5 THE JAVA HASHSET AND ITERATORS

This section provides a first introduction to one of Java's collection classes—the `HashSet`—including a feature called the *iterator*, which permits a programmer to easily step through all the elements of a collection.

### The HashSet Class

`java.util.HashSet`

A `HashSet` (from `java.util.HashSet`) is one of several container classes that are part of the Java Class Libraries. Its primary purpose is to store a collection of Java objects, such as a collection of `String` objects. Unlike our bag, a `HashSet` stores references to objects rather than storing primitive values (such as `int` values). Also, a `HashSet` cannot contain two objects that are equal; a second addition of an object that is equal to one that's already in a `HashSet` will have no effect.

We'll start with a small example that creates a `HashSet` of strings:

```
HashSet<String> heroes = new HashSet<String>();
heroes.add("Hiro");
heroes.add("Claire");           After these statements, heroes contains
heroes.add("Peter");
heroes.add("Hiro");           only three strings, since the same
                             string cannot be added twice.
```

The name of the data type is `HashSet`, but this name is augmented by `<String>` to indicate the type of elements that will reside in the set. This augmentation is called a *generic type instantiation*, and it differs from the way that we specified the underlying type for our own bag. We will learn how to write such *generic classes* of our own in Chapter 5, but for now we merely want to *use* a `HashSet` of strings, so we don't need to know how it is implemented.

Notice that in the example we tried to add four strings, but only the first three were added. That's because we cannot add the same string ("Hiro") more than once to a single `HashSet`. The `HashSet`'s notion of "same string" is determined by using the `String`'s `equals` method (see page 77).

### Some of the HashSet Members

**Constructors.** A default constructor creates an empty `HashSet`; a second constructor makes a copy of an existing `HashSet` (or, in fact, certain other kinds of

collections). Remember that when you declare a `HashSet`, you must specify the type of elements that reside in the set, such as the `String` in `HashSet<String>`.

**Members That Are Similar to the Bag.** These members are similar to our bag:

```
boolean add(E element);  
boolean remove(E element);  
int size();
```

*E must be the data type of the elements in the HashSet*

The `HashSet`'s `add` method can be used exactly like the bag's `add` method to add an item to a `HashSet`. However, the `HashSet`'s method has a boolean return value, which will be true in the case that the added element was not previously in the set and false otherwise. (In the false case, a second copy is not added.)

## Iterators

An **iterator** is an object from `java.util.Iterator` that permits a programmer to easily step through all the items in a container, examining the items and (perhaps) changing them. Each of the collection classes in the Java Class Libraries has a standard method called `iterator` that returns an iterator providing access to the items in the container. The iterator itself is also a Java object with three methods that are illustrated in this small piece of code:

`java.util.Iterator`

```
// Declare a HashSet of Strings and an Iterator for that HashSet:  
HashSet<String> heroes = new HashSet<String>();  
Iterator<String> it;  
  
// Put three elements in the HashSet:  
heroes.add("Hiro");  
heroes.add("Claire");  
heroes.add("Peter");  
  
// Use the iterator to print all the Strings in the HashSet:  
it = heroes.iterator();  
while (it.hasNext())  
{  
    System.out.println(it.next());  
}
```

*the iterator, hasNext, and next methods*

The example shows a widely used pattern described here:

1. After the elements are all placed in the collection, call the `iterator` method of the collection to create an iterator object. The example does this with the statement `it = heroes.iterator();`.

**164 Chapter 3 / Collection Classes**

2. The iterator is now set up to step through the collection's elements one after another. The iterator's `hasNext` method controls the step-through process by returning `true` if there are more elements to step through. After the last element has been stepped through, `hasNext` returns `false`.
3. The iterator's `next` method returns a reference to one element in the collection; each time `next` is activated, it will return a different element from the collection, until eventually all the elements have been provided (at which point `hasNext` will return `false`). The precondition for `next` requires that `hasNext` returns `true`.

**PITFALL****DO NOT ACCESS AN ITERATOR'S NEXT ITEM WHEN HASNEXT IS FALSE**

It is a programming error to activate `it.next( )` when `it.hasNext( )` returns `false`.

Here is the general pattern that you can use for an iterator `it` and a collection object `c`:

```
it = c.iterator();
while (it.hasNext())
{
    ...statements to access the item it.next()
}
```

**Invalid Iterators**

After an iterator has been set, it can easily move through its collection. However, changes to the collection—either additions or removals—can cause all of the collection's iterators to become invalid. When an iterator becomes invalid because of a change to its container, that iterator can no longer be used until it is assigned a new value.

**PITFALL****CHANGING A CONTAINER OBJECT CAN INVALIDATE ITS ITERATORS**

When an iterator's underlying container changes (by an addition or a removal), the iterator generally becomes invalid. Unless the class documentation says otherwise, that iterator should no longer be used until it is reassigned a new value from the changed container.

For now, we've seen enough of Java's generic collection classes and their iterators. We'll return to them in detail in Chapter 5.

### Self-Test Exercises for Section 3.5

45. What are the primary differences between a bag and a HashSet?
46. In general, how does an iterator become invalid?
47. Write a static method that has one parameter: a non-empty HashSet of strings. The return value is the average of the strings' lengths.

## CHAPTER SUMMARY

- A *collection class* is an ADT in which each object contains a collection of elements. Bags and sequences are two examples of collection classes.
- The simplest implementations of collection classes use a *partially filled array*. Using a partially filled array requires each object to have at least two instance variables: the array itself and an `int` variable to keep track of how much of the array is being used.
- When a collection class is implemented with a partially filled array, the capacity of the array should grow as elements are added to the collection. The class should also provide explicit methods to allow a programmer to control the capacity of the array.
- In a collection class, some methods allocate additional memory (such as changing the capacity of an array). These methods have the possibility of throwing an `OutOfMemoryError` (when the machine runs out of memory).
- A class may have other instance variables that are references to objects or arrays. In such a case, the `clone` method must carry out extra work. The extra work creates a new object or array for each such instance variable to refer to.
- When you design an ADT, always make an explicit statement of the rules that dictate how the instance variables are used. These rules are called the *invariant of the ADT* and should be written at the top of the implementation file for easy reference.
- You don't have to write every collection class from scratch. The Java Class Libraries provide a variety of collection classes that are useful in many different settings (such as the `HashSet`). The `HashSet` requires the programmer to specify the specific generic type of the underlying elements, and it has iterators to step through the elements one after another.



## Solutions to Self-Test Exercises

1. Lowest is zero; highest is 41.
  2. 

```
int i;
int[ ] b;
b = new int[1000];
for (i = 1; i <= 1000; i++)
    b[i-1] = i;
```
  3. 

```
int[ ] b = new int[ ] { 1,2,3,4,5 };
```
  4. `b.length`
  5. Either `p[99] = 75` or `s[99] = 75`.
  6. Only `p[99] = 75`.
  7. 42 (since `a` and `b` refer to the same array)
  8. 0 (since `b` is a clone of `a`)
  9. `NullPointerException`
  10. The array referred to by the parameter in the method is the same as the array referred to by the actual argument. So the actual argument will have its first component changed to 42.
  11. 

```
public
void copyFront(int[] a, int[] b, int n)
// Precondition: a.length and b.length are
// both greater than or equal to n.
// Postcondition: n integers have been copied
// from the front of a to the front of b.
{
    int i;
    for (i = 0; i < n; i++)
        b[i] = a[i];
}
```
  12. 

```
public void zero_some(int[] x)
// Postcondition: Every even-index element
// of x has been changed to zero.
{
    int i;
    for (i = 0; i < x.length; i += 2)
        x[i] = 0;
}
```
- After `some_zero(a)`, these elements of `a` will be zero: `a[0]`, `a[2]`, and `a[4]`.

13. 

```
public static int count
(int[ ] a, int target)
{
    int answer = 0;

    for (int item : a)
        // Check whether item is target.
        if (item == target)
            answer++;
}

return answer;
}
```

14. 

|     |     |     |     |
|-----|-----|-----|-----|
| 4   | 2   | 1   | 3   |
| [0] | [1] | [2] | [3] |

 We don't care what appears beyond `data[3]`.
15. When the 11th element is added, the `add` method will increase the capacity to 21.
16. See the two rules on page 126.
17. The constructors
18. Because the programmer who uses the `bag` class does not need to know this information.
19. If `manyItems` happens to be zero, then `manyItems*2` would also be zero, and the capacity would not increase.
20. The capacity would always be increased by one (which is enough for the one new element), but increasing one by one will be inefficient.
21. A static method is not activated by any one particular object. It is activated by writing the class name, a dot, the method name, and the argument list. For example:  
`IntArrayBag.union(b1, b2)`

## Solutions to Self-Test Exercises 167

22. The two statements can be replaced by one:  
`data[index] = data[--manyItems];`. When  
`--manyItems` appears as an expression, the  
variable `manyItems` is decremented by one,  
and the resulting value is the value of  
the expression. (On the other hand, if  
`manyItems--` appears as an expression, the  
value of the expression is the value of  
`manyItems` prior to subtracting one.) Simi-  
larly, the last two statements of `add` can be  
combined to `data[manyItems++] = element;`.

23. For the incorrect implementation of `addAll`, suppose we have a bag `b` and we activate `b.addAll(b)`. Then the private instance variable `manyItems` is the same variable as `addend.manyItems`. Each iteration of the loop adds 1 to `manyItems`; hence `addend.manyItems` is also increasing, and the loop never ends.

One warning: Some collection classes in the Java libraries have an `addAll` method that fails for the statement `b.addAll(b)`. So, before you use an `addAll` method, check the specification for restrictions.

24. At the end of the `clone` implementation, we need an additional statement to make a separate copy of the data array for the clone to use. If we don't make this copy, then our own data array and the clone's data array will be one and the same (see the pictures on page 135).

25. `System.arraycopy(x, 0, y, 0, 10);`

26. `System.arraycopy(x, 10, y, 33, 16);`

27. `public int removeAll(int target)`  
{  
 int i;  
 int count = 0;  
 i = 0;  
 while (i < manyItems)  
 {  
 if (data[i] == target)  
 { // Delete data[i] by copying  
 // the last element on top of it:  
 manyItems--;  
 data[i]=data[manyItems];  
 }

```
        count++;
    }
    else
        i++;
}
return count;
}
```

28. `public int removeMany(int... targets)`  
{  
 int count = 0;

```
    for (int target : targets)
        count += remove(target);
    return count;
}
```

29. `public static`  
`IntArrayBag intersection`  
`(IntArrayBag b1, IntArrayBag b2)`  
{  
 int i, j;  
 int count;  
 int it;  
 IntArrayBag a;

`answer = new IntArrayBag;`

```
for (i = 0; i < b1.manyItems; i++)
{
    it = b1.data[i];
    if (a.countOccurrences(it)==0)
    {
        count = Math.min(
            b1.countOccurrences(it),
            b2.countOccurrences(it)
        );
        for (j = 0; j < count; j++)
        {
            a.add(it);
        }
    }
}
return a;
}
```

30. 1, 3, 2 (in that order). The current element is the 3.

31. `data, manyItems, currentIndex`

**168 Chapter 3 / Collection Classes**

32. DoubleArraySeq s;  
int i;  
s = new DoubleArraySeq(100);  
for (i = 1; i <= 100; i++)  
 s.addAfter(i);
  
33. s.start();  
while (  
 s.isCurrent()  
 &&  
 s.current() < x  
)  
 s.advance();  
if (s.isCurrent())  
 s.addBefore(x);  
else  
 s.addAfter(x);
  
34. addAll, getCapacity, size, trimToSize
  
35. void addBefore(double element)  
{  
 int i;  
  
 if (manyItems == data.length)  
 { // Try to double the capacity  
 ensureCapacity(manyItems\*2 + 1);  
 }  
  
 if (!isCurrent())  
 currentIndex = 0;  
 for  
(i=manyItems; i>currentIndex; i--)  
 data[i] = data[i-1];  
 data[currentIndex] = element;  
 manyItems++;  
}
  
36. 24
  
37. g.currentIndex will be 3 (since the 4<sup>th</sup> element occurs at data[3]).
  
38. The removeCurrent method should be tested when the sequence's size is just 1 and when the sequence is at its full capacity. At full capacity, you should try removing the first element and the last element of the sequence.

39. Your program can be similar to Figure 3.2 on page 123.

40. Here is our method's heading, with a postcondition:

```
void remove(int target);  
// Postcondition: If target was in the  
// sequence, then the first copy of target  
// has been removed, and the element after  
// the removed element (if there is one)  
// becomes the new current element; other-  
// wise the sequence remains unchanged.
```

The easiest implementation searches for the index of the target. If this index is found, then set `currentIndex` to this index and activate the ordinary `removeCurrent` method.

41. The total time to add 1, 2, ...,  $n$  with `addAfter` is  $O(n)$ . The total time to add  $n, n-1, \dots, 1$  with `addBefore` is  $O(n^2)$ . The larger time for the second approach is because an addition at the front of the sequence requires all of the existing elements to be shifted right to make room for the new element. Hence, on the second addition, one element is shifted. On the third addition, two elements are shifted. (And so on to the  $n^{\text{th}}$  element, which needs  $n-1$  shifts.) The total number of shifts is  $1 + 2 + \dots + (n-1)$ , which is  $O(n^2)$ . To show that this sum is  $O(n^2)$ , use a technique similar to Figure 1.2 on page 20.

42. Neither of the classes *must* use an array. In later chapters, we will see both classes implemented without arrays.

43. `coef[0] = 1.0; coef[1] = -0.9; coef[2] = 0.5; coef[3] = 0.3;` the rest of `coef` is all zero.

44. Addition and subtraction are  $O(n)$ , where  $n$  is the larger degree of the two polynomials. Multiplication is  $O(m \times n)$ , where  $m$  and  $n$  are the degrees of the polynomials. (This assumes that  $m$  and  $n$  are nonzero.)

45. The bag class allows duplicate values in the container, whereas the `HashSet` class requires

unique values; also, the data type of the elements in the `HashSet` is determined by its generic type parameter.

46. By changing its underlying collection
47. Here is one solution. We assume that `java.util.HashSet` and `java.util.Iterator` have been imported:

```
public static
double ave_len(HashSet<String> c)
{
    Iterator<String> it;
    double total = 0;
```

```
if (c.size( ) == 0)
    throw new
IllegalArgumentException
("Empty HashSet!");

it = c.iterator( );
while (it.hasNext( ))
{
    total += it.next( ).length( );
}
return total/c.size( );
}
```

## PROGRAMMING PROJECTS



**1** For the `IntArrayBag` class, implement a new method called `equals` with a boolean return value and one parameter. The parameter, called `b`, is another `IntArrayBag`. The method returns `true` if `b` and the bag that activates the method have exactly the same number of every element. Otherwise, the method returns `false`. Notice that the locations of the elements in the data arrays are not necessarily the same. It is only the number of occurrences of each element that must be the same.

The worst-case time for the method should be  $O(mn)$ , where  $m$  is the size of the bag that activates the method and  $n$  is the size of `b`.

**2** A **black box** test of a class is a program that tests the correctness of a class without directly examining the private instance variables of the class. You can imagine that the private instance variables are inside an opaque black box where they cannot be seen, so all testing must occur only through activating the public methods.

Write a noninteractive black box test program for the `IntArrayBag` class. Make sure you test the boundary values, such as an empty bag, a bag with just one element, and a full bag.

**3** Study the `BagApplet` from Appendix I and write an expanded version that has three bags and buttons to activate any method of any bag. Also include a button that will carry out an action such as:

```
b1 = IntArrayBag.union(b2, b3).
```

**4** Implement the sequence class from Section 3.3. You may wish to provide some additional useful methods, such as:

(1) a method to add a new element at the front of the sequence; (2) a method to remove the element at the front of the sequence; (3) a method to add a new element at the end of the sequence; (4) a method that makes the last element of the sequence become the current element; (5) a method that returns the  $i^{\text{th}}$  element of the sequence (starting with the  $0^{\text{th}}$  at the front); and (6) a method that makes the  $i^{\text{th}}$  element become the current element.

**5** Using Appendix I as a guide, implement an applet for interactive testing of the sequence class from the previous project.

## 170 Chapter 3 / Collection Classes

**6** A bag can contain more than one copy of an element. For example, this chapter describes a bag that contains the number 4 and two copies of the number 8. This bag behavior is different from a set, which can contain only a single copy of any given element. Write a new collection class called `IntArraySet`, which is similar to a bag except that a set can contain only one copy of any given element. You'll need to change the specification a bit. For example, instead of the bag's `countOccurrences` method, you'll want a method such as this:

```
boolean contains(int target)
// Postcondition: The return value is true if
// target is in the set; otherwise, the return
// value is false.
```

Make an explicit statement of the invariant of the set ADT. Do a time analysis for each operation. At this point, an efficient implementation is not needed. For example, just adding a new element to a set will take linear time because you'll need to check that the new element isn't already present. Later we'll explore more efficient implementations.

You may also want to add additional methods to your set ADT, such as a method for subtracting one set from another.

**7** Rewrite the sequence class using a new class name, `DoubleArrayListSeq`. In the new class, the `add` method always puts the new element so that all the elements stay in order from smallest to largest. There is no `addBefore` or `addAfter` method. All the other methods are the same as in the original sequence ADT.

**8** A one-variable **polynomial** is an arithmetic expression of the form:

$$a_0 + a_1x + a_2x^2 + \dots + a_kx^k$$

Implement the polynomial class described in Section 3.4.

**9** Specify, design, and implement a class that can be one player in a game of tic-tac-toe. The constructor should specify whether the

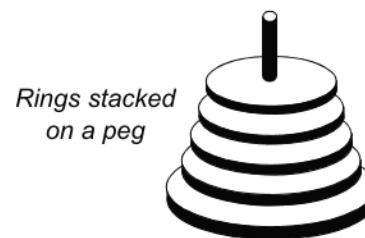
object is to be the first player (Xs) or the second player (Os). There should be a method to ask the object to make its next move and a method that tells the object what the opponent's next move is. Also include other useful methods, such as a method to ask whether a given spot of the tic-tac-toe board is occupied and, if so, whether the occupation is with an X or an O. Also include a method to determine when the game is over and whether it was a draw, an X win, or an O win.

Use the class in two programs: a program that plays tic-tac-toe against the program's user and a program that has two tic-tac-toe objects that play against each other.

**10** Specify, design, and implement a collection class that can hold up to five playing cards.

Call the class `PokerHand`, and include a method with a boolean return value to allow you to compare two poker hands. For two hands `x` and `y`, the relation `x.beats(y)` means that `x` is a better hand than `y`. If you do not play in a weekly poker game yourself, you may need to consult a card rule book for the rules on the ranking of poker hands.

**11** Specify, design, and implement a class that keeps track of rings stacked on a peg, rather like phonograph records on a spindle. An example with five rings is shown here:



The peg may hold up to 64 rings, with each ring having its own diameter. Also, there is a rule that requires each ring to be smaller than any ring underneath it. The class's methods should include: (a) a constructor that places  $n$  rings on the peg (where  $n$  may be as large as 64), and these  $n$  rings have diameters from  $n$  inches on the bottom to one inch on the top; (b) an accessor method that returns the number

of rings on the peg; (c) an accessor method that returns the diameter of the topmost ring; (d) a method that adds a new ring to the top (with the diameter of the ring as a parameter to the method); (e) a method that removes the topmost ring; and (f) a method that prints some clever representation of the peg and its rings. Make sure that all methods have appropriate preconditions to guarantee that the rule about ring sizes is enforced. Also spend time designing appropriate private instance variables.

**12** In this project, you will design and implement a class called `Towers`, which is part of a program that lets a child play a game called Towers of Hanoi. The game consists of three pegs and a collection of rings that stack on the pegs. The rings are different sizes. The initial configuration for a five-ring game is shown here, with the first tower having rings from one inch (on the top) to five inches (on the bottom).



The rings are stacked in decreasing order of their size, and the second and third towers are initially empty. During the game, the child may transfer rings one at a time from the top of one peg to the top of another. The goal is to move all the rings from the first peg to the second peg. The difficulty is that the child may not place a ring on top of one with a smaller diameter. There is the one extra peg to hold rings temporarily, but the prohibition against a larger ring on a smaller ring applies to it as well as to the other two pegs. A solution for a three-ring game is shown at the top of the next page. The `Towers` class must keep track of the status of all three pegs. You might use an array of three pegs, where each peg is an object from the previous project. The `Towers` methods are specified here:

```
Towers(int n);
// Precondition: 1 <= n <= 64.
// Postcondition: The towers have been initialized
// with n rings on the first peg and no rings on
// the other two pegs. The diameters of the first
// peg's rings are from one inch (on the top) to n
// inches (on the bottom).
```

```
int countRings(int pegNumber)
// Precondition: pegNumber is 1, 2, or 3.
// Postcondition: The return value is the number
// of rings on the specified peg.
```

```
int getTopDiameter(int pegNumber)
// Precondition: pegNumber is 1, 2, or 3.
// Postcondition: If countRings(pegNumber) > 0,
// then the return value is the diameter of the top
// ring on the specified peg; otherwise, the return
// value is zero.
```

```
void move(int startPeg, int endPeg)
// Precondition: startPeg is a peg number
// (1, 2, or 3), and countRings(startPeg) > 0;
// endPeg is a different peg number (not equal
// to startPeg), and if endPeg has at least one
// ring, then getTopDiameter(startPeg) is
// less than getTopDiameter(endPeg).
// Postcondition: The top ring has been moved
// from startPeg to endPeg.
```

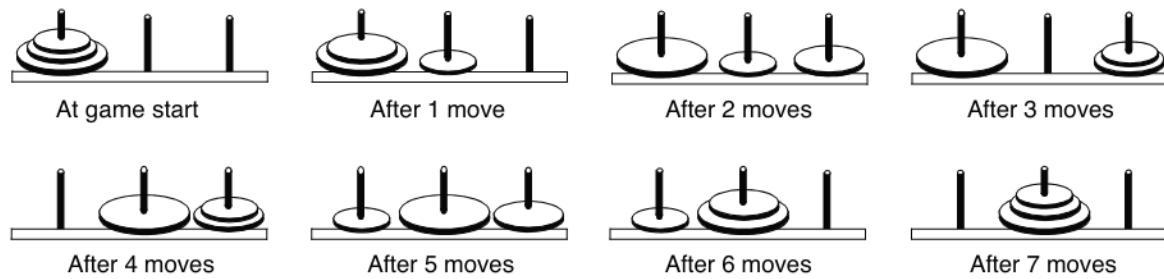
Also include a method so that a `Towers` object may be displayed easily.

Use the `Towers` object in a program that allows a child to play Towers of Hanoi. Make sure you don't allow the child to make any illegal moves.

**13** This project is to implement a class that is similar to Java's `Math.BigInteger` class. Each object in your class keeps track of an integer with an unlimited number of digits in base 10. The digits can be stored in an array of `int`, and the sign of the number can be stored in a separate instance variable, which is `+1` for a positive number and `-1` for a negative number.

The class should include several convenient constructors, such as a constructor to initialize an object from an ordinary `int`. Also write methods to carry out the usual arithmetic operators and comparison operators (to carry out arithmetic and comparisons on these big numbers).

## 172 Chapter 3 / Collection Classes



**14** Revise the bag class so that it is a bag of strings rather than integers. For the methods that add strings to the bag, you should always use the string's `clone` method to make a copy of each string and put the copy into the clone rather than the original. Also, when you are comparing two strings for equality, make sure you use the `String.equals` method rather than the `==` operator.

Modify the bag test applet from Appendix I to test your new string bag.

**15** Implement the `Statistician` class from Project 2 on page 95, but include a new method that returns the median value of all the numbers. The median is a number that is greater than or equal to at least half of the numbers and is also less than or equal to at least half of the numbers.

Because of the new median calculation, the `Statistician` will need to keep track of all the numbers, perhaps using an array. The median calculation will be easiest if you keep these numbers in order from smallest to largest.

**16** Implement the previous project with the following modification: All of the input numbers to the `Statistician` are required to be integers in the range from 0 to 100. This modification means that it's easier to keep track of all the input numbers using a single array (called `frequency`) with indexes from 0 to 100. At all times, the value of `frequency[i]` will be the number of times the number `i` has been given to the `Statistician`.

You will need to give it some thought to figure out how to use the frequency information to compute statistics such as the median and the mean.

**17** Design, specify, and implement a collection class that is similar to an array of double numbers except that it automatically grows when needed and negative indexes are also permitted.

The class should include a method to put a double number into the "array" at a specified index. For example, suppose that `v` is an object of this class. Then `v.put(3.8, 7)` would put the number 3.8 at index 7 of the "array." Since negative indexes are also allowed, the statement `v.put(9.1, -3)` would put the number 9.1 at the index -3.

The class also includes a method to retrieve the number from a specified index. For example, `v.get(7)` would return the value of the number that is currently at index 7 of the "array." If a programmer tries to get a value that has not yet been put into the array, then the `get` method will return the Java constant `Double.NaN` (which is the way Java represents a double value that is not a real number).

The class should also have two methods that return the value of the largest and the smallest indexes that have ever been set with the `put` method. If no numbers have yet been put in the "array," then the "largest" index should be `Integer.MIN_VALUE` and the "smallest" index should be `Integer.MAX_VALUE`.

**18** In this project, you will implement a new class called a **bag with receipts**. This new class is similar to an ordinary bag, but the data consists of strings, and the way that the strings are added and removed is different. Each time a string is added to a bag with receipts, the `insert` method returns a unique integer called the `receipt`. Later, when you want to remove a string, you must



## 174 Chapter 3 / Collection Classes

**23** Write a program that uses a HashSet of strings to keep track of a list of chores that you have to accomplish today. The user of the program can request several services: (1) add an item to the list of chores; (2) ask how many chores are in the list; (3) use the iterator to print the list of chores to the screen; (4) delete an item from the list; and (5) exit the program.

If you know how to read and write strings from a file, then have the program obtain its initial list of chores from a file. When the program ends, it should write all unfinished chores back to this file.

**24** Write a program that contains two arrays called `actors` and `roles`, each of size  $n$ . For each  $i$ , `actors[i]` is the name of an actor and `roles[i]` is a HashSet of strings that contains the names of the movies that the actor has appeared in. The program reads the initial information for these arrays from files in a format that you design.

Once the program is running, the user can type in the name of an actor and receive a list of all the movies for that actor. Or the user may type the name of a movie and receive a list of all the actors in that movie.

**25** An array can be used to store large integers one digit at a time. For example, the integer 1234 could be stored in the array `a` by setting `a[0]` to 1, `a[1]` to 2, `a[2]` to 3, and `a[3]` to 4. However, for this project, you might find it easier to store the digits backward, that is, place 4 in `a[0]`, place 3 in `a[1]`, place 2 in `a[2]`, and place 1 in `a[3]`.

Design, implement, and test a class in which each object is a large integer with each digit stored in a separate element of an array. You'll also need a private instance variable to keep track of the sign of the integer (perhaps a boolean variable). The number of digits may grow as the program runs, so the array may have to grow beyond its original size.

Discuss and implement other appropriate operators for this class (which is similar to Java's own `BigInteger` class).

**26** Suppose that you want to implement a bag class to hold non-negative integers, and you know that the biggest number in the bag will never be more than a few thousand. One approach for implementing this bag is to have a private instance variable that is an array of integers called `count` with indexes from 0 to  $M$  (where  $M$  is the maximum number in the bag). If the bag contains six copies of a number  $n$ , then the object has `count[n]` equal to 6 to represent this fact.

For this project, reimplement the bag class from Figure 3.1 using this idea. You will have an entirely new set of private instance variables; for the public methods, you may delete the capacity methods, but please add a new method that the programmer can use to specify the maximum number that he or she anticipates putting into the bag. Also note that the `add` method must check to see whether the current maximum index of the array is at least as big as the new number. If not, then the array size must be increased.



# CHAPTER 4

# Linked Lists

---

## LEARNING OBJECTIVES

When you complete Chapter 4, you will be able to ...

- design and implement methods to manipulate nodes in a linked list, including inserting new nodes, removing nodes, searching for nodes, and processing (such as copying) that involves all the nodes of a list.
- design and implement collection classes that use linked lists to store a collection of elements, generally using a node class to create and manipulate the linked lists.
- analyze problems that can be solved with linked lists and, when appropriate, propose alternatives to simple linked lists, such as doubly linked lists and lists with dummy nodes.

---

## CHAPTER CONTENTS

- 4.1 Fundamentals of Linked Lists
- 4.2 Methods for Manipulating Nodes
- 4.3 Manipulating an Entire Linked List
- 4.4 The Bag ADT with a Linked List
- 4.5 Programming Project: The Sequence ADT with a Linked List
- 4.6 Beyond Simple Linked Lists
  - Chapter Summary
  - Solutions to Self-Test Exercises
  - Programming Projects

## CHAPTER

## 4

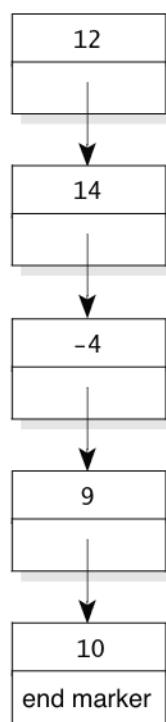
## Linked Lists

*The simplest way to interrelate or link a set of elements is to line them up in a single list... For, in this case, only a single link is needed for each element to refer to its successor.*

NIKLAUS WIRTH

*Algorithms + Data Structures = Programs*

linked lists are  
used to  
implement a list  
of elements  
arranged in  
some kind of  
order



**FIGURE 4.1**  
A Linked List  
Made of Nodes  
Connected with  
Links

We begin this chapter with a concrete discussion of a new data structure, the *linked list*, which is used to implement a list of elements arranged in some kind of order. The linked list structure uses memory that shrinks and grows as needed but in a different manner than arrays. The discussion of linked lists includes the specification and implementation of a node class, which incorporates the fundamental notion of a single element of a linked list.

Once you understand the fundamentals, linked lists can be used as part of an ADT, similar to the way that arrays have been used in previous ADTs. For example, linked lists can be used to reimplement the bag and sequence ADTs.

By the end of the chapter, you will understand linked lists well enough to use them in various programming projects (such as the revised bag and sequence ADTs) and in the ADTs of future chapters. You will also know the advantages and drawbacks of using linked lists versus arrays for these ADTs.

## 4.1 FUNDAMENTALS OF LINKED LISTS

A **linked list** is a sequence of elements arranged one after another, with each element connected to the next element by a “link.” A common programming practice is to place each element together with the link to the next element, resulting in a component called a **node**. A node is represented pictorially as a box, with the element written inside the box and the link drawn as an arrow pointing out of the box. Several typical nodes are shown in Figure 4.1. For example, the topmost node has the number 12 as its element. Most of the nodes in the figure also have an arrow pointing out of the node. These arrows, or **links**, are used to connect one node to another.

The links are represented as arrows because they do more than simply connect two nodes. The links also place the nodes in a particular order. In Figure 4.1, the five nodes form a chain from top to bottom. The first node is linked to the second node; the second node is linked to the third node; and so on, until we reach the last node. We must do something special when we reach the last node since the last node is not linked to another node. In this special case, we will replace the link in this node with a note saying “end marker.”

## Declaring a Class for Nodes

Each node contains two pieces of information: an element (which is a number for these example nodes) and an arrow. But just *what* are those arrows? Each arrow points to another node, or you could say that each arrow *refers* to another node. With this in mind, we can implement a Java class for a node using two instance variables: an instance variable to hold the element and a second instance variable that is a reference to another node. In Java, the two instance variables can be declared at the start of the class:

```
public class IntNode
{
    private int data;      // The element stored in this node
    private IntNode link; // Refers to the next node in the list
    ...
}
```

We'll provide the methods later (in Sections 4.2 and 4.3). For now we want to focus on the two instance variables: `data` and `link`. The `data` is simply an integer element, though we could have had some other kinds of elements, perhaps double numbers, or characters, or whatever. The `link` is a reference to another node. For example, the `link` variable in the first node is a reference to the second node. Our drawings will represent each link reference as an arrow leading from one node to another. In fact, we have previously used arrows to represent references to objects in Chapters 2 and 3, so these links are nothing new.

## Head Nodes, Tail Nodes

When a program builds and manipulates a linked list, the list is usually accessed through references to one or more important nodes. The most common access is through the list's first node, which is called the **head** of the list. Sometimes we maintain a reference to the last node in a linked list. The last node is the **tail** of the list. We could also maintain references to other nodes in a linked list.

Each reference to a node used in a program must be declared as a node variable. For example, if we are maintaining a linked list with references to the head and tail, then we would declare two node variables:

```
IntNode head;
IntNode tail;
```

The program can now proceed to create a linked list, always ensuring that `head` refers to the first node and `tail` refers to the last node, as shown in Figure 4.2.

### Building and Manipulating Linked Lists

Whenever a program builds and manipulates a linked list, the nodes are accessed through one or more references to nodes. Typically, a program includes a reference to the first node (the **head**) and a reference to the last node (the **tail**).

**FIGURE 4.2** Node Declarations in a Program with a Linked List**Declaration from the IntNode Class**

```
public class IntNode
{
    private int data;
    private IntNode link;
    ...
}
```

**Declaring Two Nodes in a Program**

```
IntNode head;
IntNode tail;
```

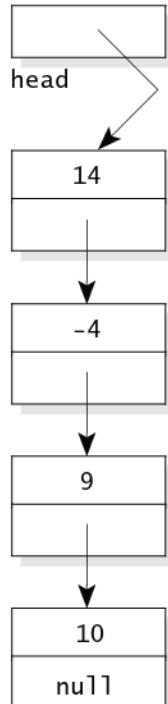
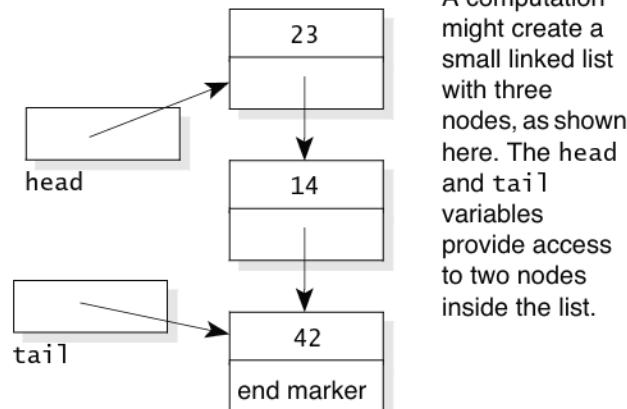
**The Null Reference**

Figure 4.3 illustrates a linked list with one new feature. Look at the link part of the final node. Instead of a reference, we have written the word `null`. The word `null` indicates the **null reference**, which is a special Java constant. You can use the null reference for any reference variable that has nothing to refer to. There are several common situations in which the null reference is used.

**The Null Reference and Linked Lists**

In Java, when a reference variable is first declared and there is not yet an object for it to refer to, it can be given an initial value of the null reference. Examples of this initial value are shown in Chapter 2 on page 54.

The null reference is used for the link part of the final node of a linked list.

When a linked list does not yet have any nodes, the null reference is used for the head and tail reference variables. Such a list is called the **empty list**.

In a program, the null reference is written as the keyword `null`.

**FIGURE 4.3**

Linked List with the Null Reference at the Final Link

**PITFALL****NULLPOINTEREXCEPTIONS WITH LINKED LISTS**

When a reference variable is null, it is a programming error to activate one of its methods or to try to access one of its instance variables. For example, a program may maintain a reference to the head node of a linked list, as shown here:

```
IntNode head;
```

Initially, the list is empty and head is the null reference. At this point, it is a programming error to activate one of head's methods. The error would occur as a `NullPointerException`.

The general rules: Never activate a method of the null reference. Never try to access an instance variable of the null reference. In both cases, the result would be a `NullPointerException`.

**Self-Test Exercises for Section 4.1**

1. Write the start of the class declaration for a node in a linked list. The data in each node is a double number.
2. Write another node declaration, but this time the data in each node should include both a double number and an integer. (Yes, a node can have many instance variables for data, but each instance variable needs to have a distinct name.)
3. Suppose a program builds and manipulates a linked list. What two special nodes would the program typically keep track of?
4. Describe two uses for the null reference in the realm of linked lists.
5. How many nodes are in an empty list? What are the values of the head and tail references for an empty list?
6. What happens if you try to activate a method of the null reference?

**4.2 METHODS FOR MANIPULATING NODES**

We're ready to write methods for the `IntNode` class, which begins like this:

```
public class IntNode
{
    private int data;      // The element stored in this node
    private IntNode link; // Refers to the next node in the list
    ...
}
```

There will be methods for creating, accessing, and modifying nodes, plus methods and other techniques for adding or removing nodes from a linked list. We begin with a constructor that's responsible for initializing the two instance variables of a new node.

### Constructor for the Node Class

The node's constructor has two arguments, which are the initial values for the node's data and link variables, as specified here:

#### ◆ Constructor for the IntNode

```
public IntNode(int initialValue, IntNode initialLink)
```

Initialize a node with specified initial data and a link to the next node. Note that the `initialLink` may be the null reference, which indicates that the new node has nothing after it.

#### Parameters:

`initialValue` – the initial data of this new node

`initialLink` – a reference to the node after this new node (the reference may be null to indicate that there is no node after this new node)

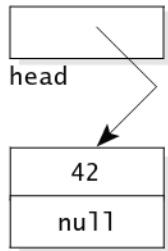
#### Postcondition:

This new node contains the specified data and a link to the next node.

The constructor's implementation copies its two parameters to the instance variables `data` and `link`:

```
public IntNode(int initialValue, IntNode initialLink)
{
    data = initialValue;
    link = initialLink;
}
```

As an example, the constructor can be used by a program to create the first node of the linked list shown in the margin.



```
IntNode head;
head = new IntNode(42, null);
```

After these two statements, `head` refers to the head node of a small linked list that contains just one node with the number 42. We'll look at the formation of longer linked lists after we see four other basic node methods.

### Getting and Setting the Data and Link of a Node

The node has an accessor method and a modification method for each of its two instance variables, as specified here:

#### ◆ `getData`

```
public int getData()
```

Accessor method to get the data from this node.

#### Returns:

the data from this node

`getData`,  
`getLink`,  
`setData`,  
`setLink`

**◆ getLink**

```
public IntNode getLink()
```

Accessor method to get a reference to the next node after this node.

**Returns:**

a reference to the node after this node (or the null reference if there is nothing after this node)

**◆ setData**

```
public void setData(int newdata)
```

Modification method to set the data in this node.

**Parameter:**

newData – the new data to place in this node

**Postcondition:**

The data of this node has been set to newData.

**◆ setLink**

```
public void setLink(IntNode newLink)
```

Modification method to set the reference to the next node after this node.

**Parameter:**

newLink – a reference to the node that should appear after this node in the linked list (or the null reference if there should be no node after this node)

**Postcondition:**

The link to the node after this node has been set to newLink. Any other node (that used to be in this link) is no longer connected to this node.

The implementations of the four methods are short. For example:

```
public void setLink(IntNode newLink)
{
    link = newLink;
}
```

### Public Versus Private Instance Variables

In addition to `setLink`, there are the three other short methods (documented previously) that we'll leave for you to implement. You may wonder why you should bother having these short methods at all. Wouldn't it be simpler and more efficient to just make `data` and `link` public and do away with the short methods altogether? Yes, public instance variables probably are simpler, and the direct access of an instance variable is more efficient than calling a method. On the other hand, debugging can be easier with access and modification methods in place because we can set breakpoints to see whenever an instance variable is accessed or modified. Also, private instance variables provide good information hiding so that later changes to the class won't affect programs that use the class.

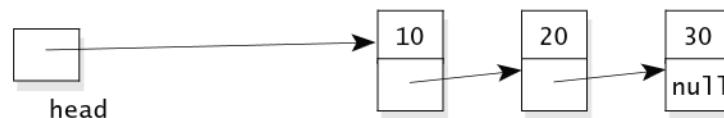
The public-versus-private question should be addressed for many of your classes, with the answer based on the intended use and required efficiency

## 182 Chapter 4 / Linked Lists

together with software engineering principles such as information hiding. For the classes in this text, we'll lean toward information hiding and avoid public instance variables.

### Adding a New Node at the Head of a Linked List

New nodes can be added at the head of a linked list. To accomplish this, the program needs a reference to the head node of a list, as shown here:

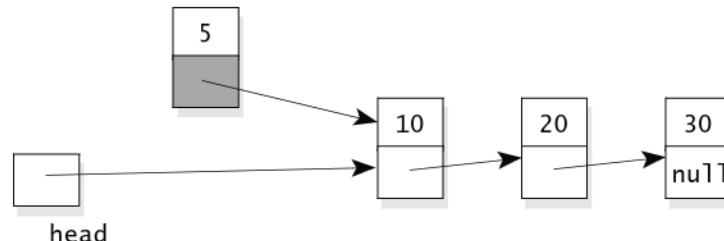


In this example, suppose we want to add a new node to the front of this list, with 5 as the data. Using the node constructor, we can write:

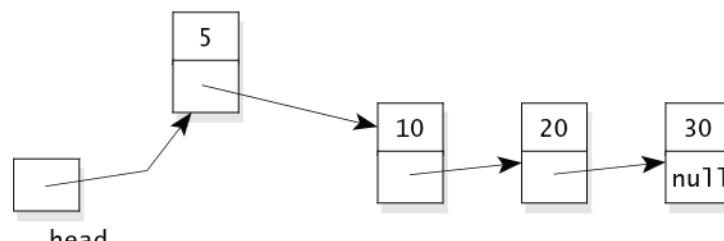
```
head = new IntNode(5, head);
```

*how to add a new node at the head of a linked list*

Let's step through the execution of this statement to see how the new node is added at the front of the list. When the constructor is executed, a new node is created with 5 as the data and with the link referring to the same node that `head` refers to. Here's what the picture looks like, with the link of the new node shaded:

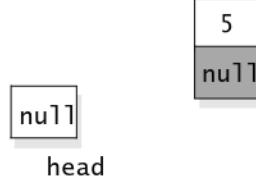


The constructor returns a reference to the newly created node, and in the statement, we wrote `head = new IntNode(5, head)`. You can read this statement as saying "make `head` refer to the newly created node." Therefore, we end up with this situation:



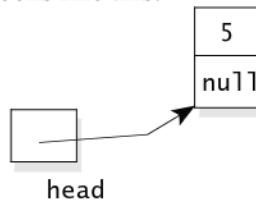
By the way, the technique works correctly even when we start with an empty list (in which the `head` reference is `null`). In this case, the statement

`head = new IntNode(5, head)` creates the first node of the list. To see this, suppose we start with a null head and execute the statement. The constructor creates a new node with 5 as the data and with `head` as the link. Since the `head` reference is null, the new node looks like this (with the link of the new node shaded):



head

After the constructor returns, `head` is assigned to refer to the new node, so the final situation looks like this:



As you can see, the statement `head = new IntNode(5, head)` has correctly added the first node to a list. If we are maintaining a reference to the tail node, then we would also set the tail to refer to this one node.

#### Adding a New Node at the Head of a Linked List

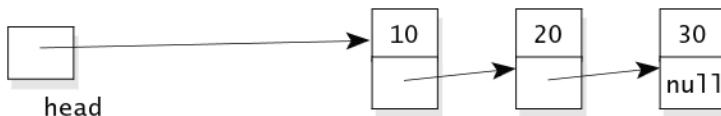
Suppose `head` is the head reference of a linked list. Then this statement adds a new node at the front of the list with the specified new data:

```
head = new IntNode(newData, head);
```

This statement works correctly even when we start with an empty list (in which case the `head` reference is null).

#### Removing a Node from the Head of a Linked List

Nodes can be removed from the head of the linked list. To accomplish this, we need a reference to the head node of a list:



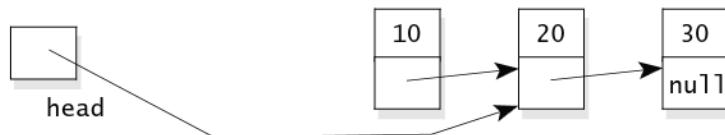
To remove the first node, we move the head so that it refers to the next node. This is accomplished with one statement:

## 184 Chapter 4 / Linked Lists

```
head = head.getLink();
```

how to remove a node from the head of a linked list

The right side of the assignment, `head.getLink()`, is a reference to the second node of the list. So, after the assignment, `head` refers to the second node:



automatic garbage collection has some inefficiency, but it's less prone to programming errors

This picture is peculiar. It looks like we still have a linked list with three nodes containing 10, 20, and 30. But if we start at the head, there are only the two nodes with 20 and 30. The node with 10 can no longer be accessed starting at the head, so it is not really part of the linked list anymore. In fact, if a situation arises in which a node can no longer be accessed from anywhere in a program, then the Java runtime system recognizes that the node has strayed, and the memory used by that node will be reused for other things. This technique of rounding up stray memory is called **garbage collection**, and it happens automatically for Java programs. In other programming languages, the programmer is responsible for identifying memory that is no longer used and explicitly returning that memory to the runtime system.

What are the trade-offs between automatic garbage collection and programmer-controlled memory handling? Automatic garbage collection is slower when a program is executing, but the automatic approach is less prone to errors, and it frees the programmer to concentrate on more important issues.

Anyway, we'll remove a node from the front of a list with the statement `head = head.getLink()`. This statement also works when the list has only one node, and we want to remove this one node. For example, consider this list:



In this situation, we can execute `head = head.getLink()`. The `getLink()` method returns the link of the head node—in other words, it returns `null`. So, the null reference is assigned to the head, ending up with this situation:



Now the head is null, which indicates that the list is empty. If we are maintaining a reference to the tail, then we would also have to set the tail reference to null. The automatic garbage collection will then take care of reusing the memory occupied by the one node.

### Removing a Node from the Head of a Linked List

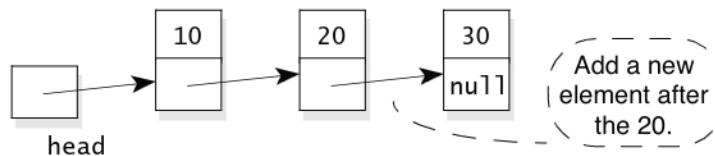
Suppose `head` is the head reference of a non-empty linked list (so that `head` is not `null`). Then this statement removes a node from the front of the list:

```
head = head.getLink( );
```

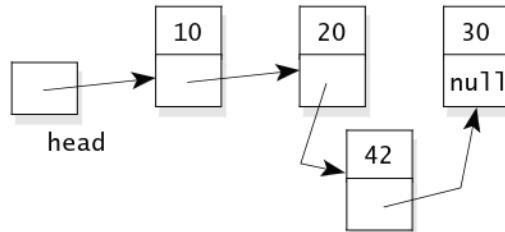
This statement works correctly even when the list has just one node (in which case the `head` reference becomes `null`).

### Adding a New Node That Is Not at the Head

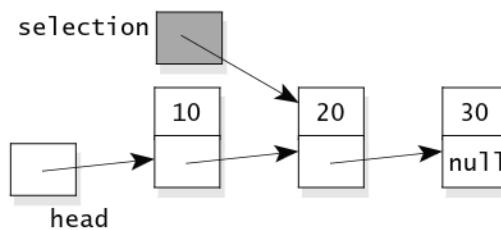
New nodes are not always placed at the head of a linked list. They may be added in the middle or at the tail of a list. For example, suppose you want to add the number 42 after the 20 in this list:



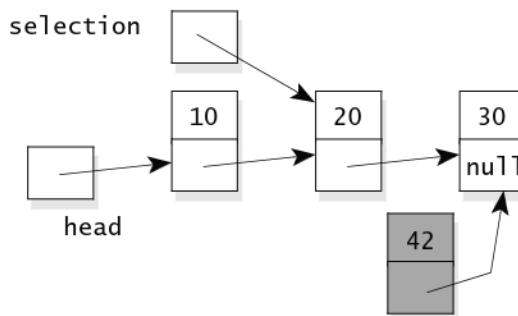
After the addition, the new, longer list has these four nodes:



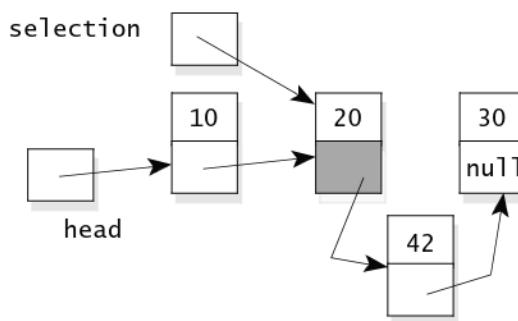
Whenever a new node is not at the head, the process requires a reference to the node that is just *before* the intended location of the new node. In our example, we would require a reference to the node that contains 20 since we want to place the new node after this node. This special node is called the “selected node,” and the new node will go just after the selected node. We’ll use the name `selection` for a reference to the selected node. So, to add an element after the 20, we would first have to set up `selection` this way:







The constructor returns a reference to the newly created node, and in the assignment statement we wrote `link = new IntNode(element, link)`. You can read this statement as saying “change the link part of the selected node so that it refers to the newly created node.” This change is made in the following drawing, which highlights the link part of the selected node:



After adding the new node with 42, you can step through the complete linked list, starting at the head node 10, then 20, then 42, and finally 30.

The approach we have used works correctly even when the selected node is the tail of a list. In this case, the new node is added after the tail. If we were maintaining a reference to the tail node, then we would have to update this reference to refer to the newly added tail.

#### Adding a New Node That Is Not at the Head

Suppose `selection` is a reference to a node of a linked list. Activating the following method adds a new node after the selected node (using `element` as the new data):

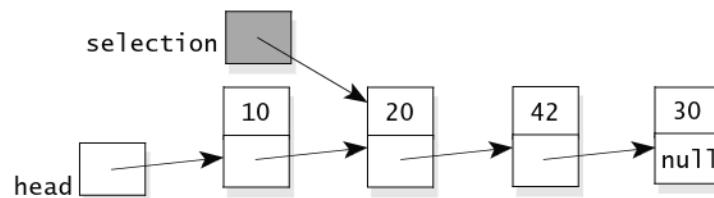
```
selection.addNodeAfter(element);
```

The implementation of `addNodeAfter` needs only one statement to accomplish its work:

```
link = new IntNode(element, link);
```

### Removing a Node That Is Not at the Head

It is also possible to remove a node that is not at the head of a linked list. The approach is similar to adding a node in the middle of a linked list. To remove a midlist node, we must set up a reference to the node that is just *before* the node we are removing. For example, to remove the 42 from the following list, we would need to set up `selection` as shown here:



As you can see, `selection` does not actually refer to the node we are deleting (the 42); instead, it refers to the node that is just before the condemned node. This is because the link of the *previous* node must be reassigned; hence, we need a reference to this previous node. The removal method's specification is shown here:

`removeNodeAfter`

◆ **removeNodeAfter**

`public void removeNodeAfter()`

Modification method to remove the node after this node.

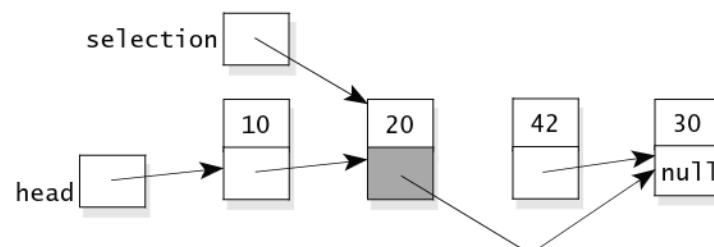
**Precondition:**

This node must not be the tail node of the list.

**Postcondition:**

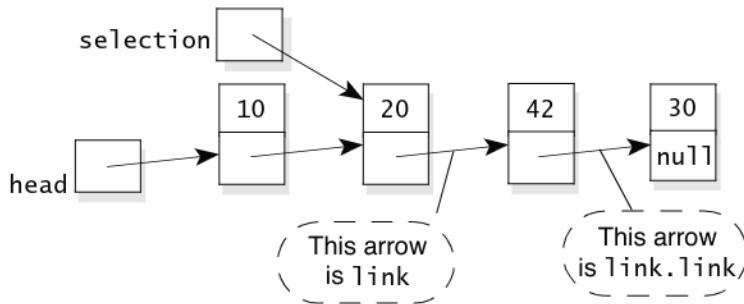
The node after this node has been removed from the linked list. If there were further nodes after that one, they are still present in the list.

For example, to remove the 42 from the list drawn above, we would activate `selection.removeNodeAfter()`. After the removal, the new list will look like this (with the changed link highlighted):



At this point, the node containing 42 is no longer part of the linked list. The list's first node contains 10; the next node has 20; and following the links we arrive at the third and last node, containing 30. Java's automatic garbage collection will reuse the removed node's memory.

The implementation of `removeNodeAfter` must alter the link of the node that activated the method. How is the alteration carried out? Let's go back to our starting position, but we'll put a bit more information in the picture:



To work through this example, you need some patterns that can be used within the method to refer to the various data and link parts of the nodes. Remember that we activated `selection.removeNodeAfter()`, so the node that activated the method has 20 for its data, and its link is indicated by the caption "This arrow is `link`." So we can certainly use these two names within the method:

- |                   |                                                                                                            |
|-------------------|------------------------------------------------------------------------------------------------------------|
| <code>data</code> | This is the data of the node that activated the method (20).                                               |
| <code>link</code> | This is the link of the node that activated the method. This link refers to the node that we are removing. |

Because the name `link` refers to a node, we can also use the names `link.data` and `link.link`:

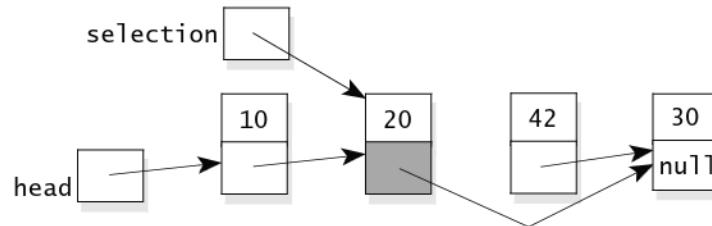
- |                        |                                                                                                                                                                                                              |
|------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>link.data</code> | This notation means "go to the node that <code>link</code> refers to and use the <code>data</code> instance variable." In our example, <code>link.data</code> is 42.                                         |
| <code>link.link</code> | This notation means "go to the node that <code>link</code> refers to and use the <code>link</code> instance variable." In our example, <code>link.link</code> is the reference to the node that contains 30. |

In the implementation of `removeNodeAfter`, we need to make `link` refer to the node that contains 30. So, using the notation just shown, we need to assign `link = link.link`. The complete implementation is at the top of the next page.

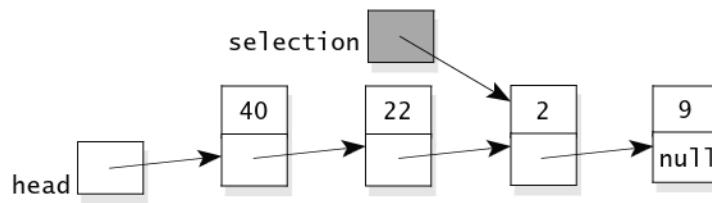
## 190 Chapter 4 / Linked Lists

```
public void removeNodeAfter( )
{
    link = link.link;
}
```

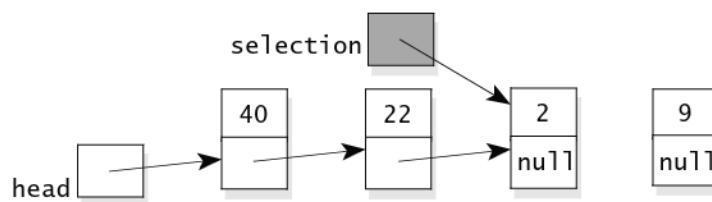
The notation `link.link` does look strange, but just read it from left to right so that it means “go to the node that `link` refers to and use the `link` instance variable that you find there.” In our example, the final situation after assigning `link = link.link` is just what we want:



The `removeNodeAfter` implementation works fine, even when we want to remove the tail node. Here’s an example in which we have set `selection` to refer to the node that’s just before the tail of a small list:



When we activate `selection.removeNodeAfter( )`, the link of the selected node will be assigned the value `null` (which is obtained from the link of the next node). The result is shown in this picture:



The tail node has been removed from the list. If the program maintains a reference to the tail node, then that reference must be updated to refer to the new tail.

In all cases, Java’s automatic garbage collection takes care of reusing the removed node’s memory.

**Removing a Node That Is Not at the Head**

Suppose `selection` is a reference to a node of a linked list. Activating the following method removes the node after the selected node:

```
selection.removeNodeAfter();
```

The implementation of `removeNodeAfter` needs only one statement to accomplish its work:

```
link = link.link;
```

**PITFALL****NULLPOINTEREXCEPTIONS WITH REMOVE NODEAFTER**

The `removeNodeAfter` method has a potential problem. What happens if the tail node activates `removeNodeAfter`? This is a programming error because `removeNodeAfter` would try to remove the node after the tail node, and there is no such node. The precondition of `removeNodeAfter` explicitly states that it must not be activated by the tail node. Still, what will happen in this case? For the tail node, `link` is the null reference, so trying to access the instance variable `link.link` will result in a `NullPointerException`.

When we write the complete specification of the node methods, we will include a note indicating the possibility of a `NullPointerException` in this method.

**Self-Test Exercises for Section 4.2**

7. Suppose `head` is a head reference for a linked list of integers. Write a few lines of code that will add a new node with the number 42 as the second element of the list. (If the list was originally empty, then 42 should be added as the first node instead of the second.)
8. Suppose `head` is a head reference for a linked list of integers. Write a few lines of code that will remove the second node of the list. (If the list originally had only one node, then remove that node instead; if it had no nodes, then leave the list empty.)
9. Examine the techniques for adding and removing a node at the head. Why are these techniques implemented as static methods rather than ordinary `IntNode` methods?
10. Write some code that could appear in a main program. The code should declare `head` and `tail` references for a linked list of integers and then create a list of nodes with the data being integers 1 through 100 (in that order). After each of the nodes is added, `head` and `tail` should still be valid references to the head and tail nodes of the current list.

### 4.3 MANIPULATING AN ENTIRE LINKED LIST

We can now write programs that use linked lists. Such a program declares some references to nodes, such as head and tail references. The nodes are manipulated with the methods and other techniques we have already seen. But all these methods and techniques deal with just one or two nodes at an isolated part of the linked list. Many programs also need techniques for carrying out computations on an entire list, such as computing the number of nodes in a list. This suggests that we should write a few more methods for the `IntNode` class—methods that carry out some computation on an entire linked list. For example, we can provide a method with this heading:

```
public static int listLength(IntNode head)
```

The `listLength` method computes the number of nodes in a linked list. The one parameter, `head`, is a reference to the head node of the list. For example, the last line of this code prints the length of a short list:

```
IntNode small; // Head reference for a small list
small = new IntNode(42, null);
small.addNodeAfter(17);
System.out.println(IntNode.listLength(small)); // Prints 2
```

By the way, the `listLength` return value is `int` so that the method can be used only if a list has fewer than `Integer.MAX_VALUE` nodes. Beyond this length, the `listLength` method will return a wrong answer because of arithmetic overflow. We'll make a note of the potential problem in the `listLength` specification.

Notice that `listLength` is a static method. It is not activated by any one node; instead, we activate `IntNode.listLength`. But why is it a *static* method? Wouldn't it be easier to write an ordinary method that is activated by the head node of the list? Yes, an ordinary method might be easier, but a static method is better because a static method can be used even for an empty list. For example, these two statements create an empty list and print the length of that list:

```
IntNode empty = null; // empty is null, representing an empty list
System.out.println(IntNode.listLength(empty)); // Prints 0
```

An ordinary method could not be used to compute the length of the empty list, because the head reference is null.

#### Manipulating an Entire Linked List

To carry out computations on an entire linked list, we will write static methods in the `IntNode` class. Each such method has one or more parameters that are references to nodes in the list. Most of the methods will work correctly even when the references are null (indicating an empty list).

## Computing the Length of a Linked List

Here is the complete specification of the `ListLength` method we've been discussing:

◆ **listLength**

```
public static int listLength(IntNode head)
```

Compute the number of nodes in a linked list.

*listLength*

**Parameter:**

`head` – the head reference for a linked list (which may be an empty list with a null head)

**Returns:**

the number of nodes in the list with the given head

**Note:**

A wrong answer occurs for lists longer than `Int.MAX_VALUE` because of arithmetic overflow.

The precondition indicates that the parameter, `head`, is the head reference for a linked list. If the list is not empty, then `head` refers to the first node of the list. If the list is empty, then `head` is the null reference (and the method returns zero since there are no nodes).

Our implementation uses a reference variable to step through the list, counting the nodes one at a time. Here are the three steps of the pseudocode, using a reference variable named `cursor` to step through the nodes of the list one at a time. (We often use the name `cursor` for such a variable since “cursor” means “something that runs through a structure.”)

1. Initialize a variable named `answer` to zero. (This variable will keep track of how many nodes we have seen so far.)
2. Make `cursor` refer to each node of the list, starting at the head node.  
Each time `cursor` moves, add 1 to `answer`.
3. `return answer`.

Both `cursor` and `answer` are local variables in the method.

The first step initializes `answer` to zero because we have not yet seen any nodes. The implementation of Step 2 is a for-loop, following a pattern that you should use whenever *all of the nodes of a linked list must be traversed*. The general pattern looks like this:

```
for (cursor = head; cursor != null; cursor = cursor.link)
{
    ...
}
```

*Inside the body of the loop, you can  
carry out whatever computation is  
needed for a node in the list.*

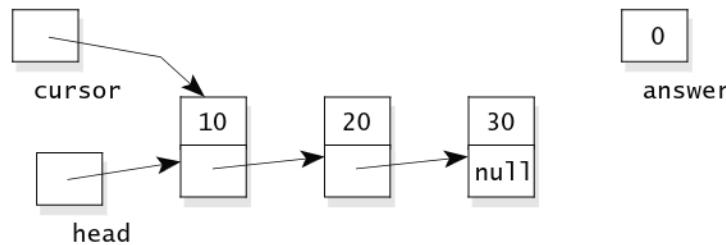
*how to traverse  
all the nodes of  
a linked list*

**194** Chapter 4 / Linked Lists

For the `listLength` method, the “computation” inside the loop is simple because we are just counting the nodes. Therefore, in our body, we will just add 1 to answer:

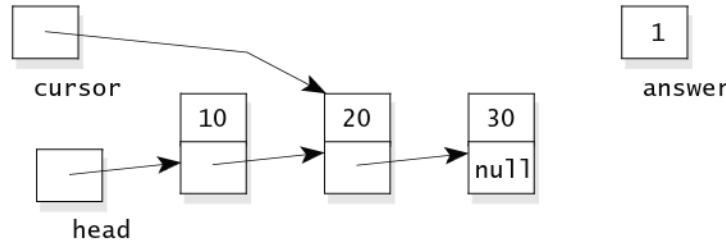
```
for (cursor = head; cursor != null; cursor = cursor.link)
    answer++;
```

Let’s examine the loop in an example. Suppose the linked list has three nodes containing the numbers 10, 20, and 30. After the loop initializes (with `cursor = head`), we have this situation:

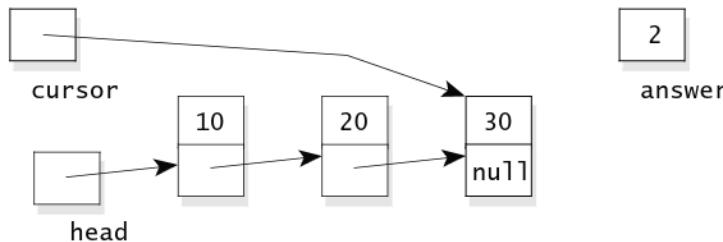


Notice that `cursor` refers to the same node that `head` refers to.

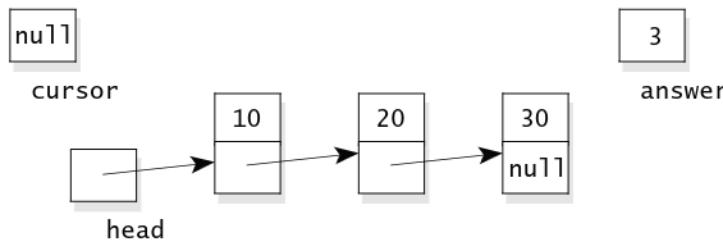
Since `cursor` is not `null`, we enter the body of the loop. Each iteration increments `answer` and then executes `cursor = cursor.link`. The effect of `cursor = cursor.link` is to copy the `link` part of the first node into `cursor` itself so that `cursor` ends up referring to the second node. In general, the statement `cursor = cursor.link` moves `cursor` to the next node. So, at the completion of the loop’s first iteration, the situation is this:



The loop continues. After the second iteration, `answer` is 2 and `cursor` refers to the third node of the list, as shown next.



Each time we complete an iteration of the loop, `cursor` refers to some location in the list, and `answer` is the number of nodes *before* this location. In our example, we are about to enter the loop's body for the third and last time. During the last iteration, `answer` is incremented to 3, and `cursor` becomes `null`:



The variable `cursor` has become `null` because the loop control statement `cursor = cursor.link` copied the `link` part of the third node into `cursor`. Since this `link` part is `null`, the value in `cursor` is now `null`.

At this point, the loop's control test `cursor != null` is false. The loop ends, and the method returns the answer 3. The complete implementation of the `listLength` method is shown in Figure 4.4.

---

**FIGURE 4.4** A Static Method to Compute the Length of a Linked List

### Implementation

```
public static int listLength(IntNode head)
{
    IntNode cursor;
    int answer;

    answer = 0;
    for (cursor = head; cursor != null; cursor = cursor.link)
        answer++;
    return answer;
}
```

*Step 2 of the pseudocode*

---

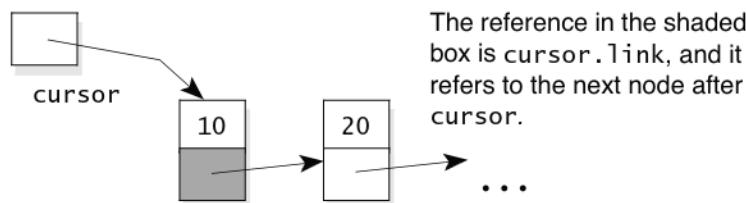
## PROGRAMMING TIP

### HOW TO TRAVERSE A LINKED LIST

You should learn the important pattern for traversing a linked list, as used in the `listLength` method (see Figure 4.4). The same pattern can be used whenever you need to step through the nodes of a linked list one at a time.

The first part of the pattern concerns moving from one node to another. Whenever we have a variable that refers to some node and we want the variable to refer to the next node, we must use the `link` part of the node. Here is the reasoning that we follow:

1. Suppose `cursor` refers to some node.
2. Then `cursor.link` refers to the next node (if there is one), as shown here:



3. To move `cursor` to the next node, we use one of these assignment statements:

```
cursor = cursor.link;  
or  
cursor = cursor.getLink();
```

Use the first version, `cursor = cursor.link`, if you have access to the `link` instance variable (inside one of the `IntNode` methods). Otherwise, use the second version, `cursor = cursor.getLink()`. In both cases, if there is no next node, then `cursor.link` will be `null`, and therefore our assignment statement will set `cursor` to `null`.

The key is to know that the assignment statement `cursor = cursor.link` moves `cursor` so that it refers to the next node. If there is no next node, then the assignment statement sets `cursor` to `null`.

The second part of the pattern shows how to traverse all of the nodes of a linked list, starting at the head node. The pattern of the loop looks like this:

```
for (cursor = head; cursor != null; cursor = cursor.link)  
{  
    ...  
        Inside the body of the loop, you can  
        carry out whatever computation is  
        needed for a node in the list.  
}
```

You'll find yourself using this pattern continually in methods that manipulate linked lists.

**PITFALL** 
**FORGETTING TO TEST THE EMPTY LIST**

Methods that manipulate linked lists should always be tested to ensure that they have the right behavior for the empty list. When `head` is `null` (indicating the empty list), our `listLength` method should return 0. Testing this case shows that `listLength` does correctly return 0 for the empty list.

**Searching for an Element in a Linked List**

In Java, a method can return a reference to a node. Hence, when the job of a subtask is to find a single node, it makes sense to implement the subtask as a method that returns a reference to that node. Our next method follows this pattern, returning a reference to a node that contains a specified element. The specification is given here:

**◆ listSearch**

```
public static IntNode listSearch(IntNode head, int target)
Search for a particular piece of data in a linked list.
```

**Parameters:**

`head` – the head reference for a linked list (which may be an empty list with a null head)  
`target` – a piece of data to search for

**Returns:**

The return value is a reference to the first node that contains the specified target. If there is no such node, the null reference is returned.

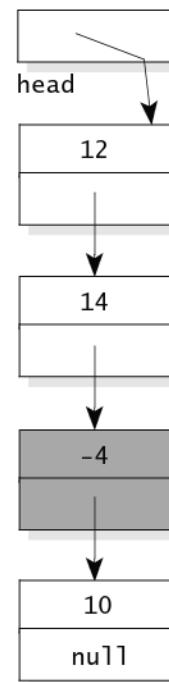
As indicated by the return type of `IntNode`, the method returns a reference to a node in a linked list. The node is specified by a parameter named `target`, which is the integer that appears in the sought-after node. For example, the activation `IntNode.listSearch(head, -4)` in Figure 4.5 will return a reference to the shaded node.

Sometimes, the specified target does not appear in the list. In this case, the method returns the null reference.

The implementation of `listSearch` is shown in Figure 4.6. Most of the work is carried out with the usual traversal pattern, using a local variable called `cursor` to step through the nodes one at a time:

```
for (cursor = head; cursor != null; cursor = cursor.link)
{
    if (target == the data in the node that cursor refers to)
        return cursor;
}
```

As the loop executes, `cursor` refers to the nodes of the list, one after another. The test inside the loop determines whether we have found the sought-after node, and if so, a reference to the node is immediately returned with the return

**listSearch**

**FIGURE 4.5**  
 Example for  
`listSearch`

**FIGURE 4.6** A Static Method to Search for a Target in a Linked List

### Implementation

```
public static IntNode listSearch(IntNode head, int target)
{
    IntNode cursor;

    for (cursor = head; cursor != null; cursor = cursor.link)
        if (target == cursor.data)
            return cursor;

    return null;
}
```

statement `return cursor`. When a return statement occurs like this inside a loop, the method returns without ado, and the loop is not run to completion.

On the other hand, should the loop actually complete by eventually setting `cursor` to `null`, then the sought-after node is not in the list. According to the method's postcondition, the method returns `null` when the node is not in the list. This is accomplished with one more return statement—`return null`—at the end of the method's implementation.

### Finding a Node by Its Position in a Linked List

Here's another method that returns a reference to a node in a linked list:

*listPosition*

◆ **listPosition**

public static IntNode listPosition(IntNode head, int position)  
Find a node at a specified position in a linked list.

**Parameters:**

head – the head reference for a linked list (which may be an empty list with a null head)  
position – a node number

**Precondition:**

position > 0

**Returns:**

The return value is a reference to the node at the specified position in the list. (The head node is position 1, the next node is position 2, and so on.) If there is no such position (because the list is too short), then the null reference is returned.

**Throws: IllegalArgumentException**

Indicates that position is not positive.



### Copying a Linked List

Our next static method makes a copy of a linked list, returning a head reference for the newly created copy. Here is the specification:

*listCopy*

◆ **listCopy**

public static IntNode listCopy(IntNode source)

Copy a list.

**Parameter:**

source – the head reference for a linked list that will be copied (which may be an empty list where source is null)

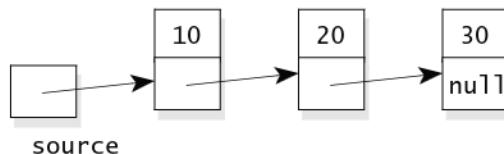
**Returns:**

The method has made a copy of the linked list starting at source. The return value is the head reference for the copy.

**Throws:** OutOfMemoryError

Indicates that there is insufficient memory for the new list.

For example, suppose that source refers to the following list:



The `listCopy` method creates a completely separate copy of the three-node list. The copy of the list has its own three nodes, which also contain the numbers 10, 20, and 30. The return value is a head reference for the new list, and the original list remains intact.

The pseudocode begins by handling one special case—the case in which the original list is empty (so that `source` is null). In this case the method simply returns `null`, indicating that its answer is the empty list. So, the first step of the pseudocode is:

1. if (`source == null`), then return `null`.

After dealing with the special case, the method uses two local variables called `copyHead` and `copyTail`, which will be maintained as the head and tail references for the new list. The pseudocode for creating this new list is given in Step 2 through Step 4 on the top of the next page.

2. Create a new node for the head node of the new list we are creating. Make both `copyHead` and `copyTail` refer to this new node, which contains the same data as the head node of the source list.
3. Make `source` refer to the second node of the original list, then the third node, then the fourth node, and so on, until we have traversed all of the original list. At each node that `source` refers to, add one new node to the tail of the new list and move `copyTail` forward to the newly added node, as follows:

```
copyTail.addNodeAfter(source.data);  
copyTail = copyTail.link;
```

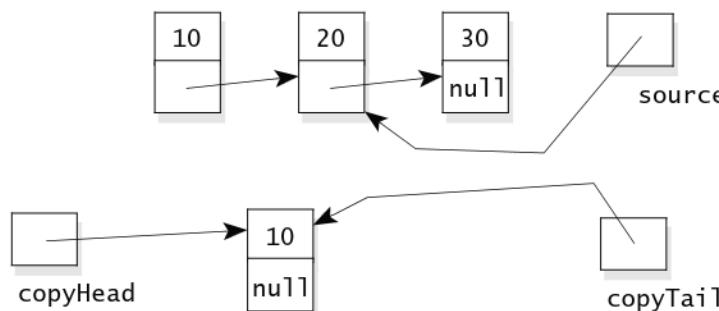
4. After Step 3 completes, return `copyHead` (a reference to the head node of the list we created).

Step 3 of the pseudocode is completely implemented by this loop:

```
while (source.link != null)  
{ // There are more nodes, so copy the next one.  
    source = source.link;  
    copyTail.addNodeAfter(source.data);  
    copyTail = copyTail.link;  
}
```

The while-loop starts by checking `source.link != null` to determine whether there is another node to copy. If there is another node, then we enter the body of the loop and move `source` forward with the assignment statement `source = source.link`. The second and third statements in the loop add a node at the tail end of the newly created list and move `copyTail` forward.

As an example, consider again the three-node list with data 10, 20, and 30. The first two steps of the pseudocode are carried out, and then we enter the body of the while-loop. We execute the first statement of the loop: `source = source.link`. At this point, the variables look like this:

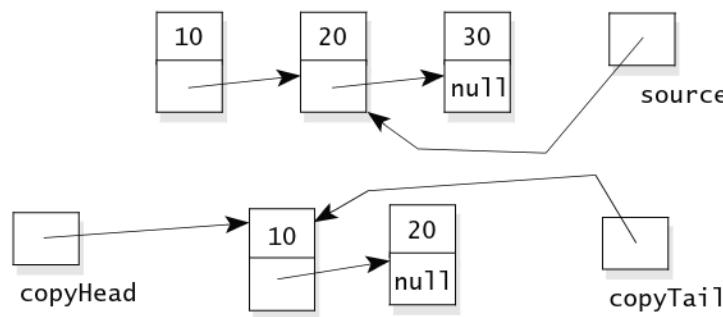


**202** Chapter 4 / Linked Lists

Notice that we have already copied the first node of the linked list. During the first iteration of the while-loop, we will copy the second node of the linked list—the node that is now referred to by `source`. The first part of copying the node works by activating one of our other methods, `addNodeAfter`:

```
copyTail.addNodeAfter(source.data);
```

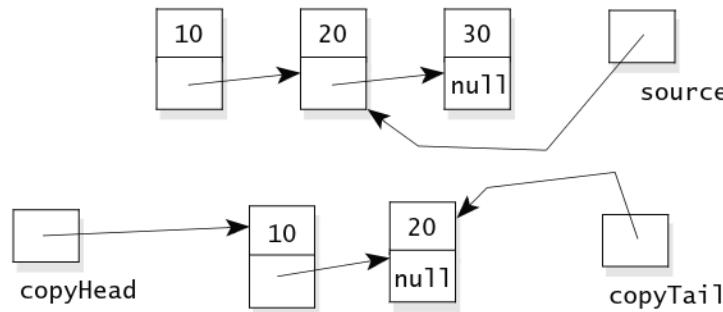
This activation adds a new node to the end of the list we are creating (i.e., *after* the node referred to by `copyTail`), and the data in the new node is the number 20 (i.e., the data from `source.data`). Immediately after adding the new node, the variables look like this:



The last statement in the while-loop body moves `copyTail` forward to the new tail of the new list:

```
copyTail = copyTail.link;
```

This is the usual way in which we make a node reference “move to the next node,” as we have seen in other methods, such as `listSearch`. After moving `copyTail`, the variables look like this:



In this example, the body of the while-loop will execute one more time to copy the third node to the new list. Then the loop will end, and the method will return the new head reference, `copyHead`.

---

**FIGURE 4.9** A Static Method to Copy a Linked ListImplementation

```
public static IntNode listCopy(IntNode source)
{
    IntNode copyHead;
    IntNode copyTail;

    // Handle the special case of an empty list.
    if (source == null)
        return null;

    // Make the first node for the newly created list.
    copyHead = new IntNode(source.data, null);
    copyTail = copyHead;

    // Make the rest of the nodes for the newly created list.
    while (source.link != null)
    {
        source = source.link;
        copyTail.addNodeAfter(source.data);
        copyTail = copyTail.link;
    }

    // Return the head reference for the new list.
    return copyHead;
}
```

---

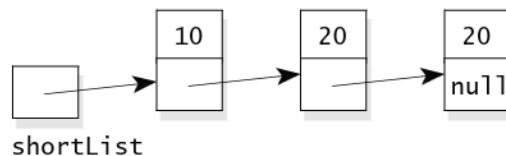
The complete implementation of `listCopy` is shown in Figure 4.9. Here's an example of how the `listCopy` method might be used in a program:

```
IntNode shortList;
IntNode copy;

shortList = new IntNode(10, null);
shortList.addNodeAfter(20);
shortList.addNodeAfter(20);
```

**204 Chapter 4 / Linked Lists**

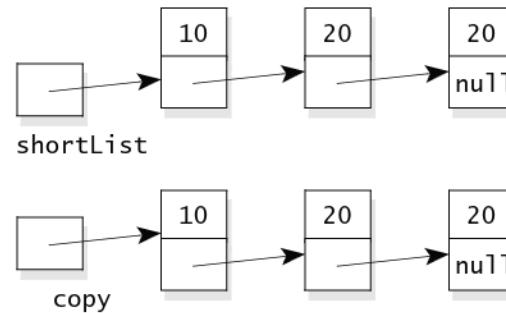
At this point, `shortList` is the head of a small list shown here:



We could now use `listCopy` to make a second copy of this list:

```
copy = IntNode.listCopy(shortList);
```

Now we have two separate lists:



*why is listCopy a static method?*

Keep in mind that `listCopy` is a static method, so we must write the expression `IntNode.listCopy(shortList)` rather than `shortList.listCopy()`. This may seem strange—why not make `listCopy` an ordinary method? The answer is that an ordinary method could not copy the empty list (because the empty list is represented by the null reference).

### A Second Copy Method, Returning Both Head and Tail References

Here's a second way to copy a list, with a slightly different specification:

`listCopyWithTail`

◆ **listCopyWithTail**

```
public static IntNode[] listCopyWithTail(IntNode source)
```

Copy a list, returning both a head and tail reference for the copy.

**Parameter:**

`source` – the head reference for a linked list that will be copied (which may be an empty list where `source` is null)

**Returns:**

The method has made a copy of the linked list starting at `source`. The return value is an array where the `[0]` element is a head reference for the copy and the `[1]` element is a tail reference for the copy.

**Throws:** `OutOfMemoryError`

Indicates that there is insufficient memory for the new list.

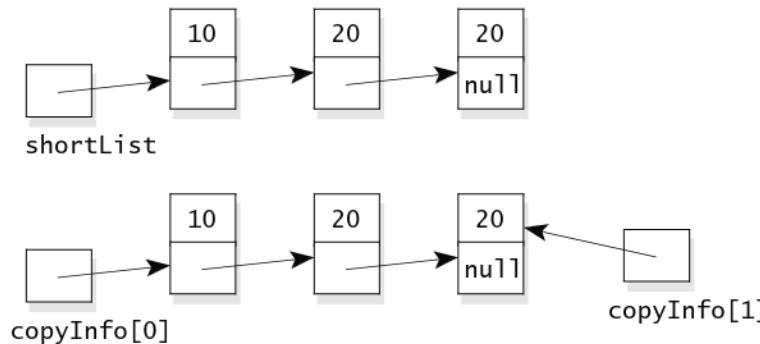
The `listCopyWithTail` method makes a copy of a list, but the return value is more than a head reference for the copy. Instead, the return value is an array with two components. The [0] component of the array contains the head reference for the new list, and the [1] component contains the tail reference for the new list. The `listCopyWithTail` method is important because many algorithms must copy a list and obtain access to both the head and tail nodes of the copy.

As an example, a program can create a small list and then create a copy with both a head and tail reference for the copy:

```
IntNode shortList;
IntNode copyInfo[ ];

shortList = new IntNode(10, null);
shortList.addNodeAfter(20);
shortList.addNodeAfter(20);
copyInfo = IntNode.listCopyWithTail(source);
```

At this point, `copyInfo[0]` is the head reference for a copy of the short list, and `copyInfo[1]` is the tail reference for the same list:



The implementation of `listCopyWithTail` is shown in the first part of Figure 4.10. It's nearly the same as `listCopy`, except there is an extra local variable called `answer`, which is an array of two `IntNode` components. These two components are set to the head and tail of the new list, and the method finishes with the return statement: `return answer`.

## 1 PROGRAMMING TIP

### A METHOD CAN RETURN AN ARRAY

The return value from a method can be an array. This is useful when the method returns more than one piece of information. For example, `listCopyWithTail` returns an array with two components, containing the head and tail references for a new list.

**FIGURE 4.10** A Second Static Method to Copy a Linked List

### Implementation

```
public static IntNode[ ] listCopyWithTail(IntNode source)
{
    // Notice that the return value is an array of two IntNode components.
    // The [0] component is the head reference for the new list and
    // the [1] component is the tail reference for the new list.
    // Also notice that the answer array is automatically initialized to contain
    // two null values. Arrays with components that are references are always
    // initialized this way.
    IntNode copyHead;
    IntNode copyTail;
    IntNode[ ] answer = new IntNode[2];

    // Handle the special case of an empty list.
    if (source == null)
        return answer; // The answer has two null references.

    // Make the first node for the newly created list.
    copyHead = new IntNode(source.data, null);
    copyTail = copyHead;

    // Make the rest of the nodes for the newly created list.
    while (source.link != null)
    {
        source = source.link;
        copyTail.addNodeAfter(source.data);
        copyTail = copyTail.link;
    }

    // Return the head and tail references for the new list.
    answer[0] = copyHead;
    answer[1] = copyTail;
    return answer;
}
```

---

### Copying Part of a Linked List

Sometimes a program needs to copy only part of a linked list rather than the entire list. The task can be done by a static method, `listPart`, which copies part of a list, as specified next.

**◆ listPart***listPart*

```
public static IntNode[ ] listPart  
    (IntNode start, IntNode end)
```

Copy part of a list, providing a head and tail reference for the new copy.

**Parameters:**

start and end – references to two nodes of a linked list

**Precondition:**

start and end are non-null references to nodes on the same linked list, with the start node at or before the end node.

**Returns:**

The method has made a copy of part of a linked list, from the specified start node to the specified end node. The return value is an array in which the [0] component is a head reference for the copy and the [1] component is a tail reference for the copy.

**Throws: IllegalArgumentException**

Indicates that start and end do not satisfy the precondition.

**Throws: OutOfMemoryError**

Indicates that there is insufficient memory for the new list.

The `listPart` implementation is given as part of the complete `IntNode` class in Figure 4.11. In all, there is one constructor, five ordinary methods, and six static methods. The class is placed in a package called `edu.colorado.nodes`.

## Using Linked Lists

Any program can use linked lists created from our nodes. Such a program must have this import statement:

```
import edu.colorado.nodes.IntNode;
```

The program can then use the various methods to build and manipulate linked lists. In fact, the `edu.colorado.nodes` package includes many different kinds of nodes: `IntNode`, `DoubleNode`, `CharNode`, etc. You can get these classes from <http://www.cs.colorado.edu/~main/edu/colorado/nodes>. (There is also a special kind of node that can handle many different kinds of data, but you'll have to wait until Chapter 5 for that.)

*nodes with  
different kinds of  
data*

To use our nodes, a programmer must have some understanding of linked lists and of our specific nodes. This is because we intend to use the node classes ourselves to build various collection classes. The different collection classes we build can be used by any programmer, with no knowledge of nodes and linked lists. This is what we will do in the rest of the chapter, providing two ADTs that use the linked lists.

**FIGURE 4.11** Specification and Implementation of the IntNode Class

### Class IntNode

❖ **public class IntNode from the package edu.colorado.nodes**

An IntNode provides a node for a linked list with integer data in each node. Lists can be of any length, limited only by the amount of free memory on the heap. But beyond Integer.MAX\_VALUE, the answer from listLength is incorrect because of arithmetic overflow.

#### Specification

◆ **Constructor for the IntNode**

```
public IntNode(int initialValue, IntNode initialLink)
```

Initialize a node with specified initial data and a link to the next node. Note that the initialLink may be the null reference, which indicates that the new node has nothing after it.

**Parameters:**

initialData – the initial data of this new node

initialLink – a reference to the node after this new node (this reference may be null to indicate that there is no node after this new node)

**Postcondition:**

This new node contains the specified data and a link to the next node.

◆ **addNodeAfter**

```
public void addNodeAfter(int element)
```

Modification method to add a new node after this node.

**Parameters:**

element – the data to be placed in the new node

**Postcondition:**

A new node has been created and placed after this node. The data for the new node is element. Any other nodes that used to be after this node are now after the new node.

**Throws:** OutOfMemoryError

Indicates that there is insufficient memory for a new IntNode.

◆ **getData**

```
public int getData()
```

Accessor method to get the data from this node.

**Returns:**

the data from this node

◆ **getLink**

```
public IntNode getLink()
```

Accessor method to get a reference to the next node after this node.

**Returns:**

a reference to the node after this node (or the null reference if there is nothing after this node)

(FIGURE 4.11 continued)

◆ **listCopy**

```
public static IntNode listCopy(IntNode source)
```

Copy a list.

**Parameter:**

source – the head reference for a linked list that will be copied (which may be an empty list where source is null)

**Returns:**

The method has made a copy of the linked list starting at source. The return value is the head reference for the copy.

**Throws:** OutOfMemoryError

Indicates that there is insufficient memory for the new list.

◆ **listCopyWithTail**

```
public static IntNode[ ] listCopyWithTail(IntNode source)
```

Copy a list, returning both a head and tail reference for the copy.

**Parameters:**

source – the head reference for a linked list that will be copied (which may be an empty list where source is null)

**Returns:**

The method has made a copy of the linked list starting at source. The return value is an array where the [0] element is a head reference for the copy and the [1] element is a tail reference for the copy.

**Throws:** OutOfMemoryError

Indicates that there is insufficient memory for the new list.

◆ **listLength**

```
public static int listLength(IntNode head)
```

Compute the number of nodes in a linked list.

**Parameter:**

head – the head reference for a linked list (which may be an empty list with a null head)

**Returns:**

the number of nodes in the list with the given head

**Note:**

A wrong answer occurs for lists longer than `Int.MAX_VALUE` because of arithmetic overflow.

(continued)

**210 Chapter 4 / Linked Lists***(FIGURE 4.11 continued)***◆ listPart**

```
public static IntNode[ ] listPart(IntNode start, IntNode end)
```

Copy part of a list, providing a head and tail reference for the new copy.

**Parameters:**

start and end – references to two nodes of a linked list

**Precondition:**

start and end are non-null references to nodes on the same linked list, with the start node at or before the end node.

**Returns:**

The method has made a copy of part of a linked list, from the specified start node to the specified end node. The return value is an array where the [0] component is a head reference for the copy and the [1] component is a tail reference for the copy.

**Throws: IllegalArgumentException**

Indicates that start and end do not satisfy the precondition.

**Throws: OutOfMemoryError**

Indicates that there is insufficient memory for the new list.

**◆ listPosition**

```
public static IntNode listPosition(IntNode head, int position)
```

Find a node at a specified position in a linked list.

**Parameters:**

head – the head reference for a linked list (which may be an empty list with a null head)  
position – a node number

**Precondition:**

position > 0

**Returns:**

The return value is a reference to the node at the specified position in the list. (The head node is position 1, the next node is position 2, and so on.) If there is no such position (because the list is too short), then the null reference is returned.

**Throws: IllegalArgumentException**

Indicates that position is 0.

**◆ listSearch**

```
public static IntNode listSearch(IntNode head, int target)
```

Search for a particular piece of data in a linked list.

**Parameters:**

head – the head reference for a linked list (which may be an empty list with a null head)  
target – a piece of data to search for

**Returns:**

The return value is a reference to the first node that contains the specified target. If there is no such node, the null reference is returned.

(continued)



**212 Chapter 4 / Linked Lists**

(FIGURE 4.11 continued)

```
public class IntNode
{
    // Invariant of the IntNode class:
    // 1. The node's integer data is in the instance variable data.
    // 2. For the final node of a list, the link part is null.
    // Otherwise, the link part is a reference to the next node of the list.
    private int data;
    private IntNode link;

    public IntNode(int initialValue, IntNode initialLink)
    {
        data = initialValue;
        link = initialLink;
    }

    public void addNodeAfter(int element)
    {
        link = new IntNode(element, link);
    }

    public int getData()
    {
        return data;
    }

    public IntNode getLink()
    {
        return link;
    }

    public static IntNode listCopy(IntNode source)
    || See the implementation in Figure 4.9 on page 203.

    public static IntNode[ ] listCopyWithTail(IntNode source)
    || See the implementation in Figure 4.10 on page 206.

    public static int listLength(IntNode head)
    || See the implementation in Figure 4.4 on page 195.
```

(continued)

(FIGURE 4.11 continued)

```
public static IntNode[ ] listPart(IntNode start, IntNode end)
{
    // Notice that the return value is an array of two IntNode components.
    // The [0] component is the head reference for the new list,
    // and the [1] component is the tail reference for the new list.
    IntNode copyHead;
    IntNode copyTail;
    IntNode[ ] answer = new IntNode[2];

    // Check for illegal null at start or end.
    if (start == null)
        throw new IllegalArgumentException("start is null");
    if (end == null)
        throw new IllegalArgumentException("end is null");

    // Make the first node for the newly created list.
    copyHead = new IntNode(start.data, null);
    copyTail = copyHead;

    // Make the rest of the nodes for the newly created list.
    while (start != end)
    {
        start = start.link;
        if (start == null)
            throw new IllegalArgumentException
                ("end node was not found on the list");
        copyTail.addNodeAfter(start.data);
        copyTail = copyTail.link;
    }

    // Return the head and tail reference for the new list.
    answer[0] = copyHead;
    answer[1] = copyTail;
    return answer;
}

public static IntNode listPosition(IntNode head, int position)
|| See the implementation in Figure 4.8 on page 199.

public static IntNode listSearch(IntNode head, int target)
|| See the implementation in Figure 4.6 on page 198.
```

(continued)

**214 Chapter 4 / Linked Lists***(FIGURE 4.11 continued)*

```
public void removeNodeAfter()
{
    link = link.link;
}

public void setData(int newData)
{
    data = newData;
}

public void setLink(IntNode newLink)
{
    link = newLink;
}
```

---

**Self-Test Exercises for Section 4.3**

11. Look at <http://www.cs.colorado.edu/~main/edu/colorado/nodes>. How many different kinds of nodes are there? If you implemented one of these nodes, what extra work would be required to implement another?
12. Suppose `locate` is a reference to a node in a linked list (and it is not the null reference). Write an assignment statement that will make `locate` move to the next node in the list. You should write two versions of the assignment—one that can appear in the `IntNode` class itself and another that can appear outside of the class. What do your assignment statements do if `locate` was already referring to the last node in the list?
13. Which of the node methods use `new` to allocate at least one new node? Check your answer by looking at the documentation in Figure 4.11 on page 208 (to see which methods can throw an `OutOfMemoryError`).
14. Suppose `head` is a head reference for a linked list with just one node. What will `head` be after `head = head.getLink()`?
15. What technique would you use if a method needs to return more than one `IntNode`, such as a method that returns both a head and tail reference for a list.
16. Suppose `head` is a head reference for a linked list. Also suppose `douglass` and `adams` are two other `IntNode` variables. Write one assignment statement that will make `douglass` refer to the first node in the list that contains the number 42. Write a second assignment statement that will make `adams` refer to the 42<sup>nd</sup> node of the list. If these nodes don't exist, then the assignments should set the variables to null.

17. Suppose a program sets up a linked list with an `IntNode` variable called `head` to refer to the first node of the list (or `head` is `null` if the list is empty). Write a few lines of Java code that will print all the numbers.
18. Implement a new static method for the `IntNode` class with one parameter that is a head reference for a linked list. The return value of the method is the number of times that the number 42 appears on the list.
19. Implement a new static method for the `IntNode` class with one parameter that is a head reference for a linked list. The return value of the method is a boolean value that is true if the list contains at least one copy of 42; otherwise, it is false.
20. Implement a new static method for the `IntNode` class with one parameter that is a head reference for a linked list. The return value of the method is the sum of all the numbers on the list.
21. Implement a new static method for the `IntNode` class with one parameter that is a head reference for a linked list. The function adds a new node to the tail of the list with the data equal to 42.
22. Implement a new static method for the `IntNode` class with one parameter that is a head reference for a linked list. The method removes the first node after the head that contains the number 42 (if there is such a node). Do not call any other methods to do any of the work.
23. Suppose `p`, `q`, and `r` are all references to nodes in a linked list with 15 nodes. The variable `p` refers to the first node, `q` refers to the 8<sup>th</sup> node, and `r` refers to the last node. Write a few lines of code that will make a new copy of the list. Your code should set *three* new variables called `x`, `y`, and `z` so that `x` refers to the first node of the copy, `y` refers to the 8<sup>th</sup> node of the copy, and `z` refers to the last node of the copy. Your code may not contain any loops, but it can use the other `IntNode` methods.

## 4.4 THE BAG ADT WITH A LINKED LIST

We're ready to write an ADT that is implemented with a linked list. We'll start with the familiar bag ADT, which we previously implemented with an array (in Section 3.2). At the end of this chapter, we'll compare the advantages and disadvantages of these different implementations. But first, let's see how a linked list is used in our second bag implementation.

### Our Second Bag—Specification

The advantage of using a familiar ADT is that you already know most of the specification. The specification, given in Figure 4.12, is nearly identical to our previous bag. The major difference is that our new bag has no worries about capacity: There is no initial capacity and no need for an `ensureCapacity`

**216 Chapter 4 / Linked Lists**

the new bag  
class is called  
*IntLinkedBag*

method. This is because our planned implementation—storing the bag’s elements on a linked list—can easily grow and shrink by adding and removing nodes from the linked list.

The new bag class will be called *IntLinkedBag*, meaning that the underlying elements are integers, the implementation will use a linked list, and the collection itself is a bag. The *IntLinkedBag* will be placed in the same package that we used in Chapter 3, `edu.colorado.collections`, as shown in the specification of Figure 4.12.

---

**FIGURE 4.12** Specification and Implementation of the *IntLinkedBag* Class

### Class *IntLinkedBag*

◆ **public class IntLinkedBag from the package edu.colorado.collections**

An *IntLinkedBag* is a collection of `int` numbers.

**Limitations:**

- (1) Beyond `Int.MAX_VALUE` elements, `countOccurrences`, `size`, and `grab` are wrong.
- (2) Because of the slow linear algorithms of this class, large bags have poor performance.

### Specification

◆ **Constructor for the *IntLinkedBag***

```
public IntLinkedBag()
```

Initialize an empty bag.

**Postcondition:**

This bag is empty.

◆ **add**

```
public void add(int element)
```

Add a new element to this bag.

**Parameter:**

`element` – the new element that is being added

**Postcondition:**

A new copy of the element has been added to this bag.

**Throws:** `OutOfMemoryError`

Indicates insufficient memory for adding a new element.

(continued)

(FIGURE 4.12 continued)

◆ **addAll**

`public void addAll(IntLinkedBag addend)`

Add the contents of another bag to this bag.

**Parameter:**

`addend` – a bag whose contents will be added to this bag

**Precondition:**

The parameter, `addend`, is not null.

**Postcondition:**

The elements from `addend` have been added to this bag.

**Throws:** `NullPointerException`

Indicates that `addend` is null.

**Throws:** `OutOfMemoryError`

Indicates insufficient memory to increase the size of this bag.

◆ **addMany**

`public void addMany(int... elements)`

Add a variable number of new elements to this bag. If these new elements would take this bag beyond its current capacity, then the capacity is increased before adding the new elements.

**Parameter:**

`elements` – a variable number of new elements that are all being added

**Postcondition:**

New copies of all the elements have been added to this bag.

**Throws:** `OutOfMemoryError`

Indicates insufficient memory to increase the size of this bag.

◆ **clone**

`public IntLinkedBag clone()`

Generate a copy of this bag.

**Returns:**

The return value is a copy of this bag. Subsequent changes to the copy will not affect the original, nor vice versa. The return value must be typecast to an `IntLinkedBag` before it is used.

**Throws:** `OutOfMemoryError`

Indicates insufficient memory for creating the clone.

◆ **countOccurrences**

`public int countOccurrences(int target)`

Accessor method to count the number of occurrences of a particular element in this bag.

**Parameter:**

`target` – the element that needs to be counted

**Returns:**

the number of times that `target` occurs in this bag

(continued)

**218 Chapter 4 / Linked Lists***(FIGURE 4.12 continued)***◆ grab**

```
public int grab()
```

Accessor method to retrieve a random element from this bag.

**Precondition:**

This bag is not empty.

**Returns:**

a randomly selected element from this bag

**Throws: IllegalStateException**

Indicates that the bag is empty.

**◆ remove**

```
public boolean remove(int target)
```

Remove one copy of a specified element from this bag.

**Parameter:**

target – the element to remove from the bag

**Postcondition:**

If target was found in this bag, then one copy of target has been removed, and the method returns true. Otherwise, this bag remains unchanged, and the method returns false.

**◆ size**

```
public int size()
```

Accessor method to determine the number of elements in this bag.

**Returns:**

the number of elements in this bag

**◆ union**

```
public static IntLinkedBag union(IntLinkedBag b1, IntLinkedBag b2)
```

Create a new bag that contains all the elements from two other bags.

**Parameters:**

b1 – the first of two bags

b2 – the second of two bags

**Precondition:**

Neither b1 nor b2 is null.

**Returns:**

a new bag that is the union of b1 and b2

**Throws: NullPointerException**

Indicates that one of the arguments is null.

**Throws: OutOfMemoryError**

Indicates insufficient memory for the new bag.

## The grab Method

The new bag has one other minor change, which is specified as part of Figure 4.12: Just for fun, we've included a new method called `grab`, which returns a randomly selected element from a bag. Later we'll use the `grab` method in some game-playing programs.

## Our Second Bag—Class Declaration

Our plan has been laid. We will implement the new bag by storing the elements in a linked list. The class will have two private instance variables: (1) a reference to the head of a linked list that contains the elements of the bag, and (2) an `int` variable that keeps track of the length of the list. The second instance variable isn't really needed, because we could use `listLength` to determine the length of the list. But by keeping the length in an instance variable, the length can be quickly determined by accessing the variable (a constant time operation). This is in contrast to actually counting the length by traversing the list (a linear time operation).

In any case, we can now write an outline for our implementation. The class goes in the package `edu.colorado.collections`, and we import the node class from `edu.colorado.nodes.IntNode`. Then we declare our new bag class with two instance variables:

```
package edu.colorado.collections;
import edu.colorado.nodes.IntNode;

class IntLinkedBag
{
    private IntNode head;      // Head reference for the list
    private int manyNodes;     // Number of nodes in the list

    || Method implementations will be placed here later.
}
```

To avoid confusion over how we are using our linked list, we now make an explicit statement of the invariant for our second design of the bag ADT.

### Invariant for the Second Bag ADT

1. The elements in the bag are stored in a linked list.
2. The head reference of the list is stored in the instance variable `head`.
3. The total number of elements in the list is stored in the instance variable `manyNodes`.

### The Second Bag—Implementation

With our invariant in mind, we can implement each of the methods, starting with the constructor. The key to simple implementations is to use the node methods whenever possible.

**Constructor.** The constructor sets `head` to be the null reference (indicating the empty list) and sets `manyNodes` to 0. Actually, these two values (null and zero) are the default values for the instance variables, so one possibility is to not implement the constructor at all. When we implement no constructor, Java provides an automatic no-arguments constructor that initializes all instance variables to their default values. If we take this approach, then our implementation should include a comment to indicate that we are using Java's automatic no-arguments constructor.

However, we will actually implement the constructor:

```
constructor
public IntLinkedBag( )
{
    head = null;
    manyNodes = 0;
}
```

Having an actual implementation makes it easier to make future changes. Also, without the implementation, we could not include a Javadoc comment to specify exactly what the constructor does.

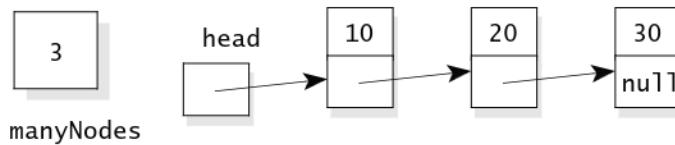
**The `clone` Method.** The `clone` method needs to create a copy of a bag. The `IntLinkedBag` `clone` method will follow the pattern introduced in Chapter 2 on page 82. Therefore, the start of the `clone` method is the code shown here:

```
clone method
public IntLinkedBag clone( )
{ // Clone an IntLinkedBag.
    IntLinkedBag answer;

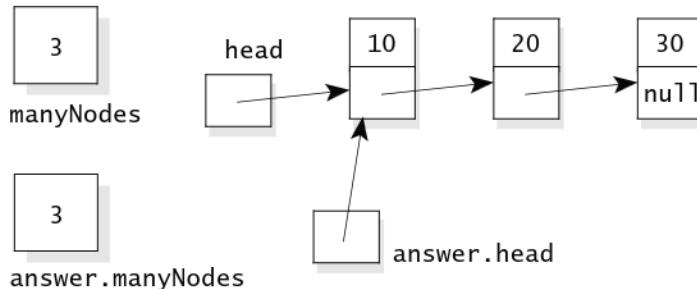
    try
    {
        answer = (IntLinkedBag) super.clone( );
    }
    catch (CloneNotSupportedException e)
    { // This exception should not occur. But if it does, it would
       // probably indicate a programming error that made
       // super.clone unavailable. The most common error would be
       // forgetting the "Implements Cloneable" clause.
        throw new RuntimeException
            ("This class does not implement Cloneable.");
    }
    ...
}
```

This is the same as the start of the `clone` method for our Chapter 3 bag. As with the Chapter 3 bag, this code uses the `super.clone` method to make `answer` be an exact copy of the bag that activated the `clone` method. With the Chapter 3 bag, we needed some extra statements at the end of the `clone` method; otherwise, the original bag and the clone would share the same array.

Our new bag, using a linked list, runs into a similar problem. To see this problem, consider a bag that contains three elements:



Now suppose we activate `clone()` to create a copy of this bag. The `clone` method executes the statement `answer = (IntLinkedBag) super.clone()`. What does `super.clone()` do? It creates a new `IntLinkedBag` object, and `answer` will refer to this new `IntLinkedBag`. But the new `IntLinkedBag` has instance variables (`answer.manyNodes` and `answer.head`) that are merely copied from the original. So, after `answer = (IntLinkedBag) super.clone()`, the situation looks like this (where `manyNodes` and `head` are the instance variables from the original bag that activated the `clone` method):



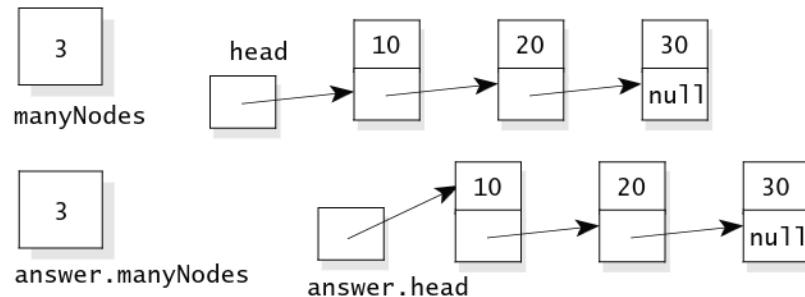
As you can see, `answer.head` refers to the original's head node. Subsequent changes to the linked list will affect both the original and the clone. This is incorrect behavior for a clone. To fix the problem, we need an additional statement before the return of the `clone` method. The purpose of the statement is to create a new linked list for the clone's `head` instance variable to refer to. Here's the statement:

```
answer.head = IntNode.listCopy(head);
```

This statement activates the `listCopy` method. The argument, `head`, is the head reference from the linked list of the bag we are copying. When the assignment

## 222 Chapter 4 / Linked Lists

statement finishes, `answer.head` will refer to the head node of the new list. Here's the situation after the copying:



The new linked list for `answer` was created by copying the original linked list. Subsequent changes to `answer` will not affect the original, nor will changes to the original affect `answer`. The complete `clone` method, including the extra statement at the end, is shown in Figure 4.13.

**FIGURE 4.13** Implementation of the Second Bag's `clone` Method

### Implementation

```

public IntLinkedBag clone( )
{
    // Clone an IntLinkedBag.
    IntLinkedBag answer;

    try
    {
        answer = (IntLinkedBag) super.clone( );
    }
    catch (CloneNotSupportedException e)
    {   // This exception should not occur. But if it does, it would probably indicate a
       // programming error that made super.clone unavailable. The most common
       // error would be forgetting the "Implements Cloneable"
       // clause at the start of this class.
        throw new RuntimeException
            ("This class does not implement Cloneable.");
    }

    answer.head = IntNode.listCopy(head); ← This step creates a new
  linked list for answer. The
  new linked list is
  separate from the original
  array so that subsequent
  changes to one will not
  affect the other.

    return answer;
}

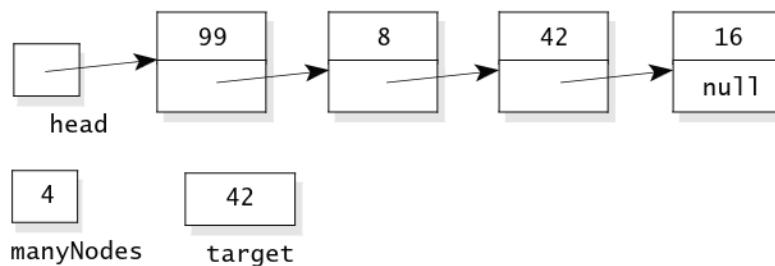
```

**PROGRAMMING TIP****CLONING A CLASS THAT CONTAINS A LINKED LIST**

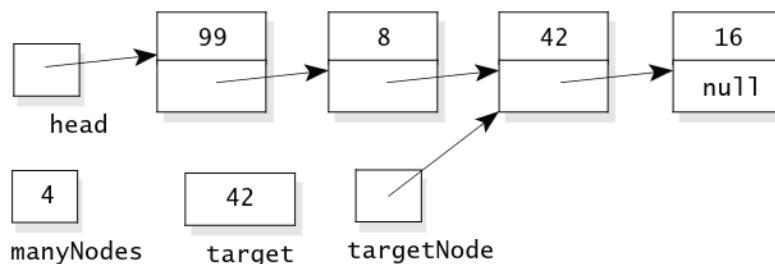
If the instance variables of a class contain a linked list, then the `clone` method needs extra work before it returns. The extra work creates a new linked list for the clone's instance variable to refer to.

**The remove Method.** There are two approaches to implementing the `remove` method. The first approach uses the removal methods we have already seen—changing the head if the removed element is at the head of the list and using the ordinary `removeNodeAfter` to remove an element that is farther down the line. This first approach is fine, although it does require a bit of thought because `removeNodeAfter` requires a reference to the node that is just *before* the element you want to remove. We could certainly find this “before” node, but not by using the node's `listSearch` method.

The second approach actually uses `listSearch` to obtain a reference to the node that contains the element to be deleted. For example, suppose our target is the number 42 in this bag:



Our approach begins by setting a local variable named `targetNode` to refer to the node that contains our target. This is accomplished with the assignment `targetNode = listSearch(head, target)`. After the assignment, the `targetNode` is set this way:



Now we can remove the target from the list with two more steps. First, copy the data from the head node to the target node, as shown at the top of the next page.

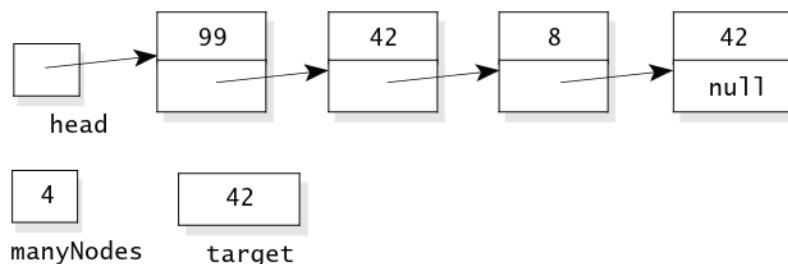


**PROGRAMMING TIP** 
**HOW TO CHOOSE BETWEEN DIFFERENT APPROACHES**

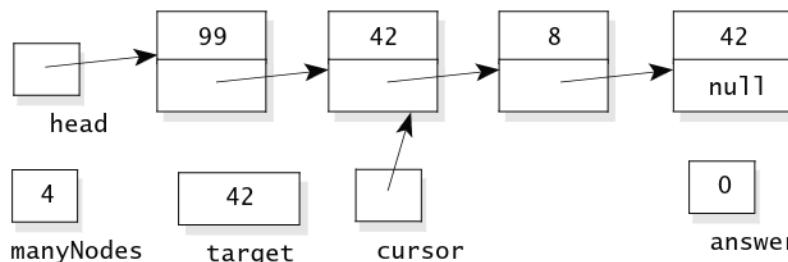
We have two possible approaches for the `remove` method. How do we select the better approach? Normally, when two different approaches have equal efficiency, we will choose the approach that makes better use of the node's methods. This saves us work and also reduces the chance of new errors from writing new code to do an old job. In the case of `remove`, we chose the second approach because it made better use of `listSearch`.

**The `countOccurrences` Method.** Two possible approaches come to mind for the `countOccurrences` method. One of the approaches simply steps through the linked list one node at a time, checking each piece of data to see whether it is the sought-after target. We count the occurrences of the target and return the answer. The second approach uses `listSearch` to find the first occurrence of the target, then uses `listSearch` again to find the next occurrence, and so on, until we have found all occurrences of the target. The second approach makes better use of the node's methods, so that is the approach we will take.

As an example of the second approach to the `countOccurrences` method, suppose we want to count the number of occurrences of 42 in this bag:

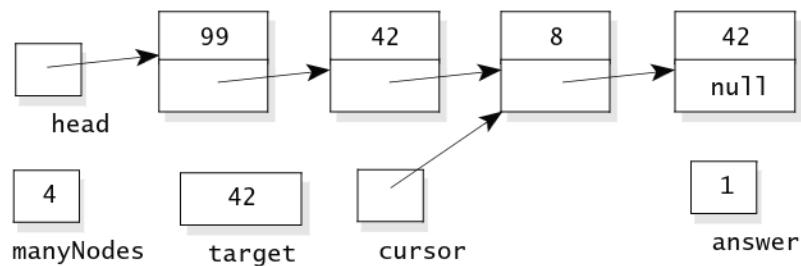


We'll use two local variables: `answer`, which keeps track of the number of occurrences we have seen so far, and `cursor`, which is a reference to a node in the list. We initialize `answer` to 0, and we use `listSearch` to make `cursor` refer to the first occurrence of the target (or to be `null` if there are no occurrences). After this initialization, we have this situation:

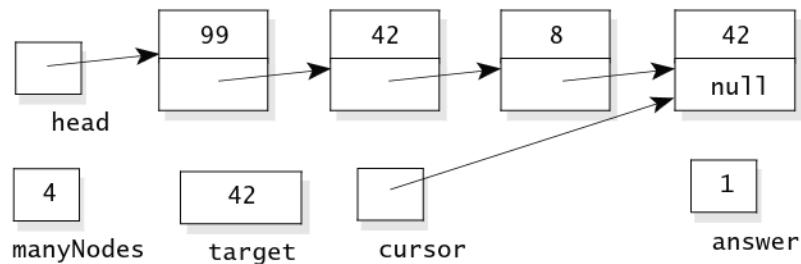


## 226 Chapter 4 / Linked Lists

Next we enter a loop. The loop stops when `cursor` becomes `null`, indicating that there are no more occurrences of the target. Each time through the loop we do two steps: (1) Add 1 to `answer`, and (2) move `cursor` to refer to the next occurrence of the target (or to be `null` if there are no more occurrences). Can we use a node method to execute Step 2? At first, it might seem that the node methods are of no use since `listSearch` finds the *first* occurrence of a given target. But there is an approach that will use `listSearch` together with the `cursor` to find the *next* occurrence of the target. The approach begins by moving `cursor` to the next node in the list, using the statement `cursor = cursor.getLink()`. In our example, this results in the following:



As you can see, `cursor` now refers to a node in the middle of a linked list. But any time that a variable refers to a node in the middle of a linked list, we can pretend that the node is the head of a smaller linked list. In our example, `cursor` refers to the head of a two-element list containing the numbers 8 and 42. Therefore, we can use `cursor` as an argument to `listSearch` in the assignment statement `cursor = IntNode.listSearch(cursor, target)`. This statement moves `cursor` to the next occurrence of the target. This occurrence could be at the `cursor`'s current spot, or it could be farther down the line. In our example, the next occurrence of 42 is farther down the line, so `cursor` is moved as shown here:



Eventually there will be no more occurrences of the target and `cursor` becomes `null`, ending the loop. At that point the method returns `answer`. The complete implementation of `countOccurrences` is shown in Figure 4.15.

**FIGURE 4.15** Implementation of countOccurrences for the Second Version of the BagImplementation

```
public int countOccurrences(int target)
{
    int answer;
    IntNode cursor;

    answer = 0;
    cursor = IntNode.listSearch(head, target);
    while (cursor != null)
    { // Each time that cursor is not null, we have another occurrence of target, so we
      // add 1 to answer and then move cursor to the next occurrence of the target.
      answer++;
      cursor = cursor.getLink();
      cursor = IntNode.listSearch(cursor, target);
    }
    return answer;
}
```

**Finding the Next Occurrence of an Element**

**The situation:** A variable named cursor refers to a node in a linked list that contains a particular element called target.

**The task:** Make cursor refer to the next occurrence of target (or null if there are no more occurrences).

**The solution:**

```
cursor = cursor.getLink();
cursor = IntNode.listSearch(cursor, target);
```

**The grab Method.** The bag has a new `grab` method, specified here:

◆ **grab**

```
public int grab()
```

Accessor method to retrieve a random element from this bag.

**Precondition:**

This bag is not empty.

**Returns:**

a randomly selected element from this bag

**Throws: IllegalStateException**

Indicates that the bag is empty.

The implementation will start by generating a random `int` value between 1 and the size of the bag. The random value can then be used to select a node from the bag, and we'll return the data from the selected node. So the body of the method will look something like this:

```
i = some random int value between 1 and the size of the bag;  
cursor = listPosition(head, i);  
return cursor.getData();
```

Of course, the trick is to generate “some random `int` value between 1 and the size of the bag.” The Java Class Libraries can help. Within `java.lang.math` is a method that (sort of) generates random numbers, with this specification:

◆ **Math.random**

```
public static double random()
```

Generates a pseudorandom number in the range 0.0 to 1.0.

**Returns:**

a pseudorandom number in the range 0.0 to 1.0 (this return value may be zero, but it's always less than 1.0)

The values returned by `Math.random` are not truly random. They are generated by a simple rule. But the numbers *appear* random, and so the method is referred to as a **pseudorandom number generator**. For most applications, a pseudorandom number generator is a close enough approximation to a true random number generator. In fact, a pseudorandom number generator has one advantage over a true random number generator: The sequence of numbers it produces is repeatable. If run twice with the same initial conditions, a pseudorandom number generator will produce exactly the same sequence of numbers. This is handy when you are debugging programs that use these sequences. When an error is discovered, the corrected program can be tested with the *same* sequence of pseudorandom numbers that produced the original error.

*Math.random*

But at this point we don't need a complete memoir on pseudorandom numbers. All we need is a way to use `Math.random` to generate a number between 1 and the size of the bag. The following assignment statement does the trick:

```
// Set i to a random number from 1 to the size of the bag:  
i = (int) (Math.random() * manyNodes) + 1;
```

Let's look at how the expression works. The method `Math.random` gives us a number that's in the range 0.0 to 1.0. The value is actually in the "half-open range" of [0 .. 1), which means that the number could be from zero up to, but not including, 1. Therefore, the expression `Math.random() * manyNodes` is in the range zero to `manyNodes`—or to be more precise, from zero up to, but not including, `manyNodes`.

The operation `(int)` is an operation that truncates a double number, keeping only the integer part. Therefore, `(int) (Math.random() * manyNodes)` is an `int` value that could be from zero to `manyNodes`-1. The expression cannot actually be `manyNodes`. Since we want a number from 1 to `manyNodes`, we add 1, resulting in `i = (int) (Math.random() * manyNodes) + 1`. This assignment statement is used in the complete `grab` implementation shown in Figure 4.16.

### The Second Bag—Putting the Pieces Together

The remaining methods are straightforward. For example, the `size` method just returns `manyNodes`. All these methods are given in the complete implementation in Figure 4.17. Take particular notice of how the bag's `addAll` method is implemented. The implementation makes a copy of the linked list for the bag that's being added. This copy is then attached at the front of the linked list for the bag that's being added to. The bag's `union` method is implemented by using the `addAll` method.

---

**FIGURE 4.16** Implementation of a Method to Grab a Random Element

#### Implementation

```
public int grab()  
{  
    int i; // A random value between 1 and the size of the bag  
    IntNode cursor;  
  
    if (manyNodes == 0)  
        throw new IllegalStateException("Bag size is zero.");  
  
    i = (int) (Math.random() * manyNodes) + 1;  
    cursor = IntNode.listPosition(head, i);  
    return cursor.getData();  
}
```

---

**FIGURE 4.17** Implementation of Our Second Bag ClassImplementation

```
// FILE: IntLinkedBag.java from the package edu.colorado.collections
// Documentation is available in Figure 4.12 on page 216 or from the IntLinkedBag link at
// http://www.cs.colorado.edu/~main/docs/.
```

```
package edu.colorado.collections;
import edu.colorado.nodes.IntNode;

public class IntLinkedBag implements Cloneable
{
    // INVARIANT for the Bag ADT:
    //   1. The elements in the Bag are stored in a linked list.
    //   2. The head reference of the list is in the instance variable head.
    //   3. The total number of elements in the list is in the instance variable manyNodes.
    private IntNode head;
    private int manyNodes;

    public IntLinkedBag( )
    {
        head = null;
        manyNodes = 0;
    }

    public void add(int element)
    {
        head = new IntNode(element, head);
        manyNodes++;
    }

    public void addAll(IntLinkedBag addend)
    {
        IntNode[ ] copyInfo;

        if (addend == null)
            throw new IllegalArgumentException("addend is null.");
        if (addend.manyNodes > 0)
        {
            copyInfo = IntNode.listCopyWithTail(addend.head);
            copyInfo[1].setLink(head); // Link the tail of the copy to my own head...
            head = copyInfo[0];      // and set my own head to the head of the copy.
            manyNodes += addend.manyNodes;
        }
    }
}
```

(continued)

(FIGURE 4.17 continued)

```
public void addMany(int... elements)
{
    // Activate the ordinary add method for each integer in the elements array.
    for (int i : elements)
        add(i);
}

public IntLinkedBag clone()
|| See the implementation in Figure 4.13 on page 222.
```

```
public int countOccurrences(int target)
|| See the implementation in Figure 4.15 on page 227.
```

```
public int grab()
|| See the implementation in Figure 4.16 on page 229.
```

```
public boolean remove(int target)
|| See the implementation in Figure 4.14 on page 224.
```

```
public int size()
{
    return manyNodes;
}

public static IntLinkedBag union(IntLinkedBag b1, IntLinkedBag b2)
{
    if (b1 == null)
        throw new IllegalArgumentException("b1 is null.");
    if (b2 == null)
        throw new IllegalArgumentException("b2 is null.");

    IntLinkedBag answer = new IntLinkedBag();

    answer.addAll(b1);
    answer.addAll(b2);
    return answer;
}
```

### Self-Test Exercises for Section 4.4

24. Which methods would need to be altered if you wanted to keep the numbers in order from smallest to largest in the bag’s list? Would this allow some other methods to operate more efficiently?
25. Suppose you want to use a bag in which the elements are double numbers instead of integers. How would you do this?
26. Write a few lines of code to declare a bag of integers and place the integers 42 and 8 in the bag. Then grab a random integer from the bag, printing it. Finally, print the size of the bag.
27. In general, which is preferable: an implementation that uses the node methods or an implementation that manipulates a linked list directly?
28. Suppose that `p` is a reference to a node in a linked list of integers and `p.getData()` has a copy of an integer called `d`. Write two lines of code that will move `p` to the next node that contains a copy of `d` (or set `p` to `null` if there is no such node). How can you combine your two statements into just one?
29. Describe the steps taken by `countOccurrences` if the target is not in the bag.
30. Describe one of the boundary values for testing `remove`.
31. Write an expression that will give a random integer between -10 and 10.
32. Do big-*O* time analyses of the bag’s methods.
33. What would go wrong if you forgot to include the `listCopy` statement in the bag’s `clone` method?

## 4.5 PROGRAMMING PROJECT: THE SEQUENCE ADT WITH A LINKED LIST

In Section 3.3 on page 145 we gave a specification for a sequence ADT that was implemented using an array. Now you can reimplement this ADT using a *linked list* as the data structure rather than an array. Start by rereading the ADT’s specification on page 151 and then return here for some implementation suggestions.

### The Revised Sequence ADT—Design Suggestions

Using a linked list to implement the sequence ADT seems natural. We’ll keep the elements stored on a linked list in their sequence order. The “current” element on the list can be maintained by an instance variable that refers to the node that contains the current element. When the `start` method is activated, we set this cursor to refer to the first node of the linked list. When `advance` is activated, we move the cursor to the next node on the linked list.

With this in mind, we propose five private instance variables for the new sequence class:

- The first variable, `manyNodes`, keeps track of the number of nodes in the list.
- `head` and `tail`—These are references to the head and tail nodes of the linked list. If the list has no elements, then these references are both `null`. The reason for the tail reference is the `addAfter` method. Normally this method adds a new element immediately after the current element. But if there is no current element, then `addAfter` places its new element at the tail of the list, so it makes sense to maintain a connection with the list's tail.
- `cursor`—Refers to the node with the current element (or `null` if there is no current element).
- `precursor`—Refers to the node before the current element (or `null` if there is no current element or if the current element is the first node). Can you figure out why we propose a *precursor*? The answer is the `addBefore` method, which normally adds a new element immediately *before* the current element. But there is no `node` method to add a new node before a specified node. We can only add new nodes after a specified node. Therefore, the `addBefore` method will work by adding the new element *after* the precursor node—which is also just *before* the cursor node.

The sequence class you implement could have integer elements, double number elements, or several other possibilities. The choice of element type will determine which kind of node you use for the linked list. If you choose integer elements, then you will use `edu.colorado.nodes.IntNode`. All the node classes, including `IntNode`, are available for you to view at <http://www.cs.colorado.edu/~main/edu/colorado/nodes>.

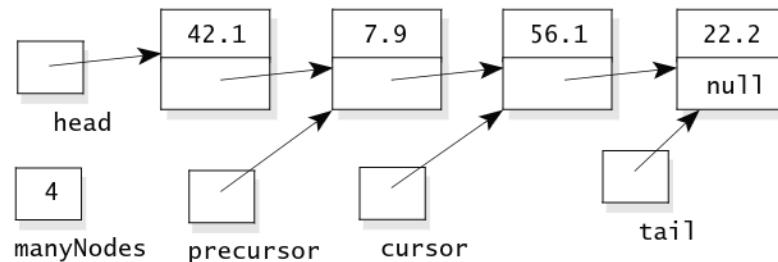
For this programming project, you should use double numbers for the elements and follow these guidelines:

- The name of the class is `DoubleLinkedSeq`.
- You'll use `DoubleNode` for your node class.
- Put your class in the package `edu.colorado.collections`.
- Follow the specification from Figure 4.18 on page 235. This specification is also available at the `DoubleLinkedSeq` link at  
<http://www.cs.colorado.edu/~main/docs/>.

Notice that the specification states a limitation that, beyond `Int.MAX_VALUE` elements, the `size` method does not work (though all other methods should be okay).

## 234 Chapter 4 / Linked Lists

Here's an example of the five instance variables for a sequence with four elements and the current element at the third location:



Notice that `cursor` and `precursor` are *references* to two nodes—one right after the other.

*what is the invariant of the new list ADT?*

Start your implementation by writing the invariant for the new sequence ADT. You might even write the invariant in large letters on a sheet of paper and pin it up in front of you as you work. Each of the methods counts on that invariant being true when the method begins, and each method is responsible for ensuring that the invariant is true when the method finishes.

As you implement each modification method, you might use the following matrix to increase your confidence that the method works correctly.

|                                          | manyNodes | head | tail | cursor | precursor |
|------------------------------------------|-----------|------|------|--------|-----------|
| An empty list                            |           |      |      |        |           |
| A non-empty list with no current element |           |      |      |        |           |
| Current element at the head              |           |      |      |        |           |
| Current element at the tail              |           |      |      |        |           |
| Current element not at head or tail      |           |      |      |        |           |

Here's how to use the matrix: Suppose you have just implemented one of the modification methods, such as `addAfter`. Go through the matrix one row at a time, executing your method with pencil and paper. For example, with the first row of the matrix, you would try `addAfter` to see its behavior for an empty list. As you execute each method by hand, keep track of the five instance variables and put five check marks in the row to indicate that the final values correctly satisfy the invariant.

### The Revised Sequence ADT—Clone Method

The sequence class has a `clone` method to make a new copy of a sequence. The sequence you are copying activates the `clone` method, and we'll call it the "original sequence." As with all `clone` methods, you should start with the pattern from page 82. After activating `super.clone`, the extra work must make a separate copy of the linked list for the clone and correctly set the clone's head, tail, cursor, and precursor. We suggest that you handle the work with the following three cases:

- If the original sequence has no current element, then simply copy the original's linked list with `listCopyWithTail`. Then set both `precursor` and `cursor` to null.
- If the current element of the original sequence is its first element, then copy the original's linked list with `listCopyWithTail`. Then set the `precursor` to null and set `cursor` to refer to the head node of the newly created linked list.
- If the current element of the original sequence is after its first element, then copy the original's linked list in two pieces using `listPart`: The first piece goes from the head node to the precursor; the second piece goes from the cursor to the tail. Put these two pieces together by making the link part of the precursor node refer to the cursor node. The reason for copying in two separate pieces is to easily set the `precursor` and `cursor` of the newly created list.

**FIGURE 4.18** Specification for the Second Version of the DoubleLinkedList Class

#### Class DoubleLinkedList

##### ◆ **public class DoubleLinkedList from the package edu.colorado.collections**

A `DoubleLinkedList` is a sequence of double numbers. The sequence can have a special "current element," which is specified and accessed through four methods that are not available in the bag class (`start`, `getCurrent`, `advance`, and `isCurrent`).

##### **Limitations:**

Beyond `Int.MAX_VALUE` elements, the `size` method does not work.

#### Specification

##### ◆ **Constructor for the DoubleLinkedList**

```
public DoubleLinkedList( )  
Initialize an empty sequence.
```

##### **Postcondition:**

This sequence is empty.

(continued)

**236 Chapter 4 / Linked Lists***(FIGURE 4.18 continued)***◆ addAfter and addBefore**

```
public void addAfter(double element)
public void addBefore(double element)
```

Adds a new element to this sequence either before or after the current element.

**Parameter:**

element – the new element that is being added

**Postcondition:**

A new copy of the element has been added to this sequence. If there was a current element, addAfter places the new element after the current element, and addBefore places the new element before the current element. If there was no current element, addAfter places the new element at the end of the sequence, and addBefore places the new element at the front of the sequence. The new element always becomes the new current element of the sequence.

**Throws: OutOfMemoryError**

Indicates insufficient memory for a new node.

**◆ addAll**

```
public void addAll(DoubleLinkedListSeq addend)
```

Place the contents of another sequence at the end of this sequence.

**Parameter:**

addend – a sequence whose contents will be placed at the end of this sequence

**Precondition:**

The parameter, addend, is not null.

**Postcondition:**

The elements from addend have been placed at the end of this sequence. The current element of this sequence remains where it was, and the addend is also unchanged.

**Throws: NullPointerException**

Indicates that addend is null.

**Throws: OutOfMemoryError**

Indicates insufficient memory to increase the size of the sequence.

**◆ advance**

```
public void advance()
```

Move forward so that the current element is now the next element in the sequence.

**Precondition:**

isCurrent( ) returns true.

**Postcondition:**

If the current element was already the end element of the sequence (with nothing after it), then there is no longer any current element. Otherwise, the new element is the element immediately after the original current element.

**Throws: IllegalStateException**

Indicates that there is no current element, so advance may not be called.

(continued)



**238 Chapter 4 / Linked Lists***(FIGURE 4.18 continued)***◆ removeCurrent**

```
public void removeCurrent()
```

Remove the current element from this sequence.

**Precondition:**

`isCurrent()` returns true.

**Postcondition:**

The current element has been removed from the sequence, and the following element (if there is one) is now the new current element. If there was no following element, then there is now no current element.

**Throws: IllegalStateException**

Indicates that there is no current element, so `removeCurrent` may not be called.

**◆ size**

```
public int size()
```

Accessor method to determine the number of elements in this sequence.

**Returns:**

the number of elements in this sequence

**◆ start**

```
public void start()
```

Set the current element at the front of the sequence.

**Postcondition:**

The front element of this sequence is now the current element (but if the sequence has no elements at all, then there is no current element).

---

**Self-Test Exercises for Section 4.5**

34. Suppose a sequence contains your three favorite numbers, and the current element is the first element. Draw the instance variables of this sequence using our implementation.
35. Write a new method to remove a specified element from a sequence of double numbers. The method has one parameter (the element to remove). After the removal, the current element should be the element after the removed element (if there is one).
36. Which of the sequence methods use the `new` operator to allocate at least one new node?
37. Which of the sequence methods use `DoubleNode.listPart`?

## 4.6 BEYOND SIMPLE LINKED LISTS

### Arrays Versus Linked Lists and Doubly Linked Lists

Many ADTs can be implemented with either arrays or linked lists. Certainly, the bag and the sequence ADT could each be implemented with either approach.

Which approach is better? There is no absolute answer, but there are certain operations that are better performed by arrays and others where linked lists are preferable. This section provides some guidelines.

**Arrays Are Better at Random Access.** The term **random access** refers to examining or changing an arbitrary element that is specified by its position in a list. For example: *What is the 42<sup>nd</sup> element in the list?* Or another example: *Change the element at position 1066 to a 7.* These are constant time operations for an array. But in a linked list, a search for the  $i^{\text{th}}$  element must begin at the head and will take  $O(i)$  time. Sometimes there are ways to speed up the process, but even improvements remain linear time in the worst case.

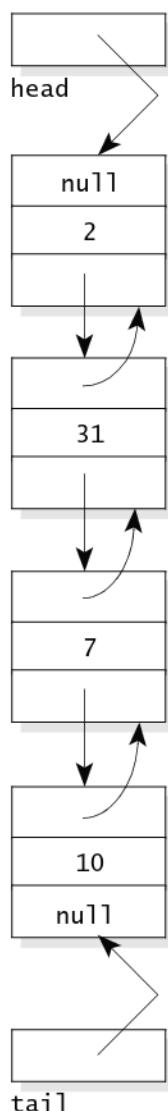
*If an ADT makes significant use of random-access operations, then an array is better than a linked list.*

**Linked Lists Are Better at Additions/Removals at a Cursor.** Our sequence ADT maintains a *cursor* that refers to a “current element.” Typically, a cursor moves through a list one element at a time without jumping around to random locations. If all operations occur at the cursor, then a linked list is preferable to an array. In particular, additions and removals at a cursor generally are linear time for an array (since elements that are after the cursor must *all* be shifted up or back to a new index in the array). But these operations are constant time operations for a linked list. Also remember that effective additions and removals in a linked list generally require maintaining both a cursor and a *precursor* (which refers to the node before the cursor).

*If an ADT’s operations take place at a cursor, then a linked list is better than an array.*

**Resizing Can Be Inefficient for an Array.** A collection class that uses an array generally provides a method to allow a programmer to adjust the capacity as needed. But changing the capacity of an array can be inefficient. The new memory must be allocated and initialized, and the elements are then copied from the old memory to the new memory. If a program can predict the necessary capacity ahead of time, then capacity is not a big problem because the object can be given sufficient capacity from the outset. But sometimes the eventual capacity is unknown, and a program must continually adjust the capacity. In this situation, a linked list has advantages. When a linked list grows, it grows one node at a time, and there is no need to copy elements from old memory to new memory.

*If an ADT is frequently adjusting its capacity, then a linked list might be better than an array.*



**FIGURE 4.19**  
Doubly Linked List

**Doubly Linked Lists Are Better for a Two-Way Cursor.** Sometimes list operations require a cursor that can move forward and backward through a list—a kind of **two-way cursor**. This situation calls for a **doubly linked list**, which is like an ordinary linked list except that each node contains two references: one linking to the next node and one linking to the previous node. An example of a doubly linked list of integers is shown in Figure 4.19 (in the margin) along with references to its head and tail. Here is the start of a declaration for a doubly linked list with integer data:

```
public class IntTwoWayNode
{
    private int data;
    private IntTwoWayNode backlink;
    private IntTwoWayNode forelink;
    ...
}
```

The **backlink** refers to the previous node, and the **forelink** refers to the next node in the list.

*If an ADT's operations take place at a two-way cursor, then a doubly linked list is the best implementation.*

### Dummy Nodes

Many linked list operations require special treatment when the operation occurs on an empty list or the operation occurs at the head or tail node. The special treatment requires extra programming that results in more opportunity for programming errors. Because of this, programmers often create an extra node at the beginning of each list (called a **dummy head node**) and an extra node at the end of each list (called a **dummy tail node**). The data in these two dummy nodes is not part of the list. The nodes are present only to make it easier to program certain operations. Dummy nodes may be used with ordinary (singly linked) lists or with doubly linked lists.

A list with a dummy head node and a dummy tail node has another advantage: The references to the head and tail are set up just once (using the dummy nodes). After they are set up, the head and tail will never need to be changed and never need to refer to some other node. On the other hand, without dummy nodes, the head or tail will need to refer to new nodes whenever a node is inserted or removed at the front or back of the list.

For example, consider the problem of removing a node from a doubly linked list that is maintained with references to both the head and the tail, as in Figure 4.19. If there are no dummy nodes, then there are many cases to consider:

1. Suppose there is only one node in the list, and we are removing it. Then both the head and the tail must be set to null.
2. Suppose we are removing the head node from a list with more than one node. In this case, we must reset the head reference variable to

refer to the second node and also set the backward link of this new head node to null.

3. Suppose we are removing the tail node from a list with more than one node. In this case, we must reset the `tail` reference variable to refer to the next-to-last node and also set the forward link of this new tail to null.
4. Suppose we are removing a node from the middle of a list (neither the head nor the tail). In this case, we can set up two references: `before` (which refers to the node before the node we're removing) and `after` (which refers to the node after the node we're removing). Then, assuming we have methods to set the forward and backward links:

```
before.setForwardLink(after);  
after.setBackwardLink(before);
```

There are lots of cases to consider and lots of cases that might go wrong. But if we have a dummy head node and a dummy tail node, then these dummy nodes are never removed, and from the four cases, only Case 4 is possible. Case 4 is also nice because it never needs to change `head` or `tail` reference variables.

*If you are programming a linked list from scratch (with no preexisting methods to manipulate the list), then consider maintaining both a dummy head node and a dummy tail node. In this case, methods for searching the list may require extra care (to not search the dummy nodes), but both insertion and removal are significantly easier to program.*

## Java's List Classes

The Java Class Libraries have two basic list classes. One version (called `ArrayList`) uses an array as its underlying data structure, and the other (called `LinkedList`) uses a linked list. The two different list classes have many (but not all) methods with the same names and specifications, although the implementations are different. In the next chapter, we'll see a mechanism (called an *interface*) that enforces these identical specifications, but for now we'll list only some of those common methods to allow you to use either of these classes in your own programs.

**Constructors.** As with the `HashSet` (Section 3.5), each list class's default constructor creates an empty list, and a second constructor makes a copy of an another existing collection (either an `ArrayList`, a `LinkedList`, a `HashSet`, or other collections that we'll see later). Also, as with the `HashSet`, when you declare a list, you must specify the type of elements that reside in the list, such as the `String` in `ArrayList<String>` or `LinkedList<String>`.

## 242 Chapter 4 / Linked Lists

**Basic methods.** There are basic methods for adding elements, removing elements, and checking the size (in which E is the type of the elements in the list):

```
boolean add(E element);
void add(int index, E element);
boolean remove(E element);
E remove(int index);
int size();
```

The first version of the add method adds the new element to the tail end of the list and then returns `true`. The `true` return value simply means that the add operation was successful, and this value is always true for the lists (although other kinds of collections, such as the `HashSet`, may fail and return `false`). The second version of add also adds an element, but it does so at a particular location. The number that specifies the location can range from 0 (meaning to insert at the head of the list) to the size of the list (meaning to insert at the tail). If the insertion is not at the tail, then one or more elements in the list will be shifted to higher locations. Can you imagine how the implementation of the add method differs between the `ArrayList` and the `LinkedList`?

There are also two remove methods. The first version searches for a specified element in the list. If the element is found, then its first occurrence is removed and the method returns `true`; otherwise, the method returns `false`. The second remove version has a precondition that the index parameter lies in the range from 0 up to, but not including, the size of the list. The function removes the element at that index. Notice that these indexes start at 0 rather than 1, so the indexing of these lists is similar to an ordinary array.

### ListIterators

#### ListIterator

In Section 3.5, we first saw iterators in the context of a `HashSet`. Java's lists have a similar feature that's implemented via a data type called `ListIterator`. A `ListIterator` has a cursor that steps through its list, but unlike the cursor that we saw for our sequence class (Section 4.5), a `ListIterator`'s cursor generally lies *in between* two of its list elements. For example, suppose we create a list with three strings: "Hiro", "Claire", and "Peter". Then, this code will create a `ListIterator` that lies between "Claire" and "Peter":

```
// Declare a list of Strings and an Iterator for that list:
LinkedList<String> heroes = new LinkedList<String>();
ListIterator<String> it;

// Put elements in the list, and set iterator between "Claire" and "Peter":
heroes.add("Hiro");
heroes.add("Claire");
heroes.add("Peter");
it = heroes.listIterator(2);
```

The iterator returned by `heroes.listIterator(2)` will be on the `heroes` list; the argument, 2, indicates that the iterator will be positioned after the first two elements ("Hiro" and "Claire") but before "Peter".

Once a `ListIterator` is set, a program can use its `add` method to add a new element after the iterator's cursor. With our example list, we could write this code to add "Sylar" between "Claire" and "Peter":

```
it = heroes.ListIterator(2); // Cursor between "Claire" and "Peter"
it.add("Sylar");           // Put "Sylar" at this spot in the list
```

Here are a few other `ListIterator` methods (but not a complete list):

|                               |                                                                 |
|-------------------------------|-----------------------------------------------------------------|
| <code>it.remove()</code>      | <i>// Remove the element that's just after the cursor</i>       |
| <code>it.hasNext()</code>     | <i>// True if the cursor is not at the tail end of the list</i> |
| <code>it.next()</code>        | <i>// Move the cursor forward one element</i>                   |
| <code>it.hasPrevious()</code> | <i>// True if the cursor is not at the start of the list</i>    |
| <code>it.previous()</code>    | <i>// Move the cursor backward one element</i>                  |

`remove,`  
`hasNext, next,`  
`hasPrevious,`  
`previous`

Both the `next` and the `previous` functions return a reference to the element that the cursor jumps over.

### Making the Decision

When you are implementing a new class or you are using one of the JCL classes, you're often faced with a decision about whether to use an array-based list or a linked list. The decision should be based on your knowledge of which operations occur in the ADT, which operations you expect to be performed most often, and whether you expect your arrays to require frequent capacity changes. Figure 4.20 summarizes these considerations.

**FIGURE 4.20** Guidelines for Choosing Between an Array and a Linked List

|                                        |                                             |
|----------------------------------------|---------------------------------------------|
| Frequent random access operations      | Use an array.                               |
| Operations occur at a cursor           | Use a linked list.                          |
| Operations occur at a two-way cursor   | Use a doubly linked list.                   |
| Frequent capacity changes              | A linked list avoids resizing inefficiency. |
| Programming a linked list from scratch | Consider using a dummy head and tail.       |

### Self-Test Exercises for Section 4.6

38. What underlying data structure is quickest for random access?
39. What underlying data structure is quickest for additions/removals at a cursor?
40. What underlying data structure is best when a cursor must move both forward and backward?
41. What is the typical worst-case time analysis for changing the capacity of a collection class that is implemented with an array?
42. For the `IntTwoWayNode` declaration on page 240, implement a method to remove a node from its list. The node that activates the method is the node to be removed. Assume that there are public methods to get or set the forward or backward link of a node.

## CHAPTER SUMMARY

- A *linked list* consists of nodes; each *node* contains some data and a link to the next node in the list. The link part of the final node contains the null reference.
- Typically, a linked list is accessed through a *head reference* that refers to the *head node* (i.e., the first node). Sometimes a linked list is accessed elsewhere, such as through the *tail reference* that refers to the last node.
- You should be familiar with the methods of our node class, which provides fundamental operations to manipulate linked lists. These operations follow basic patterns that every programmer uses.
- Our linked lists can be used to implement ADTs. Such an ADT will have one or more private instance variables that are references to nodes in a linked list. The methods of the ADT will use the node methods to manipulate the linked list.
- You have seen two ADTs implemented with linked lists: a bag and a sequence. You will see more in the chapters that follow.
- ADTs often can be implemented in many different ways, such as with an array or with a linked list. In general, arrays are better at *random access*; linked lists are better at *additions/removals at a cursor*.
- A *doubly linked list* has nodes with two references: one to the next node and one to the previous node. Doubly linked lists are a good choice for supporting a cursor that moves forward and backward.
- Including dummy head and tail nodes on a linked list will simplify the programming for insertion and removal of nodes.
- The Java Class Libraries include sequence classes based on arrays (`ArrayList`) and based on linked lists (`LinkedList`).

**Solutions to Self-Test Exercises**

1. 

```
public class DoubleNode
{
    double data;
    DoubleNode link;
    ...
}
```
2. 

```
public class DINode
{
    double d_data;
    int i_data;
    DINode link;
}
```
3. The head node and the tail node.
4. The null reference is used for the link part of the final node of a linked list; it is also used for the head and tail references of a list that doesn't yet have any nodes.
5. No nodes. The head and tail are null.
6. A NullPointerException is thrown.
7. Using techniques from Section 4.2:  

```
if (head == null)
    head = new IntNode(42, null);
else
    head.addNodeAfter(42);
```
8. Using techniques from Section 4.2:  

```
if (head != null)
{
    if (head.getLink() == null)
        head = null;
    else
        head.removeNodeAfter();
}
```
9. They cannot be implemented as ordinary methods of the IntNode class, because they must change the head reference (making it refer to a new node).
10. 

```
IntNode head;
IntNode tail;
```

```
int i;
head = new IntNode(1, null);
tail = head;
for (i = 2; i <= 100; i++)
{
    tail.addNodeAfter(i);
    tail = tail.getLink();
}
```

11. There are eight different nodes for the eight primitive data types (boolean, int, long, byte, short, double, float, and char). These are called BooleanNode, IntNode, and so on. There is one more class simply called Node, which will be discussed in Chapter 5. The data type in the Node class is Java's Object type. So there are nine different nodes in all.

If you implement one of these nine node types, implementing another one takes little work: just change the type of the data and the type of any method parameters that refer to the data.

12. Within the IntNode class, you may write:  

```
locate = locate.link;
```

Elsewhere, you must write:  

```
locate = locate.getLink();
```

If locate is already referring to the last node before the assignment statement, then the assignment will set locate to null.
13. The new operator is used in the methods add-NodeAfter, listCopy, listCopyWithTail, and listPart.
14. It will be the null reference.
15. The listCopyWithTail method does exactly this by returning an array with two IntNode components.
16. 

```
douglass =
    IntNode.listSearch(head, 42);
adams =
    IntNode.listPosition(head, 42);
```

## 246 Chapter 4 / Linked Lists

```

17. IntNode cursor;
   for (cursor = head;
        cursor != null;
        cursor = cursor.link;
    )
        System.out.print(cursor.data);

18. public static int count42
    (IntNode head)
{
    int count = 0;
    IntNode cursor;
    for (cursor = head;
         cursor != null;
         cursor = cursor.link;
    )
    {
        if (cursor.data == 42)
            count++;
    }
    return count;
}

19. public static boolean has42
    (IntNode head)
{
    IntNode cursor;
    for (cursor = head;
         cursor != null;
         cursor = cursor.link;
    )
    {
        if (cursor.data == 42)
            return true;
    }
    return false;
}

20. public static int sum
    (IntNode head)
{
    int count = 0;
    IntNode cursor;
    for (cursor = head;
         cursor != null;
         cursor = cursor.link;
    )
    {
        count += cursor.data;
    }
    return count;
}

21. public static void tail42
    (IntNode head)
{
    IntNode cursor;
    if (head = null)
        head = new IntNode(42, null);
    else
    {
        // Move cursor to tail...
        cursor = head;
        while(cursor.link != null)
            cursor = cursor.link;
        // ...and add the new node:
        cursor.addNodeAfter(42);
    }
}

22. public static void remove42
    (IntNode head)
{
    // Remove first node after the head that
    // contains a data of 42 (if there is one).
    IntNode cursor;
    if (head = null)
    { // Empty list
        return;
    }

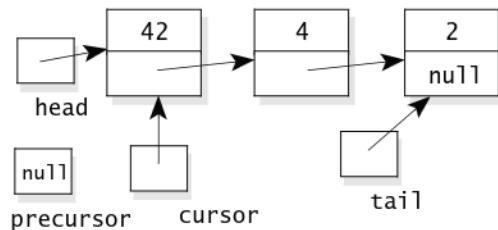
    // Step through each node of the list,
    // always looking to see whether the
    // next node has data of 42.
    cursor = head;
    while (cursor->link != null)
    {
        if (cursor->link->data == 42)
        { // The next node's data is 42,
          // so remove the next node
          // and return.
          cursor->link =
              cursor->link->link;
          return;
        }
    }
}

23. IntNode[ ] array;
array = IntNode.listPart(p, q);
x = array[0];
y = array[1];
array = IntNode.listPart(q.link, r);
y.link = array[0];
z = array[1];

24. These would change: add, addAll, remove,
and union. You could then implement a more
efficient countOccurrences that stopped
when it found a number bigger than the target
(but it would still have linear worst time).

```

25. Use DoubleNode instead of IntNode. There are a few other changes, such as changing some parameters from int to double.
26. We could write this code:
- ```
IntLinkedBag exercise =
    new IntLinkedBag();
exercise.add(42);
exercise.add(8);
System.out.println
    (exercise.grab());
System.out.println
    (exercise.size());
```
27. Generally, we will choose the approach that makes the best use of the node methods. This saves us work and also reduces the chance of introducing new errors from writing new code to do an old job. The preference would change if the other approach offered better efficiency.
28. The two lines of code we have in mind:
- ```
p = p.getLink();
p = listSearch(p, d);
```
- These two lines are the same as this line:
- ```
p = listSearch(p.getLink(), d);
```
29. When the target is not in the bag, the first assignment statement to cursor will set it to null. This means that the body of the loop will not execute at all, and the method returns the answer 0.
30. Test the case where you are removing the last element from the bag.
31. (int) (Math.random() \* 21) - 10
32. All the methods are constant time except for remove, grab, countOccurrences, and clone (all of which are linear); the addAll method (which is  $O(n)$ , where  $n$  is the size of the addend); and the union method (which is  $O(m+n)$ , where  $m$  and  $n$  are the sizes of the two bags).
33. The new bag and the old bag would then share the same linked list.
34. manyNodes is 3; the other instance variables are shown at the top of the next column.



35. First check that the element occurs somewhere in the sequence. If it doesn't, then return with no work. If the element is in the sequence, then set the current element to be equal to this element and activate the ordinary remove method.
36. The two add methods both allocate dynamic memory, as do addAll, clone, and concatenation.
37. The clone method should use listPart, as described on page 235.
38. Arrays are quickest for random access.
39. Linked lists are quickest for additions/removals at a cursor.
40. A doubly linked list
41. At least  $O(n)$ , where  $n$  is the size of the array prior to changing the size. If the new array is initialized, then there is also  $O(m)$  work, where  $m$  is the size of the new array.
42. 

```
public void removeTwoWay
{
    IntTwoWayNode before;
    IntTwoWayNode after;

    before = backlink;
    after forelink;

    if (before != null)
        before.forelink = after;

    if (after != null)
        after.backlink = before;

    forelink = backlink = null;
}
```



## PROGRAMMING PROJECTS

**1** For this project, you will use the bag of integers from Section 4.4. The bag includes the `grab` method from Figure 4.16 on page 229. Use this class in an applet that has three components:

1. A button
2. A small text field
3. A large text area

Each time the button is clicked, the applet should read an integer from the text field and put this integer in a bag. Then a random integer is grabbed from the bag, and a message is printed in the text area—something like “My favorite number is now ....”

**2** Write a new static method for the node class. The method has one parameter, which is a head node for a linked list of integers. The method computes a new linked list, which is the same as the original list but with all repetitions removed. The method’s return value is a head reference for the new list.

**3** Write a method with three parameters. The first parameter is a head reference for a linked list of integers, and the next two parameters are integers `x` and `y`. The method should write a line to `System.out` containing all integers in the list that are between the first occurrence of `x` and the first occurrence of `y`.

**4** Write a method with one parameter that is a head reference for a linked list of integers. The method creates a new list that has the same elements as the original list but in the reverse order. The method returns a head reference for the new list.

**5** Write a method that has two linked list head references as parameters. Assume that linked lists contain integer data, and on each list, every element is less than the next element on the same list. The method should create a new linked list that contains all the elements on both lists, and the new linked list should also be ordered (so that every element is less than the next element in the list). The new linked list should not eliminate duplicate elements (i.e., if the same element appears in both input lists, then two copies are placed in the newly constructed linked list). The method should return a head reference for the newly constructed linked list.

**6** Write a method that starts with a single linked list of integers and a special value called the splitting value. The elements of the list are in no particular order. The method divides the nodes into two linked lists: one containing all the nodes that contain an element less than the splitting value and one that contains all the other nodes. If the original linked list had any repeated integers (i.e., any two or more nodes with the same element in them), then the new linked list that has this element should have the same number of nodes that repeat this element. It does not matter whether you preserve the original linked list or destroy it in the process of building the two new lists, but your comments should document what happens to the original linked list. The method returns two head references—one for each of the linked lists that were created.

**7** Write a method that takes a linked list of integers and rearranges the nodes so that the integers stored are sorted into the order of smallest to largest, with the smallest integer in the node at the head of the list. Your method should preserve repetitions of integers. If the original list had any integers occurring more than once, then the changed list will have the same number of each integer. For concreteness you will use lists of



**250** Chapter 4 / Linked Lists

operations will be easier to implement (or more efficient) if you keep the nodes in order from smallest to largest exponent. Also, some operations will be more efficient if you include an extra instance variable that is a reference to the most recently accessed node. (Otherwise, you would need to start over at the head of the list every time.)

**14** Implement a node class in which each node contains a double number and each node is linked both forward and backward. Include methods that are similar to the node class from this chapter, but change the removal method so that the node that activates the method removes itself from the list.

**15** Revise the Statistician with median (Programming Project 15 on page 172) so that it stores the input numbers on a doubly linked list using the doubly linked node class from the previous project. Rather than a head reference, you should have an instance variable that keeps track of the node that contains the current median and two instance variables that tell how many numbers are before and after this median. Keep the numbers in order from smallest to largest. Whenever a new item is inserted, you should check to see whether it goes before or after the median. After the new item is inserted, you might have to move the median pointer forward or backward.

**16** Use a doubly linked list to implement the sequence class from Section 4.5. With a doubly linked list, there is no need to maintain a precursor. Your implementation should include a retreat member function that moves the cursor backward to the previous element. Also, use a dummy head and a dummy tail node. The data in these dummy nodes can be the Java constant Double.NaN, which is used for a double variable that is “not a number.”

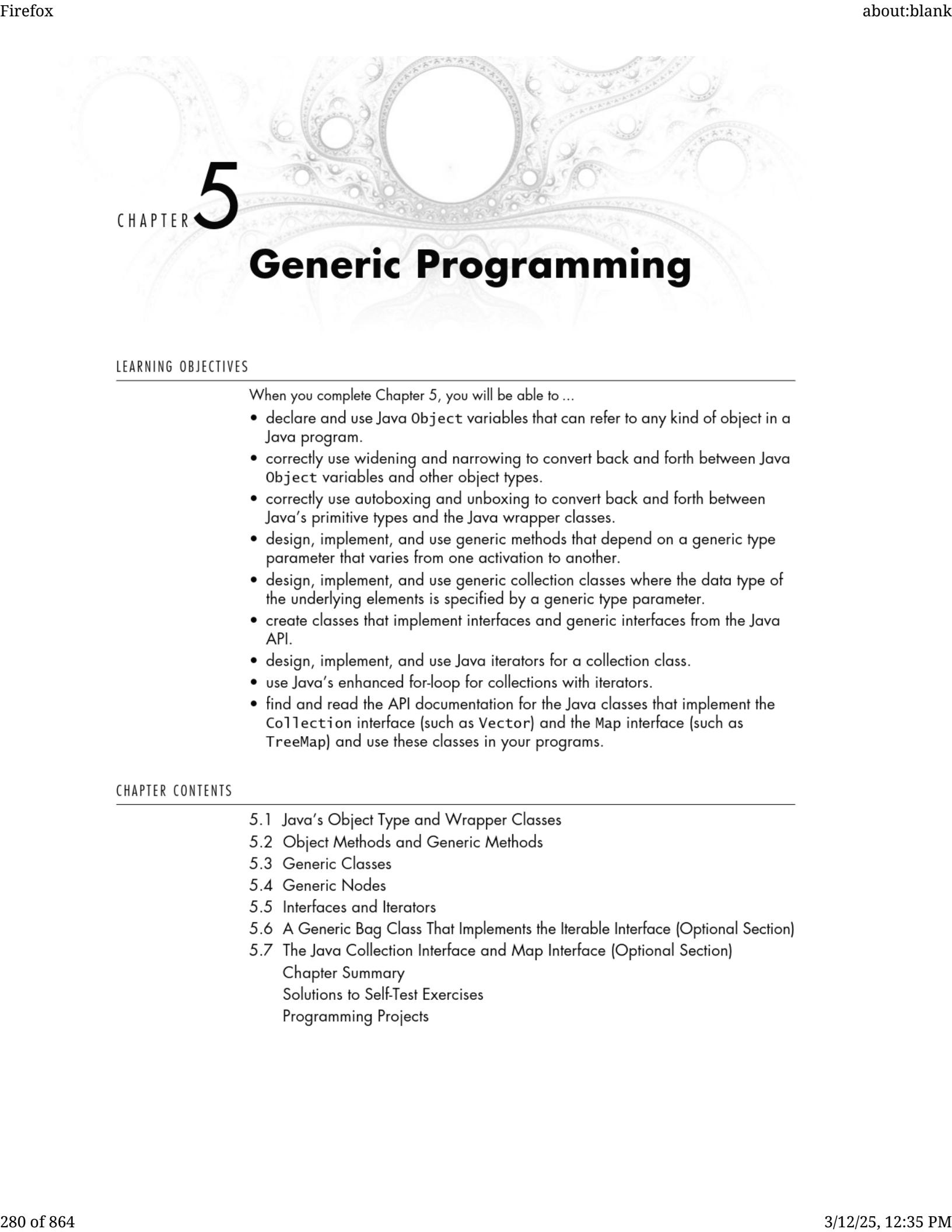
**17** In this project, you will implement a variation of the linked list called a circular linked list. The link field of the final node of a circular linked list is not NULL; instead, the link member of the tail pointer points back to the first node. In this project, an extra reference variable is used to refer to the beginning of the list; this variable will be NULL if the list is empty. Revise the third bag class developed in this chapter to use a circular linked-list implementation.

**18** Use a circular linked list to run a simple simulation of a card game called Crazy 8s. You can use any online rules that you find. The circular linked list is to hold the players (who sit in a circle while playing the game). Each player should be an object from a player class that you write yourself. You also design and use other relevant classes, such as a class for a single playing card and perhaps a class for a deck of cards.

**19** Reimplement the bag class from Figure 4.17 so that the items of the bag are stored with a new technique. Here’s the idea: Each node of the new linked list contains *two* integers. The first integer is called the *count*, and the second integer is called the *data*. As an example, if a node has a count of 6 and data of 10, then this means that the bag has six copies of the number 10.

The nodes of the linked list should be kept in order from the smallest data (at the head of the list) to the largest (at the tail of the list). You should never have two different nodes with the same data, and if the count in a node drops to zero (meaning there are no copies of that node’s data), then the node should be removed from the linked list.

The public member functions of your new class should be identical to those in Figure 4.17.



# CHAPTER 5

# Generic Programming

---

## LEARNING OBJECTIVES

When you complete Chapter 5, you will be able to ...

- declare and use Java Object variables that can refer to any kind of object in a Java program.
- correctly use widening and narrowing to convert back and forth between Java Object variables and other object types.
- correctly use autoboxing and unboxing to convert back and forth between Java's primitive types and the Java wrapper classes.
- design, implement, and use generic methods that depend on a generic type parameter that varies from one activation to another.
- design, implement, and use generic collection classes where the data type of the underlying elements is specified by a generic type parameter.
- create classes that implement interfaces and generic interfaces from the Java API.
- design, implement, and use Java iterators for a collection class.
- use Java's enhanced for-loop for collections with iterators.
- find and read the API documentation for the Java classes that implement the Collection interface (such as Vector) and the Map interface (such as TreeMap) and use these classes in your programs.

---

## CHAPTER CONTENTS

- 5.1 Java's Object Type and Wrapper Classes
- 5.2 Object Methods and Generic Methods
- 5.3 Generic Classes
- 5.4 Generic Nodes
- 5.5 Interfaces and Iterators
- 5.6 A Generic Bag Class That Implements the Iterable Interface (Optional Section)
- 5.7 The Java Collection Interface and Map Interface (Optional Section)
  - Chapter Summary
  - Solutions to Self-Test Exercises
  - Programming Projects

# CHAPTER 5

# Generic Programming

*I never knew just what it was, and I guess I never will.*

TOM PAXTON  
“The Marvelous Toy”

generic classes

**P**rofessional programmers often write classes that have general applicability in many settings. To some extent, our classes do this already. Certainly the bag and sequence classes can be used in many different settings. However, both our bag and our sequence suffer from the fact that they require the underlying element type to be fixed. We started with a bag of integers (`IntArrayBag`). If we later need a bag of double numbers, then a second implementation (`DoubleArrayList`) is required. We can end up with eight different bags—one for each of Java’s eight primitive types. But even then we’re not done; we could use a bag of locations (with the `Location` class in Figure 2.5 on page 67) or a bag of strings (using Java’s built-in `String` class, which is not one of the primitive types). It seems as if the onslaught of bags will never stop.

A **generic class** is a new feature of Java that provides a solution to this problem. For example, a single generic bag can be used for a bag of integers, but it can also be used for a bag of double numbers, or strings, or whatever else is needed. By using Java’s `Object` type, a generic bag can even hold a mixture of objects of different types.

This chapter shows how to build your own generic classes and how to use some of the generic collection classes that Java provides. In addition, you’ll learn about related issues, such as the effective use of Java’s `Object` type and wrapper classes, which are addressed in the first section of the chapter.

## Eight Primitive Types

byte  
short

int

long

float

double

char

boolean

*...and everything else is a reference to an object*

## 5.1 JAVA’S OBJECT TYPE AND WRAPPER CLASSES

A Java variable can be one of the eight primitive data types. Anything that’s not one of the eight primitive types is a reference to an object. For example, if you declare a `Location` variable, that variable is capable of holding a reference to a `Location` object; if you declare a `String` variable, that variable is capable of holding a reference to a `String` object.

Java has an important built-in data type called `Object`. An `Object` variable is capable of holding a reference to any kind of object. To see how this is useful, let’s look at how assignment statements can go back and forth between an `Object` variable and other data types.

## Widening Conversions

Here's some code to show how an assignment can be made to an `Object` variable. The code declares a `String` with the value "Objection overruled!". A second `Object` variable is then declared and made to refer to the same string.

```
String s = new String("Objection overruled!");
Object obj;

obj = s;
```

At this point, there is only one string—"Objection overruled!"—with both `s` and `obj` referring to this one string, as shown here:



Assigning a specific kind of object to an `Object` variable is an example of a **widening conversion**. The term "widening" means that the `obj` variable has a "wide" ability to refer to different things; in fact, `obj` is very "wide" because it *could* refer to any kind of object. On the other hand, the variable `s` is relatively narrow with regard to the things that it can refer to—`s` can refer only to a `String`.

### Widening Conversions with Reference Variables

Suppose `x` and `y` are reference variables. An assignment `x = y` is a **widening conversion** if the data type of `x` is capable of referring to a wider variety of things than the type of `y`.

Example:

```
String s = new String("Objection overruled!");
Object obj;
obj = s;
```

Java permits all widening conversions.

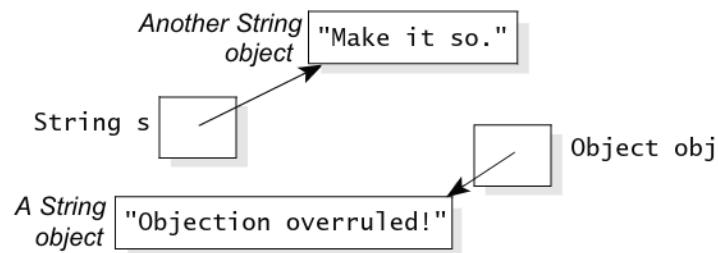
### Narrowing Conversions

After the widening conversion `obj = s`, our program can continue to do other things, perhaps even making `s` refer to a new string, as shown here:

```
String s = new String("Objection overruled!");
Object obj;

obj = s;
s = new String("Make it so.');
```

At this point, `s` refers to a new string, and `obj` still refers to the original string, as shown here:



At a later point in the program, we can make `s` refer to the original string once again with an assignment statement, but the assignment needs more than merely `s = obj`. In fact, the Java compiler forbids the assignment `s = obj`. As far as the compiler knows, `obj` could refer to anything; it does not have to refer to a `String`, so the compiler forbids `s = obj`. The way around the restriction is to use the typecast expression shown here:

```
s = (String) obj;
```

The expression `(String) obj` tells the compiler that the reference variable `obj` is really referring to a `String` object. This is a typecast, as discussed in Chapter 2. With the typecast, the compiler allows the assignment statement, though you must still be certain that `obj` actually does refer to a `String` object. Otherwise, when the program runs, there will be a `ClassCastException` (see “Pitfall: `ClassCastException`” on page 82).

The complete assignment `s = (String) obj` is an example of a **narrowing conversion**. The term “narrowing” means that the left side of the assignment has a smaller ability to refer to different things than the right side.

### Narrowing Conversion with Reference Variables

Suppose `x` and `y` are reference variables, and `x` has a smaller ability to refer to things than `y`. A narrowing conversion using a typecast can be made to assign `x` from `y`.

Example:

```
String s = new String("Objection overruled!");
Object obj;
obj = s;
...
s = (String) obj;
```

By using a typecast, Java permits all narrowing conversions, though a `ClassCastException` may be thrown at runtime if the object does not satisfy the typecast.

Narrowing conversions also occur when a method returns an `Object` and the program assigns that `Object` to a variable of a particular type. For example, the `IntLinkedBag` class from Chapter 4 has a `clone` method:

#### ◆ clone

```
public IntLinkedBag clone()
```

Generate a copy of this bag.

#### Returns:

The return value is a copy of this bag. Subsequent changes to the copy will not affect the original, nor vice versa. The return value must be typecast to an `IntLinkedBag` before it is used.

In the implementation of this method, we have the assignment:

```
IntLinkedBag answer;
answer = (IntLinkedBag) super.clone();
```

The `super.clone` method from Java's `Object` class returns an `Object`. Of course, in this situation, we know that the `Object` is really an `IntLinkedBag`, but the compiler doesn't know that. Therefore, to use the `answer` from the `super.clone` method, we must insert the narrowing typecast, `(IntLinkedBag)`. This tells the compiler that the programmer knows that the actual value is an `IntLinkedBag`, so it is safe to assign it to the `answer` variable.

### Methods That Returns an Object

If a method returns an `Object`, then a narrowing conversion is usually needed to actually use the return value.

## Wrapper Classes

For a programmer, it's annoying that *almost* everything is a Java object. It's those eight darn primitive types that cause the problem. Java's wrapper classes offer a partial solution to this annoyance. A **wrapper class** is a class in which each object holds a primitive value. For example, a wrapper class called `Integer` is designed so that each `Integer` object holds one `int` value. The two most important `Integer` methods are the constructor (which creates an `Integer` object from an `int` value) and the `intValue` method (which accesses the `int` value stored in an `Integer` object). Here are the specifications for these two `Integer` methods:

Primitive	Wrapper Class
<code>byte</code>	<code>Byte</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>
<code>char</code>	<code>Character</code>

### ◆ Constructor for the Integer class

```
public Integer(int value)
Construct an Integer object from an int value.
```

#### Parameter:

`value` – the `int` value for this `Integer` to hold

#### Postcondition:

This `Integer` has been initialized with the specified `int` value.

### ◆ intValue

```
public int intValue()
```

Accessor method to retrieve the `int` value of this object.

#### Returns:

the `int` value of this object

Here's an example to show how an integer is placed into an `Integer` object (called a **boxing conversion**) and taken back out (an **unboxing conversion**):

```
int i = 42;
int j;
Integer example;
example = new Integer(i); // The example has value 42 (boxing)
j = example.intValue(); // j is now 42 too (unboxing)
```

## Autoboxing and Auto-Unboxing Conversions

automatically convert from a primitive value to a wrapper object, or vice versa

Java has a convenience that simplifies the process of putting a primitive value into a wrapper object and taking it back out. In many situations, the Java compiler will recognize that a wrapper object is needed and automatically convert a primitive value to the wrapper object (an **autoboxing conversion**). Java will also convert from a wrapper object to the corresponding primitive type value in many situations (an **auto-unboxing conversion**). For example, the assignments shown above can be simplified:

```
example = i;      // Autoboxing occurs in an assignment statement
j = example;     // Auto-unboxing occurs in an assignment statement
```

In a similar way, if a method expects an `Integer` parameter, the actual argument may be a primitive `int` value, and an autoboxing conversion will occur. If a method expects an `int` parameter, the actual argument may be an `Integer` object, and an auto-unboxing conversion will occur.

### Boxing and Unboxing Conversions

A **boxing conversion** occurs when a primitive value is converted to a wrapper object of the corresponding type. An **unboxing conversion** occurs when a wrapper object is converted to the corresponding primitive type. These may occur with an explicit conversion or automatically when the compiler detects the need for the conversion.

Examples:

```
int i = 42;
int j;
Integer example;
example = new Integer(i); // Boxing
example = i;             // Autoboxing
j = example.intValue();  // Unboxing
j = example;              // Auto-unboxing
```

### Advantages and Disadvantages of Wrapper Objects

The advantage of putting a value in a wrapper object is that the wrapper object is a full-blown Java object. For example, it can be put into the generic bag that we will create in Section 5.3. The disadvantage is that the ordinary primitive operations are no longer directly available. For example, suppose `x`, `y`, and `z` are `Integer` objects. Although the statement `z = x + y;` is legal, its evaluation is somewhat slow because the `x` and `y` `Integer` objects must first be auto-unboxed (converted to primitive `int` values) before the “`+`” operation can be applied. The answer must then undergo an autoboxing conversion before it can be stored back in the `Integer z`.

### Self-Test Exercises for Section 5.1

1. Can an `Object` variable refer to a `String`? To a `Location`? To an `IntArrayBag`?
2. Write some code that includes both a widening conversion and a narrowing conversion. Draw a circle by the widening and a triangle by the narrowing.

**258 Chapter 5 / Generic Programming**

3. Write an example of a narrowing conversion that will cause a `ClassCastException` at runtime.
4. Write some code that creates a `Character` wrapper object called `example`, initializing with the char 'w'.
5. Describe an advantage and a disadvantage of wrapper objects.
6. Suppose that `x`, `y` and `z` are all `Double` objects. Describe all the boxing and unboxing that occurs in the assignment `z = x + y;`.

## 5.2 OBJECT METHODS AND GENERIC METHODS

Sometimes it seems that programmers intentionally make extra work for themselves. For example, suppose we write this method:

```
public static Integer middle(Integer[] data)
// If the array is non-empty, then the function returns an element from
// near the middle of the array; otherwise, the return value is null.
{
    if (data.length == 0)
        { // The array has no elements, so return null.
            return null;
        }
    else
        { // Return an element near the center of the array.
            return data[data.length/2];
        }
}
```

This is a fine method, reliably returning an element from near the middle of any non-empty array. Such a method is useful in certain kinds of searches. For example (with an `Integer` `i` and a non-empty `Integer` array `ia`):

```
i = middle(ia); // Set i to something from the middle of the array ia.
```

Now, suppose that tomorrow you have another program that needs to do the same thing with `Character` objects:

```
public static Integer middle(Integer[] data)...
```

*we write lots of  
different middle  
methods*

As you start writing the code, you realize that the only difference between this new method and the original version is that we must change every occurrence of the word `Integer` to `Character`. Later, if we want to carry out the same task with values of some other type, we might write another method, and another .... In fact, a single program might use many `middle` methods. When one of the functions is used, the compiler looks at the type of the argument and selects the appropriate version of the `middle` method. This works fine, but with this approach, you do need to write a different method for each different data type.

## Object Methods

One way to avoid many different `middle` methods is to write just one method using the `Object` data type:

```
public static Object middle(Object [] data)
{
    if (data.length == 0)
    { // The array has no elements, so return null.
        return null;
    }
    else
    { // Return an element near the center of the array.
        return data[data.length/2];
    }
}
```

This approach also works fine. We can use this version of the `middle` method with an `Integer` array and a narrowing conversion to convert the return value from an `Object` to an `Integer`:

```
i = (Integer) middle(ia); // i is an Integer; ia is an Integer array.
```

We need only one version of the `Object` method, but there are some potential type errors that can't be caught until the program is running. For example, a programmer might accidentally use the `middle` method with a `Character` array, but assign the result to an `Integer` variable. Such a mistake would compile, but at run time there would be a `ClassCastException` from the illegal narrowing conversion. Another approach, called generic methods, offers a safer approach that cannot result in a type mismatch.

## Generic Methods

A **generic method** is a method that is written with the types of the parameters not fully specified. Our `middle` method is written this way as a generic method:

```
public static <T> T middle(T[] data)
{
    if (data.length == 0)
    { // The array has no elements, so return null.
        return null;
    }
    else
    { // Return an element near the center of the array.
        return data[data.length/2];
    }
}
```

**260 Chapter 5 / Generic Programming**

Notice that the name T appears in angle brackets ( $<T>$ ) right before the return type in the method header. This name is called the **generic type parameter**. The generic type parameter indicates that the method depends on some particular class, but the programmer is not going to specify exactly what that class is. Perhaps T will eventually be an Integer, or a Character, or who knows what else. The name, T, that we used for the generic type parameter may be any legal Java identifier, but the designers of Java recommend single capital letters such as T (an abbreviation of “type”) or E (an abbreviation of “element”).

The generic type parameter always appears in angle brackets right before the return type of the method. This is how the compiler knows that the method is a generic method. In the rest of the method’s implementation, the generic type may be used just like any other class name, and for most situations, it must be used at least once in the parameter list. In our example, T is used twice (once as the return type of the method and once as the data type of the array in the parameter list):

```
public static <T> T middle(T[] data)...
```

When a generic method is activated, the compiler infers the correct class for the generic type parameter. For example:

```
i = middle(ia); // i is an Integer; ia is an Integer array.
```

a generic method allows any class for the argument, and the compiler can detect certain type errors

In this case, the compiler sees the Integer array, ia, and determines that the data type of T must be Integer. This allows the compiler to detect potential type errors at compile time, before the program is running. For example:

```
c = middle(ia); // c is a Character; ia is an Integer array-type error!
```

For this assignment statement, the compiler will indicate that the Integer return value cannot be assigned to a Character variable.

Later, we will see generic methods with more than one generic type parameter. For example, here is a method with two generic type parameters, each of which occurs twice in the parameter list:

```
public static <S,T> bool most(S[] sa, S st, T[] ta, T tt)
// Returns true if st is in sa more often than tt is in ta.
```

**PITFALL****GENERIC METHOD RESTRICTIONS**

erasure

The data type inferred by the compiler for the generic type must always be a class type (not one of the primitive types). In addition, you may not call a constructor for the generic type, nor may you create a new array of elements of that type.

These restrictions are a consequence of a compilation technique called **erasure**, in which the exact data type of a generic type is unknown at run time when a generic method is running.

### Generic Methods

A **generic method** is written by putting a generic type parameter in angle brackets before the return type in the heading of the method. For example:

```
static <T> T middle(T[] data)...
```

The generic data type is often named with a single capital letter, such as T for “type.” The name T can then be used in the rest of the method implementation as if it were a real class name (with a few exceptions).

A programmer can activate a generic method just like any other method. The compiler will look at the data types of the arguments and infer a correct type for each generic type parameter. The inferred types help catch type errors at compile time. For example:

```
// i is an integer; ia is an Integer array; c is a Character  
i = middle(ia); // This is fine  
c = middle(ia); // Compile-time type error
```

Some additional features of generic methods, such as the ability to restrict the type of the argument to specific kinds of objects, will be covered in Chapter 13.

### Self-Test Exercises for Section 5.2

7. Describe the main purpose of a generic method.
8. What is the principal disadvantage of using an object method as compared to a generic method?
9. What is the generic type parameter, and where does it first appear in the method implementation? Where else may it appear in the implementation? In what one other location does it nearly always appear?
10. Name two things that can usually be done with a class type but which are forbidden for a generic type.
11. Write a generic method for counting the number of occurrences of a target in an array. If the target is non-null, then use `target.equals` to determine equality.
12. Write the `most` method that was specified at the end of this section.



We can then activate member functions such as `sbag.add("Thunder")` or `ibag.add(42)`. This `ibag` activation will autobox the 42, so that the argument is an `Integer` object rather than a primitive `int`. However, the compiler will generate a typechecking error for a statement such as `ibag.add("Thunder")`. You cannot add a `String` object to a bag of `Integer` objects.

## PITFALL

### GENERIC CLASS RESTRICTIONS

In a generic class, the type used to instantiate the generic type parameter must be a class (not a primitive type). In addition, within the implementation of the generic class, you may not call a constructor for the generic type, nor may you create a new array of elements of that type.

These restrictions are identical to the restrictions for a generic method (page 260), and are a consequence of the compilation method that Java uses for generic classes.

### Details for Implementing a Generic Class

Figure 5.2 on page 271 gives the complete implementation of the generic `ArrayList` class. The implementation is mostly unsurprising: We took the original `IntArrayList` implementation, changed the class name to `ArrayList<E>`, and throughout the implementation we use the unknown type `E` as the type of element in the bag (rather than `int`). Still, there are a few issues that you need to examine, and we'll discuss those next.

### Creating an Array to Hold Elements of the Unknown Type

Within the generic class implementation, we are not allowed to create a new array of objects of the unknown type `E`. This causes a problem for our bag because we need an array that holds references to objects of type `E`. The solution is to declare `data` as an array of Java `Objects` (so the class has the private instance variable `private Object[ ] data;`), and we initialize it this way in the constructor:

```
public ArrayList( )
{
    final int INITIAL_CAPACITY = 10;
    manyItems = 0;
    data = new Object[INITIAL_CAPACITY];
}
```

In the rest of the `ArrayList` code, we will ensure that we put only objects of type `E` into the array. (An alternative to this approach is briefly discussed in Figure 5.1 on the next page.)

**FIGURE 5.1 An Older Approach to Implementing Generic Collection Classes**

In our `ArrayBag<E>`, we use an array of objects, `private Object[ ] data;`, and our programming ensures that we put only `E` objects into the array. An alternative that has been used in older code uses an array of `E` objects instead: `private E[ ] data;`. In the constructor, the array is still created as an array of ordinary objects, and a typecast allows the `data` variable to refer to it, in this way:

```
public ArrayBag( )
{
    final int INITIAL_CAPACITY = 10;
    manyItems = 0;
    data = (E[ ]) new Object[INITIAL_CAPACITY];
}
```

This alternative was used in earlier editions of this text and also in the original implementations of the Java generic collection classes. But the alternative is discouraged because it has a technical, uncaught type error at run time when the `(E[ ])` variable is set to refer to an `(Object[ ])` array. The latest version of the Java generic collection classes use `private Object[ ] data;` as we do in the text.

### Retrieving E Objects from the Array

We know from our programming that the objects in the data array will always be `E` objects, but the Java compiler does not know this. Therefore, whenever we retrieve a component of the array, we must apply a typecast to the type `E` (as previously discussed on page 79). For example, the needed typecast is highlighted in this one statement of the `ArrayBag`'s `grab` method:

```
return (E) data[i]; // From the end of the grab method
```

This typecast `(E)` is one of several places where Java compilers issue warnings about possible errors that can occur at run time. We discuss these situations next.

### Warnings in Generic Code

Our `ArrayBag` implementation from Figure 5.2 on page 271 has several places where Java compilers will generate warnings. It's okay to have these warnings in your code, but you must be aware of the situations that cause the possible errors, so you can avoid these situations in your programming.

**Typecasts.** In our `grab` method, we take the Java object, `data[i]`, and typecast it to an `E` object. This is safe because it was our own programming (such as the `add` method) that put the object into `data[i]`, and at that point, we knew it was an `E` object.

But during run time, the information about the actual data type of a generic object is always unavailable. Therefore, there is no way for the program to check (while it is running) whether the typecast is correct. The warning that Java produces in this situation is an “unchecked cast” warning. This means that if the typecast is illegal at run time, then the error will not be caught as it usually is.

As a programmer, it is your responsibility to notice the warning and ensure that the value (`data[i]` in this case) actually is the correct data type (`E` in this case).

A second typecast that causes a similar problem is in the `ArrayBag`'s `clone` method at this statement:

```
answer = (ArrayBag<E>) super.clone(); // From the clone method
```

In this case, the `super.clone()` method returns a generic object that has had its type information erased at run time. Even though its type cannot be checked at run time, we know that this object is an `ArrayBag<E>` object, so the assignment is safe, despite the warning.

**Variable Arity Methods.** The `addMany` method of the `ArrayBag<E>` class is a variable arity method, meaning that it has a variable number of arguments that are put into an array for the `addMany` method to access (see page 115). This results in the compiler creating an array of generic objects, which usually is forbidden because it can result in unchecked typecast errors that are similar to those we have already seen.

Java compilers can issue warnings at two locations for this combination of variable arity and generic types. One location (starting with Java 1.7) is within the generic class at the declaration of the variable arity method. A second location (which occurs in only certain situations) is at the point where a program activates the variable arity method.

Provided that you access the array only as an array of `E` objects, then your programming is safe, and the warnings are not relevant.

## PROGRAMMING TIP

### SUPPRESSING UNCHECKED WARNINGS

If you write classes so that the possible runtime errors cannot occur, then you may document this and suppress the warnings with this annotation immediately before any method that contains a warning: `@SuppressWarnings("unchecked")`. Notice that there is no semicolon after this annotation; the line can be placed by itself just prior to any method declaration.

Starting with Java 1.7, there is a different annotation that can be used before a variable arity method: `@SafeVarargs`. This version of the annotation will suppress both the warning at the declaration of the variable arity method and the warnings that sometimes occur when the variable arity method is activated, but its use is restricted to only static or final methods.

### Using ArrayBag as the Type of a Parameter or Return Value

The `ArrayBag` has several methods that have bags as parameters or as the return type. For example, our `IntArrayBag` has an `addAll` method:

```
public void addAll(IntArrayBag addend)...
```

When a program activates `b1.addAll(b2)`, all the integers from `b2` are put in the bag `b1`.

For the generic bag, the way to implement such a parameter is to specify its data type as `ArrayBag<E>` (rather than just `ArrayBag`):

```
public void addAll(ArrayBag<E> addend)...
```

This will allow us to activate `b1.addAll(b2)` for two bags, `b1` and `b2`, but only if the type of the elements in `b1` is the same as the type of elements in `b2`.

For our array bag, all the uses of the `ArrayBag` data type within its own implementation will be written as `ArrayBag<E>`.

### Counting the Occurrences of an Object

Counting the occurrences of an object in a bag requires some thought. Here's an example to get you thinking. Suppose you construct a bag of `Location` objects and create two identical `Location` objects with coordinates of  $x=2$ ,  $y=4$  (using the `Location` class from Section 2.4). The locations are then added to the bag:

```
ArrayBag<Location> spots = new ArrayBag<Location>();
Location p1 = new Location(2, 4); // x=2 and y=4
Location p2 = new Location(2, 4); // Also at x=2 and y=4

spots.add(p1);
spots.add(p2);
```

The two locations are identical but separate objects, as shown in this drawing:



Keep in mind that the boolean expression `(p1 == p2)` is `false` because “`==`” returns `true` only if `p1` and `p2` refer to the exact same object (as opposed to two separate objects that happen to contain identical values). On the other hand, the `Location` class has an `equals` method with this specification:

**◆ equals (from the Location class)**

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

Compare this Location to another object for equality.

**Parameter:**

obj – an object with which this Location is compared

**Returns:**

A return value of true indicates that obj refers to a Location object with the same value as this Location. Otherwise, the return value is false.

Both p1.equals(p2) and p2.equals(p1) are true.

So here's the question to get you thinking: With both locations in the bag, what is the value of spots.countOccurrences(p1)? In other words, how many times does the target p1 appear in the bag? The answer depends on exactly how we implement countOccurrences, with these two possibilities:

1. countOccurrences could step through the elements of the bag, using the “==” operator to look for the target. In this case, we find one occurrence of the target, p1, and spots.countOccurrences(p1) is 1.
2. countOccurrences could step through the elements of the bag, using equals to look for the target. In this case, both locations are equal to the target, and spots.countOccurrences(p1) is 2.

Every class has an equals method. For example, the equals method of the String class returns true when the two strings have the same sequence of characters. The equals method of the Integer wrapper class returns true when the two Integer objects hold the same int value. Because the equals method is always available, we'll use the second approach toward counting occurrences—and spots.countOccurrences(p1) is 2. Our bag documentation will make it clear that we count the occurrences of a non-null element by using its equals method.

**Generic Collections Should Use the equals Method**

When a generic collection tests for the presence of a non-null element, you should generally use the equals method rather than the “==” operator.

## The Collection Is Really a Collection of References to Objects

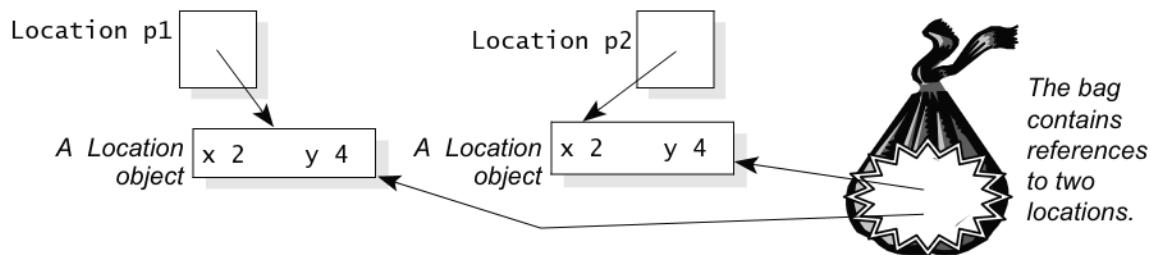
There is another aspect of generic collections that you must understand. Although we use phrases such as “bag of objects,” what we really have is a collection of *references* to objects. In other words, the bag does not contain separate copies of each object. Instead, the bag contains only references to each object that is added to the bag.

## 268 Chapter 5 / Generic Programming

Let's draw some pictures to see what this means. Once again, we'll create two identical locations and put them in a bag:

```
ArrayBag<Location> spots = new ArrayBag<Location>();
Location p1 = new Location(2, 4); // x=2 and y=4
Location p2 = new Location(2, 4); // Another at x=2 and y=4
spots.add(p1);
spots.add(p2);
```

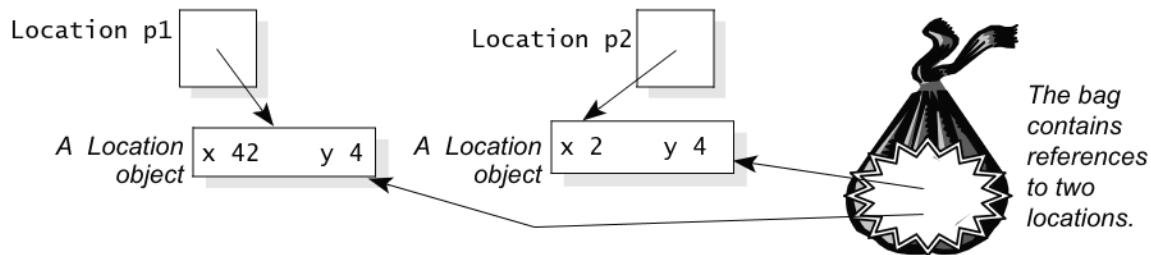
Here's a picture of the two locations and the bag after the five statements:



The references to the two locations are now in the bag. As we have already discussed, `spots.countOccurrences(p1)` is 2 because there are two locations with the coordinates  $x = 2$ ,  $y = 4$ . What happens if we change `p1` or `p2`? For example, after putting the two locations in the bag, we could execute this statement:

```
p1.shift(40, 0);
```

This shifts `p1` by 40 units along the `x`-axis and leaves `y` alone, so `p1` is now at  $x = 42$ ,  $y = 4$ , as shown here:



The bag still has two locations, but one of those locations has shifted to  $x = 42$ ,  $y = 4$ . Therefore, `spots.countOccurrences(p1)` is now just 1 (there is one location in the bag with the coordinates of `p1`).

## Set Unused References to Null

Our bag uses a partially filled array. For example, a bag might contain 14 elements, but the array could be much larger than 14. When the bag contained primitive data, such as integers, we didn't really care what values were stored in the unused part of the array. If a bag stored 14 integers, then we didn't care what was beyond the first 14 locations of the array.

However, our `ArrayBag` no longer uses primitive types. Instead, we have a partially filled array of references to objects. To help Java collect unused memory, we should write our methods so that the unused part of the array contains only null references. For our `ArrayBag` class, this means that whenever we remove an element, we will assign `null` to the array location we are no longer using.

### Set Unused Reference Variables to Null

When a collection class is converted to a generic collection class, make sure any unused reference variables are set to `null`. This usually occurs in methods that remove elements from the collection. Setting these variables to `null` will allow Java to collect any unused memory.

## Steps for Converting a Collection Class to a Generic Class

Here's a summary of exactly how we changed the `IntArrayBag` (from Section 3.2) into the generic `ArrayBag` class (in Figure 5.2 on page 271).

**1. The Name of the Class.** Each occurrence of the old class name, `IntArrayBag`, is changed to the new name, `ArrayBag<E>`. The name `ArrayBag` indicates that we have an array implementation of a bag, but we do not have any single underlying data type (such as `int`) because the new bag is a generic bag that holds an unspecified kind of element.

Notice that the name of the constructors is just `ArrayBag` (without the `<E>`).

**2. The Type of the Data Array.** Change the type of the data array to the declaration `private Object[] data;`. Make sure that all the arrays that you create are `Object` arrays (such as in the constructor).

**3. The Type of the Underlying Element.** Find all the remaining spots where the old class used `int` to refer to an element in the bag. Change these spots to the generic type parameter `E`. For example, the old bag had a method with this heading to remove an element: `public boolean remove(int target)`. The new bag has `public boolean remove(E target)`. Be careful because some `int` values do not refer to elements in the bag, and those must stay `int` (such as the `manyItems` instance variable).

**4. Change Static Methods to Generic Static Methods.** The class may have some static methods. For example, the `IntArrayBag` had this static method:

```
public static IntArrayBag union  
(IntArrayBag b1, IntArrayBag b2)...
```

Any static method that depends on the generic type `E` must be changed to a generic method, which means that `<E>` appears just before the return type in the method's heading, like this for our `ArrayBag`:

```
public static <E> ArrayBag<E> union  
(ArrayBag<E> b1, ArrayBag<E> b2)...
```

**5. Use Typecasts When an Element Is Retrieved from the Array.** Our class has one location in the `grab` method where an item is retrieved from the array and returned from the method. We know that the retrieved item is actually an `E` object, so we may insert a typecast at this point in the code:

```
return (E) data[i];
```

**6. Suppress Warnings.** As discussed earlier, some code in generic collection classes generates “unchecked” warnings. Examine each of these spots, as discussed on page 264, and suppress the warnings (which is safe since our programming ensures that the data array contains only objects of type `E`).

**7. Update Equality Tests.** Find all the spots where the old class used “`==`” or “`!=`” to compare two elements. Change these spots so that they use the `equals` method instead. For example, instead of `target != data[index]`, we will write `!target.equals(data[index])`.

**8. Decide How to Treat the Null Reference.** The new bag stores references to objects. You must decide whether to allow the null reference to be placed in the bag. For our bag, we'll allow the null reference to be placed in the bag, and we'll indicate this in the documentation. Some of the bag's methods will need special cases to deal with null references. For example, `countOccurrences` will search for a null target using “`==`” to count the number of times that `null` is in the bag, but a non-null target is counted by using its `equals` method.

**9. Set Unused Reference Variables to Null.** For our bag, this occurs in the `remove` method. Each time we remove an element, there is one array location that is no longer being used, and we set that array location to `null` (rather than letting it continue to refer to an object).

**10. Update All Documentation.** All documentation must be updated to show that the bag is a collection of references to objects.

Any collection class can be converted to a collection of objects by following these steps. For example, Programming Project 1 on page 313 asks you to convert the `Sequence` class from Section 3.3.

As for the new `ArrayBag` class, its specification and implementation are given in Figure 5.2. The changes are marked in the figure, and you'll also find one other improvement: We have included a `grab` method to allow a program to grab a randomly selected object from the bag. The implementation of the new `grab` method is similar to the `grab` method in the linked list version of the bag (see Section 4.4).

### Deep Clones for Collection Classes

The `clone` method creates a copy of an `ArrayBag`. As with other `clone` methods, adding or removing elements from the original will not affect the copy, nor vice versa. However, these elements are now references to objects; both the original and the copy contain references to the same underlying objects. Changing these underlying objects will affect both the original and the copy. An alternative cloning method, called **deep cloning**, can avoid the problem, as discussed in Programming Project 5 on page 313.

---

**FIGURE 5.2** Specification and Implementation for the `ArrayBag`

#### Generic Class `ArrayBag`

❖ **public class `ArrayBag<E>` from the package `edu.colorado.collections`**

An `ArrayBag<E>` is a collection of references to `E` objects.

**Limitations:**

- (1) The capacity of one of these bags can change after it's created, but the maximum capacity is limited by the amount of free memory on the machine. The constructors, `add`, `clone`, and `union` will result in an `OutOfMemoryError` when free memory is exhausted.
- (2) A bag's capacity cannot exceed the largest integer, 2,147,483,647 (`Integer.MAX_VALUE`). Any attempt to create a larger capacity results in failure due to an arithmetic overflow.
- (3) Because of the slow linear algorithms of this class, large bags will have poor performance.

#### Specification

❖ **Constructor for the `ArrayBag<E>`**

`public ArrayBag()`

Initialize an empty bag with an initial capacity of 10. Note that the `add` method works efficiently (without needing more memory) until this capacity is reached.

**Postcondition:**

This bag is empty and has an initial capacity of 10.

**Throws: `OutOfMemoryError`**

Indicates insufficient memory for `new Object[10]`.

(continued)

**272 Chapter 5 / Generic Programming**

(FIGURE 5.2 continued)

**◆ Second Constructor for the ArrayBag<E>**

```
public ArrayBag(int initialCapacity)
```

Initialize an empty bag with a specified initial capacity. Note that the add method works efficiently (without needing more memory) until this capacity is reached.

**Parameter:**

initialCapacity – the initial capacity of this bag

**Precondition:**

initialCapacity is non-negative.

**Postcondition:**

This bag is empty and has the given initial capacity.

**Throws: IllegalArgumentException**

Indicates that initialCapacity is negative.

**Throws: OutOfMemoryError**

Indicates insufficient memory for new Object[initialCapacity].

**◆ add**

```
public void add(E element)
```

Add a new element to this bag. If this new element would take this bag beyond its current capacity, then the capacity is increased before adding the new element.

**Parameter:**

element – the new element that is being added

**Postcondition:**

A new copy of the element has been added to this bag.

**Throws: OutOfMemoryError**

Indicates insufficient memory for increasing the capacity.

**Note:**

Creating a bag with capacity beyond Integer.MAX\_VALUE causes arithmetic overflow.

**◆ addAll**

```
public void addAll(ArrayBag<E> addend)
```

Add the contents of another bag to this bag.

**Parameter:**

addend – a bag whose contents will be added to this bag

**Precondition:**

The parameter, addend, is not null.

**Postcondition:**

The elements from addend have been added to this bag.

**Throws: NullPointerException**

Indicates that addend is null.

**Throws: OutOfMemoryError**

Indicates insufficient memory to increase the size of this bag.

**Note:**

Creating a bag with capacity beyond Integer.MAX\_VALUE causes arithmetic overflow.

(continued)

(FIGURE 5.2 continued)

◆ **addMany**

`public void addMany(E... elements)`

Add a variable number of new elements to this bag. If these new elements would take this bag beyond its current capacity, then the capacity is increased before adding the new elements.

**Parameter:**

`elements` – a variable number of new elements that are all being added

**Postcondition:**

New copies of all the elements have been added to this bag.

**Throws:** `OutOfMemoryError`

Indicates insufficient memory for increasing the capacity.

**Note:**

Creating a bag with capacity beyond `Integer.MAX_VALUE` causes arithmetic overflow.

◆ **clone**

`public ArrayBag<E> clone()`

Generate a copy of this bag.

**Returns:**

The return value is a copy of this bag. Subsequent changes to the copy will not affect the original, nor vice versa.

**Throws:** `OutOfMemoryError`

Indicates insufficient memory for creating the clone.

◆ **countOccurrences**

`public int countOccurrences(E target)`

Accessor method to count the number of occurrences of a particular element in this bag.

**Parameter:**

`target` – the reference to an `E` object to be counted

**Returns:**

The return value is the number of times that `target` occurs in this bag. If `target` is non-null, then the occurrences are found using the `target.equals` method.

◆ **ensureCapacity**

`public void ensureCapacity(int minimumCapacity)`

Change the current capacity of this bag.

**Parameter:**

`minimumCapacity` – the new capacity for this bag

**Postcondition:**

This bag's capacity has been changed to at least `minimumCapacity`. If the capacity was already at or greater than `minimumCapacity`, then the capacity is left unchanged.

**Throws:** `OutOfMemoryError`

Indicates insufficient memory for new `Object[minimumCapacity]`.

(continued)

**274 Chapter 5 / Generic Programming***(FIGURE 5.2 continued)***♦ getCapacity**

```
public int getCapacity()
```

Accessor method to determine the current capacity of this bag. The add method works efficiently (without needing more memory) until this capacity is reached.

**Returns:**

the current capacity of this bag

**♦ grab**

```
public E grab()
```

Accessor method to retrieve a random element from this bag.

**Precondition:**

This bag is not empty.

**Returns:**

a randomly selected element from this bag

**Throws: IllegalStateException**

Indicates that the bag is empty.

**♦ remove**

```
public boolean remove(E target)
```

Remove one copy of a specified element from this bag.

**Parameter:**

target – the element to remove from this bag

**Postcondition:**

If target was found in this bag, then one copy of target has been removed and the method returns true. Otherwise, this bag remains unchanged, and the method returns false.

**♦ size**

```
public int size()
```

Accessor method to determine the number of elements in this bag.

**Returns:**

the number of elements in this bag

**♦ trimToSize**

```
public void trimToSize()
```

Reduce the current capacity of this bag to its size (i.e., the number of elements it contains).

**Postcondition:**

This bag's capacity has been changed to its current size.

**Throws: OutOfMemoryError**

Indicates insufficient memory for altering the capacity.

(continued)



**276 Chapter 5 / Generic Programming**

(FIGURE 5.2 continued)

```
public ArrayBag(int initialCapacity)
{
    if (initialCapacity < 0)
        throw new IllegalArgumentException
            ("initialCapacity is negative: " + initialCapacity);
    manyItems = 0;
    data = new Object[initialCapacity];
}

public void add(E element)
|| The implementation is unchanged from the original in Figure 3.7 on page 137.

public void addAll(ArrayBag<E> addend)
|| The implementation is unchanged from the original in Figure 3.7 on page 137.

@SuppressWarnings("unchecked")<-->
public void addMany(E... elements)
|| The implementation is unchanged from the original in Figure 3.7 on page 137.

@SuppressWarnings("unchecked")<-->
public ArrayBag<E> clone( )
{ // Clone an ArrayBag<E>.
    ArrayBag<E> answer;

    try
    {
        answer = (ArrayBag<E>) super.clone( );
    }
    catch (CloneNotSupportedException e)
    {
        // This exception should not occur. But if it does, it would probably indicate a
        // programming error that made super.clone unavailable.
        // The most common error would be forgetting the "implements Cloneable"
        // clause at the start of this class.
        throw new RuntimeException
            ("This class does not implement Cloneable");
    }

    answer.data = data.clone( );
    return answer;
}
```

See the discussion of  
warnings on page 265.

(continued)

(FIGURE 5.2 continued)

```
public int countOccurrences(E target)
{
    int answer;
    int index;

    answer = 0;

    if (target == null)           ←
    { // Count how many times null appears in the bag.
        for (index = 0; index < manyItems; index++)
            if (data[index] == null)
                answer++;
    }
    else
    { // Use target.equals to determine how many times the target appears.
        for (index = 0; index < manyItems; index++)
            if (target.equals(data[index]))
                answer++;           ←
    }

    return answer;
}

public void ensureCapacity(int minimumCapacity)
{
    Object[ ] biggerArray;

    if (data.length < minimumCapacity)
    {
        biggerArray = new Object[minimumCapacity];
        System.arraycopy(data, 0, biggerArray, 0, manyItems);
        data = biggerArray;
    }
}

public int getCapacity( )
|| The implementation is unchanged from the original in Figure 3.7 on page 140.
```

*Special code is needed to handle the case where the target is null.*

*For a non-null target, we use target.equals instead of the “==” operator.*

(continued)

**278 Chapter 5 / Generic Programming**

(FIGURE 5.2 continued)

```
@SuppressWarnings("unchecked") // See the warnings discussion on page 265.
public E grab( )
{
    int i;

    if (manyItems == 0)
        throw new IllegalStateException("Bag size is zero.");

    i = (int) (Math.random() * manyItems); // From 0 to manyItems-1
    return (E) data[i];
}

public boolean remove(E target)
{
    int index; // The location of target in the data array

    // First, set index to the location of target in the data array.
    // If target is not in the array, then index will be set equal to manyItems.
    if (target == null)
    { // Find the first occurrence of the null reference in the bag.
        index = 0;
        while ((index < manyItems) && (data[index] != null))
            index++;
    }
    else
    { // Use target.equals to find the first occurrence of the target.
        index = 0;
        while ((index < manyItems) && (!target.equals(data[index])))
            index++;
    }

    if (index == manyItems)
        return false; // The target was not found, so nothing is removed.
    else
    { // The target was found at data[index].
        manyItems--;
        data[index] = data[manyItems];
        data[manyItems] = null; <----- The unused array location
                                is set to null to allow Java
                                to collect the unused
                                memory.
        return true;
    }
}

public int size( )
|| The implementation is unchanged from the original in Figure 3.7 on page 140.
```

(continued)

(FIGURE 5.2 continued)

```
public void trimToSize( )
{
    Object[ ] trimmedArray;

    if (data.length != manyItems)
    {
        trimmedArray = new Object[manyItems];
        System.arraycopy(data, 0, trimmedArray, 0, manyItems);
        data = trimmedArray;
    }
}

public static <E> ArrayBag<E> union(ArrayBag<E> b1, ArrayBag<E> b2)
{
    // If either b1 or b2 is null, then a NullPointerException is thrown.
    ArrayBag<E> answer =
        new ArrayBag<E>(b1.getCapacity( ) + b2.getCapacity( ));

    System.arraycopy(b1.data, 0, answer.data, 0, b1.manyItems);
    System.arraycopy(b2.data, 0, answer.data, b1.manyItems, b2.manyItems);
    answer.manyItems = b1.manyItems + b2.manyItems;
    return answer;
}
```

---

### Using the Bag of Objects

Using the bag of objects is easy. The program imports `edu.colorado.collections.ArrayBag`, and then a bag of objects can be declared. The same program may have several different bags for different purposes, or it may even have some of the original bags, such as an `IntArrayBag`.

Figure 5.3 shows a demonstration program that uses three bags of strings. The program asks the user to type several adjectives and names. These words are placed in the bags, and then elements are grabbed out of the bags for the program to write a silly story called “Life.”

**FIGURE 5.3** Demonstration Program for the ArrayBag Generic ClassJava Application Program

```
// FILE: Author.java
// This program reads some words into bags. Then a silly story is written using these words.
import edu.colorado.collections.ArrayBag;
import java.util.Scanner; ←
public class Author
{
    private static Scanner stdin = new Scanner(System.in);

    public static void main(String[ ] args)
    {
        final int WORDS_PER_BAG = 4; // Number of items per bag
        final int MANY_SENTENCES = 3; // Number of sentences in story

        ArrayBag<String> good   = new ArrayBag<String>(WORDS_PER_BAG);
        ArrayBag<String> bad    = new ArrayBag<String>(WORDS_PER_BAG);
        ArrayBag<String> names  = new ArrayBag<String>(WORDS_PER_BAG);
        int line;

        // Fill the three bags with items typed by the program's user.
        System.out.println("Help me write a story.\n");
        getWords(good, WORDS_PER_BAG, "adjectives that describe a good mood");
        getWords(bad, WORDS_PER_BAG, "adjectives that describe a bad mood");
        getWords(names, WORDS_PER_BAG, "first names");
        System.out.println("Thank you for your kind assistance.\n");

        // Use the items to write a silly story.
        System.out.println("LIFE");
        System.out.println("by A. Computer\n");
        for (line = 1; line <= MANY_SENTENCES; line++)
        {
            System.out.print((String) names.grab( ));
            System.out.print(" was feeling ");
            System.out.print((String) bad.grab( ));
            System.out.print(", yet he/she was also ");
            System.out.print((String) good.grab( ));
            System.out.println(".");
        }
        System.out.println("Life is " + (String) bad.grab( ) + ".\n");
        System.out.println("The " + (String) good.grab( ) + " end.");
    }
}
```

The Scanner class is  
described in Appendix B.

(continued)

(FIGURE 5.3 continued)

```
public static void getWords(ArrayBag<String> b, int n, String prompt)
// Postcondition: The parameter, prompt, has been written as a prompt
// to System.out. Then n strings have been read using stdin.next,
// and these strings have been placed in the bag.
{
    String userInput;
    int i;

    System.out.print("Please type " + n + " " + prompt);
    System.out.println(", separated by spaces.");
    System.out.println("Press the <return> key after the final entry:");
    for (i = 1; i <= n; i++)
    {
        userInput = stdin.next();
        b.add(userInput);
    }
    System.out.println();
}
```

### A Sample Dialogue

Help me write a story.

Please type 4 adjectives that describe a good mood, separated by spaces.

Press the <return> key after the final entry:

**joyous happy lighthearted glad**

Please type 4 adjectives that describe a bad mood, separated by spaces.

Press the <return> key after the final entry:

**sad glum melancholy blue**

Please type 4 first names, separated by spaces.

Press the <return> key after the final entry:

**Michael Janet Tim Hannah**

Thank you for your kind assistance.

LIFE by A. Computer

Tim was feeling glum, yet he/she was also glad.

Michael was feeling melancholy, yet he/she was also joyous.

Hannah was feeling blue, yet he/she was also joyous.

Life is blue.

The lighthearted end.

---

### Details of the Story-Writing Program

The bags are declared in the demonstration program as you would expect:

```
ArrayBag<String> good    = new ArrayBag<String>(WORDS_PER_BAG);
ArrayBag<String> bad;    = new ArrayBag<String>(WORDS_PER_BAG);
ArrayBag<String> names;  = new ArrayBag<String>(WORDS_PER_BAG);
```

The number, WORDS\_PER\_BAG, is 4 in the story program, so each of these bags has an initial capacity of 4. To get the actual words from the user, the program calls `getWords`, with this heading:

```
public static void getWords
(ArrayBag<String> b, int n, String prompt)
```

The method uses the third parameter, `prompt`, as part of a message that asks the user to type `n` words. For example, if `prompt` is the string constant "first names" and `n` is 4, then the `getWords` method writes this prompt:

```
Please type 4 first names, separated by spaces.
Press the <return> key after the final entry:
```

The method then reads `n` strings by using the `Scanner` class (described in Appendix B).

As the strings are read, the `getWords` method places them in the bag by activating `b.add`. In all, the main program activates `getWords` three times to get three different kinds of strings. The literary merit of the program's story is debatable, but the ability to use bags of objects is clearly important.

### Self-Test Exercises for Section 5.3

13. We converted an `IntArrayBag` to a generic `ArrayBag<E>`. During the conversion, does every occurrence of `int` get changed to `E`?
14. Our bag is a bag of references to objects. Can the null reference be placed into our bag?
15. Write some code that declares a bag of integers and then puts the numbers 1 through 10 into the bag. The numbers are objects of the wrapper class, `Integer`, not the primitive class `int`.
16. The original `countOccurrences` method tested for the occurrence of a target by using the boolean expression `target == data[index]`. What different boolean expression is used in the `countOccurrences` method of the bag of objects?
17. What technique did the story-writing program use to read strings from the keyboard?
18. Consider the following code that puts a `Location` into a bag, changes the name, and then tests whether the original name is in the bag. The `Location` class is from Figure 2.5 on page 67. What does the code print?

```
ArrayBag<Location> points = new ArrayBag<Location>();
Location origin = new Location(0,0);
Location moving = new Location(0,0);
points.add(moving);
moving.shift(5,10);
System.out.println(points.countOccurrences(origin));
System.out.println(points.countOccurrences(moving));
```

## 5.4 GENERIC NODES

### Nodes That Contain Object Data

Our node class from Chapter 4 can also be converted to a generic class. In the conversion, the new class, called `Node`, allows us to build linked lists of nodes in which the type of data in each node is determined by the generic type parameter. Here is a comparison of the original `IntNode` declaration with our new generic class:

#### Original IntNode:

```
public class IntNode
{
    private int data;
    IntNode link;
    || The methods use
    || the int data.
}
```

#### Generic Node Class:

```
public class <E> Node
{
    private E data;
    Node<E> link;
    || The methods use
    || the E data.
}
```

With the new `Node` class, each node contains a piece of data that is a reference to an `E` object, but we don't specify exactly what `E` is. The implementation of the methods is based on the `IntNode` methods, following the same steps that we used before on page 269. The resulting `Node` class is online at <http://www.cs.colorado.edu/~main/edu/colorado/nodes/node.java>. You may find yourself using it beyond this book, too.

## PITFALL

### MISUSE OF THE EQUALS METHOD

When you convert a collection class to contain objects, it is tempting to blindly change every occurrence of the `==` operator to use the `equals` method instead. There are two pitfalls to beware of.

First, beware of null references. You cannot activate the `equals` method of a null reference. For example, here is the implementation of the new `listSearch` method from the generic `Node` class. The method searches the linked list for an

**284 Chapter 5 / Generic Programming**

occurrence of a particular target and returns a reference to the node that contains the target. If no such node is found, then the null reference is returned:

```
public static <E> Node<E> listSearch(Node<E> head, E target)
{
    Node<E> cursor;

    if (target == null)
    { // Search for a node in which the data is the null reference.
        for (cursor = head; cursor != null; cursor = cursor.link)
            if (cursor.data == null)
                return cursor;
    }
    else
    { // Search for a node that contains the non-null target.
        for (cursor = head; cursor != null; cursor = cursor.link)
            if (target.equals(cursor.data))
                return cursor;
    }

    return null;
}
```

The target may be the null reference; in that case, we're searching for a node in which the data is null. This search is carried out in the first part of the large if-statement. We test whether a particular node contains null data with the boolean expression `cursor.data == null`. On the other hand, for a non-null target, the search is carried out in the else-part of the large if-statement. We test whether a particular node contains a non-null target with the boolean expression `target.equals(cursor.data)`.

In general, the `equals` method may be used only when you know that the target is non-null.

The second thing to beware of: You should change an equality test “`==`” to the `equals` method only where the program is comparing data. Don't change other comparisons. For example, here is the `listPart` method of the generic `Node` class:

```
public static <E> Object[] listPart(Node<E> start, Node<E> end)
{
    Node<E> copyHead;
    Node<E> copyTail;
    Node<E> cursor;
    Object[ ] answer = new Object[2];

    // Check for illegal null at start or end.
    if (start == null)
        throw new IllegalArgumentException("start is null.");

```

*Note that the return type must now be  
an array of Objects since generic arrays  
are forbidden.*

```
if (end == null)
    throw new IllegalArgumentException("end is null.");

// Make the first node for the newly created list.
copyHead = new Node<E>(start.data, null);
copyTail = copyHead;
cursor = start;

// Make the rest of the nodes for the newly created list.
while (cursor != end)
{
    cursor = cursor.link;
    if (cursor == null)
        throw new IllegalArgumentException
            ("end node was not found on the list.");
    copyTail.addNodeAfter(cursor.data);
    copyTail = copyTail.link;
}

// Return the head and tail references.
answer[0] = copyHead;
answer[1] = copyTail;
return answer;
}
```

The method copies part of a linked list, extending from the specified start node to the specified end node. The large while-loop does most of the work, and this loop continues while `cursor != end`. This boolean expression is checking whether `cursor` refers to a different node than `end`. The expression becomes `false` when `cursor` refers to the exact same node that `end` refers to, and at that point the loop ends. Do not change this expression to use the `equals` method.

When the purpose of a boolean expression is to test whether two references refer to the exact same object, then use the “`==`” or “`!=`” operator. Do not use the `equals` method.

## Other Collections That Use Linked Lists

Using the generic `Node` class, we can implement other collections that use linked lists. For example, we can implement another bag of objects that stores its objects in a linked list. In fact, we’ll do exactly this in Section 5.6, but first you need to see a Java feature called *interfaces* that supports generic programming.

## Self-Test Exercises for Section 5.4

19. Why was extra code added to the new `listSearch` method?

**286 Chapter 5 / Generic Programming**

20. Suppose the while-loop in `listPart` was changed to:
- ```
while (!cursor.equals(end))
```

Give an example in which this incorrect `listPart` method would not copy all of the nodes that it is supposed to copy.

21. Suppose `x` and `y` are non-null references to two nodes. The data in each node is a non-null `Object`. Write two boolean expressions: (1) an expression that is `true` if `x` and `y` refer to exactly the same node, and (2) an expression that is `true` if the data from the `x` node is equal to the data from the `y` node. Use the `equals` method where appropriate.

## 5.5 INTERFACES AND ITERATORS

Java provides another feature to support generic programming: *interfaces*, which allow methods to work with objects of a class when only a limited amount of information is known about that class. This section shows you how to use some basic interfaces that are provided as part of Java's Application Programming Interface (API). We concentrate on the `Iterator` interface, which is particularly important for data structures programming.

Programmers can also create new interfaces (that aren't part of the API), but we won't cover that topic here.

### Interfaces

*interface: a list of related methods for a class to implement*

A Java **interface** is primarily a list of related methods that a programmer may want to implement in a single class. For example, Java's `AudioClip` interface indicates that a class that implements the `AudioClip` interface will have three methods with these headings:

```
public void loop()
public void play()
public void stop()
```

*advantages of implementing a known interface*

The `AudioClip` interface is intended for classes in which each object can produce a sound on a computer with sound capabilities. For example, the `play` method is supposed to play the sound one time. A class that has these three methods (and perhaps other methods, too) is said to **implement the AudioClip interface**. If you're writing a class with these kinds of audio capabilities, then implementing the `AudioClip` interface will help other programmers understand your intentions and use your class in a way that is consistent with other classes that provide the same capabilities. In addition, using interfaces supports generic programming because programs can be written that use `AudioClip` variables, and these programs will work with any possible implementation of the interface.



**288 Chapter 5 / Generic Programming**

```
public boolean hasNext( )
public E next( )
public void remove( )
```

As with any generic class, we could have used some name other than E, but it's common practice to use the name E for "element."

Intuitively, an iterator is able to step through a sequence of elements, each of which is an E object. Usually, the sequence comes from a collection such as a generic bag. A program activates `hasNext()` to determine whether there are any more elements in the sequence. If there are more elements, then `hasNext()` returns `true`, and the program can then activate `next()` to obtain the next element.

An iterator also has a `remove` method. This method removes the element that was given by the most recent call to `next()`. Sometimes an Iterator does not allow elements to be removed. In this case, activating the `remove` method results in an `UnsupportedOperationException`.

We could write many classes that implement the `Iterator` interface. We're going to write one particular generic class called `Lister`. When a `Lister` is constructed, it is given a reference to the head of a generic linked list of objects. The `Lister` stores a copy of this reference variable in its own instance variable, called `current`. Each subsequent activation of `next` returns one element from the linked list and moves `current` on to the next element. When the last element has been returned, `hasNext` will return `false`. The `Lister` does not allow removal of elements, so any attempt to activate `remove` results in an exception.

The specification and implementation of the `Lister` class are shown in Figure 5.4.

### How to Write a Generic Class That Implements a Generic Interface

A generic class can implement a generic interface by following the same three steps that we've seen for an ordinary interface:

1. **Read the interface's documentation.** For the `Iterator` generic interface, this documentation is at the `Iterator` link of <http://download.oracle.com/javase/7/docs/api/index.html>.
2. **Tell the compiler that you are implementing a generic interface.** For our generic class, `Lister`, we write:

```
public class <E> Lister implements Iterator<E>
```

3. **Implement the class in the usual way.** We'll discuss some of the `Lister` implementation details on page 291.

**FIGURE 5.4** Specification and Implementation for the Lister Class

### Generic Class Lister

◆ **public class Lister<E> from the package edu.colorado.nodes**

A Lister<E> implements Java's Iterator<E> generic interface for a linked list of objects of type E. Note that this implementation does not support the remove method. Any activation of remove results in an UnsupportedOperationException.

#### Specification

◆ **Constructor for the Lister<E>**

```
public Lister(Node<E> head)
```

Initialize a Lister with a particular linked list of objects.

**Parameter:**

head – a head reference for a linked list of objects

**Postcondition:**

Subsequent activations of next will return the elements from this linked list, one after another. If the linked list changes in any way before all the elements have been returned, then the subsequent behavior of this Lister is unspecified.

◆ **hasNext**

```
public boolean hasNext( )
```

Determine whether there are any more elements in this Lister.

**Returns:**

true if there are more elements in this Lister; false otherwise

◆ **next**

```
public E next( )
```

Retrieve the next element of this Lister.

**Precondition:**

hasNext( )

**Returns:**

The return value is the next element of this Lister. Note that each element is returned only once, and then the Lister automatically advances to the next element.

**Throws:** NoSuchElementException

Indicates that hasNext() is false.

◆ **remove**

```
public E remove( )
```

Although remove is part of the Iterator interface, it is not provided in this implementation.

**Throws:** UnsupportedOperationException

This exception is always thrown!

(continued)

**290** Chapter 5 / Generic Programming

(FIGURE 5.4 continued)

Implementation

```
// File: Lister.java from the package edu.colorado.nodes
// Documentation is available on the previous page or from the Lister link at
// http://www.cs.colorado.edu/~main/docs/.
package edu.colorado.nodes;

import java.util.Iterator;
import java.util.NoSuchElementException;
import edu.colorado.nodes.Node;

public class Lister<E> implements Iterator<E>
{
    // Invariant of the Lister class:
    //   The instance variable current is the head reference for the linked list that contains
    //   the elements that have not yet been provided by the next method. If there
    //   are no more elements to provide, then current is the null reference.
    private Node<E> current;

    public Lister(Node<E> head)
    {
        current = head;
    }

    public boolean hasNext( )
    {
        return (current != null);
    }

    public E next( )
    {
        E answer;

        if (!hasNext( ))
            throw new NoSuchElementException("The Lister is empty.");

        answer = current.getData( );
        current = current.getLink( );

        return answer;
    }

    public void remove( )
    {
        throw new UnsupportedOperationException("Lister has no remove method.");
    }
}
```

---

## The Lister Class

We implement the `Lister` class as part of our package `edu.colorado.nodes`. It is a generic class that depends on an unspecified type, `E`, which is the type of element in a generic linked list. The `Iterator` interface is part of `java.util`, so the `Lister` implementation starts by importing `java.util.Iterator`. Also, the `next` method throws a `NoSuchElementException` to indicate that it has run out of elements, so we also import `java.util.NoSuchElementException`. The `remove` method also throws an exception, but no import statement is needed because `UnsupportedOperationException` is part of `java.lang` (which is automatically imported for any program).

Any program that creates linked lists can use the `Lister` class. For example, here's some code that creates a list of strings and then uses a `Lister` to step through those strings one at a time:

```
import edu.colorado.nodes.Node;
import edu.colorado.nodes.Lister;
...
Node<String> head;      // Head node of a small linked list
Node<String> middle;    // Second node of the same list
Node<String> tail;      // Tail node of the same list
Lister<String> print;   // Used to print the small linked list

// Create a small linked list.
tail = new Node<String>("Larry", null);
middle = new Node<String>("Curly", tail);
head = new Node<String>("Moe", middle);
// The list now has "Moe", "Curly", and "Larry". We'll print these strings.
print = new Lister<String>(head);
while (print.hasNext())
    System.out.println(print.next());
```

The while-loop steps through the elements of the list, printing the three strings:

```
Moe
Curly
Larry
```

### PITFALL



#### DON'T CHANGE A LIST WHILE AN ITERATOR IS BEING USED

The `Lister` contains one warning in the constructor documentation: If the linked list changes in any way before all the elements have been returned, then the subsequent behavior of the `Lister` is unspecified. In other words, while the `Lister` is being used, the underlying linked list must not be altered. The reason for this is that the `Lister` uses the original linked list rather than making a copy of that list. This is the way that many of Java's built-in iterators work. However, an alternative is to make a copy of the original linked list, which we will ask you to do in a self-test exercise.

**292 Chapter 5 / Generic Programming****The Comparable Generic Interface**

Java has a generic interface called `Comparable<T>` that requires just one method:

```
public int compareTo(T obj)
```

This interface is intended for any class in which two objects `x` and `y` can always be compared to each other with one of three possible results:

- `x` is less than `y`
- `x` and `y` are equal to each other
- `y` is less than `x`

Many classes have this kind of natural ordering among objects. For example, two `Integer` objects can be compared. Therefore, Java's `Integer` class is implemented as:

```
public class Integer implements Comparable<Integer>...
```

Because the `Integer` class implements the `Comparable<Integer>` interface, it must have a `compareTo` method with this heading (where the generic type parameter has been instantiated as `Integer`):

```
public int compareTo(Integer obj)
```

**FIGURE 5.5 Some Java Classes That Implement a Comparable Interface**

In each case, a negative return value means that the value of `x` is less than the value of `y`; a zero return value means that the two values are equal; a positive return value means that the value of `x` is greater than the value of `y`.

| Java class | Implements                               | What is compared by <code>x.compareTo(y)</code>                                                                                                                                                                                                                                                                         |
|------------|------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Character  | <code>Comparable&lt;Character&gt;</code> | The ASCII values of <code>x</code> and <code>y</code>                                                                                                                                                                                                                                                                   |
| Date       | <code>Comparable&lt;Date&gt;</code>      | A Date is a specific point in time on a specific date. When <code>x.compareTo(y)</code> is negative, the Date <code>x</code> occurred before the Date <code>y</code> .                                                                                                                                                  |
| Double     | <code>Comparable&lt;Double&gt;</code>    | The double values of <code>x</code> and <code>y</code> .                                                                                                                                                                                                                                                                |
| Integer    | <code>Comparable&lt;Integer&gt;</code>   | The int values of <code>x</code> and <code>y</code> .                                                                                                                                                                                                                                                                   |
| String     | <code>Comparable&lt;String&gt;</code>    | The strings <code>x</code> and <code>y</code> are compared lexicographically, which means that strings of lowercase letters are compared alphabetically. When <code>x.compareTo(y)</code> is negative, it means that <code>x</code> is alphabetically before <code>y</code> (if they are strings of lowercase letters). |

The documentation for the `compareTo` method is lengthy, but the gist is simple: Any two objects `x` and `y` of the class can be compared by activating `x.compareTo(y)`. The method returns an integer that is negative (if `x` is less than `y`), zero (if `x` and `y` are equal), or positive (if `x` is greater than `y`).

The `Comparable` interface uses the name `T` for the generic type parameter (rather than `E`). We could have used any name instead of `T`, but `T` is common because it refers to any “type” rather than an “element” in a collection class.

Figure 5.5 shows some of the other Java classes that implement a `Comparable` interface.

### Parameters That Use Interfaces

Interfaces support generic programming because it is often possible to implement a method based only on knowledge about an interface. For example, we can write a method that plays an `AudioClip` a specified number of times:

```
public static void playRepeatedly(AudioClip clip, int n)
{ // Play clip n times.
    int i;

    for (i = 0; i < n; i++)
        clip.play();
}
```

The data type of the actual argument for the `clip` parameter may be any data type that implements the `AudioClip` interface.

#### Using an Interface as the Type of a Parameter

When an interface name is used as the type of a method’s parameter, the actual argument must be a data type that implements the interface.

A generic method may use a generic interface as the data type of a parameter. For example, we can write a method that computes how many of the elements in an array of `T` objects are less than some non-null target, provided that the data type of the target is some type that implements `Comparable<T>`. The method’s implementation looks like this, depending on the generic type parameter `T` (as indicated by `<T>` before the return type):

**294 Chapter 5 / Generic Programming**

```
public static <T> int smaller(T[] data, Comparable<T> target)
{ // The return value is the number of objects in the data array that
  // are less than the non-null target b (using b.compareTo to compare
  // b to each object in the data array).
  int answer = 0;

  for (T next : data)
  {
    if (b.compareTo(next) > 0)
    { // b is greater than the next element of data.
      answer++
    }
  }
  return answer;
}
```

In this example, the argument for `data` could be a `String` array, and `target` could be a `String` (since the `String` class implements `Comparable<String>`). Or `data` could be an `Integer` array, and `target` could be an `Integer`. Or `data` could be any array of `T` objects, so long as the type of `target` implements `Comparable<T>`.

In each of these examples, we used an interface as the type of a parameter. This results in restrictions on what type of argument can be used with the method. In Chapter 13, we'll see how to formulate other, more powerful restrictions.

### Using `instanceof` to Test Whether a Class Implements an Interface

A programmer can test whether a given object actually does implement a specified interface. For example, suppose you are writing a method to test whether an object called `info` implements the `AudioClip` interface. If so, the `play` method is activated; otherwise, some other technique is used to display some information about the object. The method could begin like this:

```
if ((information instanceof AudioClip)
    information.play();
else
  ...
```

The test is carried out with the `instanceof` operator, with the general form:

variable `instanceof` interface-or-class-name

The test is `true` if the variable on the left is a reference to an object of the specified type. Notice that the type name can be an ordinary class name (such as `(information instanceof String)`), or it may be the name of an interface (such as `(information instanceof AudioClip)`).

You can also test whether an object implements a generic interface. You can test for a specific instantiation of the generic interface, such as this:

```
if ((example instanceof Iterator<String>)) ...
```

For example, the boolean expression just shown will be true if `example` is a `Lister<String>` object, since the `Lister<String>` class implements the `Iterator<String>` interface.

### The Cloneable Interface

Throughout the book, all of our collection classes have implemented the `Cloneable` interface, which has a somewhat peculiar meaning. You might think that the `Cloneable` interface specifies that the class must implement a `clone` method, but this is wrong; there are no methods specified in the `Cloneable` interface. So what's the purpose of the `Cloneable` interface? The purpose comes from the behavior of the `clone()` method of Java's `Object` class. That `clone` method carries out these two steps:

1. Check to see whether the class has implemented the `Cloneable` interface. If not, a `CloneNotSupportedException` is thrown.
2. Otherwise, the `clone` is made by copying each of the instance variables of the original.

As you can see, the `Object clone` method checks to see whether the object has implemented the `Cloneable` interface. It does this by using the boolean test `(obj instanceof Cloneable)`, and in fact, the only real purpose of implementing the `Cloneable` interface is so the `Object clone` method can test `(obj instanceof Cloneable)`. Therefore, if you write a `clone` method of your own and that `clone` method activates `super.clone` from Java's `Object` type, then your class must implement `Cloneable` to avoid a `CloneNotSupportedException`.

### Self-Test Exercises for Section 5.5

22. Find the documentation for Java's `CharacterSequence` interface in the API. What methods does this interface require?
23. Write a generic method with one parameter that is a head reference for a linked list of nodes. The method looks through all the nodes of the list and returns a count of the number of nodes that contain null data. Use a `Lister` object to search through the linked list.
24. Write a generic method with one parameter that is a head reference for a linked list of nodes and a second parameter `x` that is a `Comparable<T>` object. The precondition of the method requires that `x` is non-null. The method returns a reference to the data in the first node that it finds in

**296 Chapter 5 / Generic Programming**

which the data is greater than or equal to  $x$ . If there is no such node, then the method returns null.

25. Reimplement the `Lister` constructor so that a copy of the original linked list is used, rather than using the original linked list directly.
26. Write a boolean expression that will be true if and only if the data type of a variable  $x$  implements the `Comparable<String>` interface.
27. How many methods are required for a class that implements `Cloneable`?
28. Write a class, `ArrayIterator`, which implements the `Iterator<E>` interface. The constructor has one argument, which is an array of  $E$  objects. The `ArrayIterator` returns the components of the array one at a time through its `next` method.

## 5.6 A GENERIC BAG CLASS THAT IMPLEMENTS THE ITERABLE INTERFACE (OPTIONAL SECTION)

Normally, when a program needs to keep track of some elements, it puts the elements into a bag or some other collection rather than building a linked list in a haphazard manner. Then, if the program needs to step through the elements, it will ask the bag to provide an iterator that contains all of the elements.

With this in mind, we'll implement one more bag class. The bag will be called `LinkedBag` in the package `edu.colorado.collections`, with these features:

- It is a generic bag of objects (with a generic type parameter  $E$ ).
- The elements are stored in a linked list.
- The bag implements a generic interface called `Iterable`, which requires one method:

```
public Iterator<E> iterator()
```

This new method returns an iterator that can be used to step through all the elements currently in the bag. (In fact, the iterator that it returns will be a `Lister<E>` object, using the generic `Lister` class from the previous section.)

*the generic  
Iterable interface  
requires one  
method that  
returns an  
iterator*

Here's part of a program that uses the new `iterator` method:

```
import edu.colorado.collections.LinkedBag;
import java.util.Iterator;
...
LinkedBag<String> stooges = new LinkedBag<String>();
Iterator<String> print; // Used to print the strings from the bag
```

```
// Put a few strings in the bag.  
stooges.add("Larry");  
stooges.add("Curly");  
stooges.add("Moe");  
  
// The bag now has "Moe", "Curly", and "Larry". We'll print these strings.  
print = stooges.iterator();  
while (print.hasNext())  
    System.out.println(print.next());
```

The while-loop will print the strings “Moe”, “Curly”, and “Larry”—although the order of printing depends on exactly how the bag stores the elements in the linked list.

## PROGRAMMING TIP

### ENHANCED FOR-LOOPS FOR THE ITERABLE INTERFACE

Java includes an enhanced version of the for-loop that can be used with any class that implements the `Iterable` interface. In the case of the `stooges` bag shown above, we could rewrite the loop to print the names as follows:

```
// The bag now has "Moe", "Curly", and "Larry". We'll print these strings.  
for (String next : stooges)  
    System.out.println(next);
```

The new version of the for-loop is similar to the enhanced for-loops for arrays (page 111). The general format is shown here:

This form of the for-loop is called **iterating over a collection**. The general format is given here, although you can use any variable name instead of `item`:

```
for (type of the collection elements item : name of the collection)  
{  
    // Do something with item.  
    // The first time through the loop, item will be set equal to the  
    // first element that the collection's iterator provides; the second time  
    // through the loop, item will be set equal to the iterator's next  
    // element; and so on.  
    ...  
}
```

This form of the for-loop can be used with any collection that implements the `Iterable` interface.

### Implementing a Bag of Objects Using a Linked List and an Iterator

For the most part, the implementation of the new bag is the same as the Chapter 4 class `IntLinkedBag`, which stored its elements in a linked list. To convert to a bag of Java Objects, we'll follow the steps listed on page 269. We'll also implement the new `iterator` method, which needs only one line to construct a `Lister` that contains all the elements. Here's the implementation:

```
public Iterator<E> iterator()
// Method of the LinkedBag class to return an Iterator that
// contains all the elements that are currently in the bag.
{
    return (Iterator<E>) new Lister<E>(head);
}
```

In this implementation, `head` is the bag's instance variable that is a reference to the head node of the linked list where the bag keeps its elements. There's little new in the rest of the implementation of `edu.colorado.collections.LinkedBag`, so we've put it online at <http://www.cs.colorado.edu/~main/edu/colorado/collections/LinkedBag.java>.

## PROGRAMMING TIP

### EXTERNAL ITERATORS VERSUS INTERNAL ITERATORS

The `iterator` method of the bag class provides an iterator for all the elements in the bag. This kind of iterator is called an **external iterator** because it is an object separate from the bag. This is different from our original bags (which had only a `countOccurrences` method to examine elements). The `iterator` method also differs from the sequence class of Section 3.3. With a sequence, a program can step through the elements, but only by accessing the sequence's current element. This kind of access, directly through a method of the class, is called an **internal iterator**.

Internal iterators are a simple solution to the problem of stepping through the elements of a collection class, but internal iterators also have a problem. Suppose your program needs to step through the elements of a bag in two different ways. With an internal iterator, the two traversals of the collection elements must occur sequentially—stepping through all the elements for the first time, then restarting the iterator, and finally stepping through all the elements for the second time. However, sometimes you would like to have the two different traversals occurring simultaneously. The first traversal could start, stepping through part of the elements, but before the first traversal is finished, the second traversal starts its work. Simultaneous traversals are not easily handled with internal iterators.

The problem is solved by having a method that provides an external iterator—an object that is separate from the bag. The bag's `iterator` method does just

*A Generic Bag Class That Implements the Iterable Interface (Optional Section)* 299

this. If we need to step through the elements of a bag two different times, then we can create two separate external iterators for the bag.

In general, a method that provides an external iterator (and thus implements the `Iterable` interface) is better than providing just an internal iterator.

### Summary of the Four Bag Implementations

If nothing else, you should now know how to program your way out of a paper bag using one of the four bag classes listed in Figure 5.6. The bag class is a good example to show all these different approaches to implementing a collection class, but keep in mind that the same approaches can be used for other collection classes.

### Self-Test Exercises for Section 5.6

29. What is the primary difference between the `LinkedBag` suggested in this section and the `IntLinkedBag` from Section 4.4?
30. What is the primary difference between the generic `LinkedBag` suggested in this section and the generic `ArrayBag` from Section 5.3?
31. Write some code that allows the user to type in 10 strings. Each string is put in a bag (using the `LinkedBag` from Section 5.6). When the user is finished, the 10 strings are printed.
32. Compare the benefits of an internal iterator to a method that provides an external iterator.
33. What kind of collection can be used in Java's enhanced for-loop?

**FIGURE 5.6** Our Four Bag Classes

| Approach                                                                                    | Class                                              | Where to find it                                                                                                                                                |
|---------------------------------------------------------------------------------------------|----------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Store integer elements in an array.                                                         | <code>edu.colorado.collections.IntArrayBag</code>  | Figure 3.1 on page 119 and Figure 3.7 on page 137                                                                                                               |
| Store integer elements in a linked list.                                                    | <code>edu.colorado.collections.IntLinkedBag</code> | Figure 4.12 on page 216 and Figure 4.17 on page 230                                                                                                             |
| Generic class storing elements in an array.                                                 | <code>edu.colorado.collections.ArrayBag</code>     | Figure 5.2 on page 271                                                                                                                                          |
| Generic class with a linked list.<br>This version also has a method to provide an iterator. | <code>edu.colorado.collections.LinkedBag</code>    | <a href="http://www.cs.colorado.edu/~main/edu/colorado/collections/LinkedBag.java">http://www.cs.colorado.edu/~main/edu/colorado/collections/LinkedBag.java</a> |



**FIGURE 5.7** Part of the API Documentation for the Collection Interface

From the **Collection** link at <http://download.oracle.com/javase/7/docs/api/index.html>  
**Interface Collection<E>**

| Method      | Summary                                                                                                                                         |
|-------------|-------------------------------------------------------------------------------------------------------------------------------------------------|
| boolean     | <code>add(E o)</code><br>Ensures that this collection contains the specified element.                                                           |
| boolean     | <code>addAll(Collection&lt;? extends E&gt; c)</code><br>Adds all of the elements in the specified collection to this collection.                |
| void        | <code>clear()</code><br>Removes all of the elements from this collection.                                                                       |
| boolean     | <code>contains(E o)</code><br>Returns true if this collection contains the specified element.                                                   |
| boolean     | <code>containsAll(Collection&lt;?&gt; c)</code><br>Returns true if this collection contains all of the elements in the specified collection.    |
| boolean     | <code>equals(E o)</code><br>Compares the specified object with this collection for equality.                                                    |
| int         | <code>hashCode()</code><br>Returns the hash code value for this collection.                                                                     |
| boolean     | <code>isEmpty()</code><br>Returns true if this collection contains no elements.                                                                 |
| Iterator<E> | <code>iterator()</code><br>Returns an iterator over the elements in this collection.                                                            |
| boolean     | <code>remove(E o)</code><br>Removes a single instance of the specified element from this collection, if it is present.                          |
| boolean     | <code>removeAll(Collection&lt;?&gt; c)</code><br>Removes all of this collection's elements that are also contained in the specified collection. |
| boolean     | <code>retainAll(Collection&lt;?&gt; c)</code><br>Retains only the elements in this collection that are contained in the specified collection.   |
| int         | <code>size()</code><br>Returns the number of elements in this collection.                                                                       |
| Object[ ]   | <code>toArray()</code><br>Returns an array containing all of the elements in this collection.                                                   |

**302 Chapter 5 / Generic Programming**

### The TreeMap Class

The generic `TreeMap` class has two generic type parameters: `K` for the keys and `V` for the type of the values. However, Java's `java.util.TreeMap` class (which implements the `Map` interface) has one extra requirement: The keys must come from a class that implements the `Comparable<K>` interface (from page 292). This allows a `TreeMap` to activate `k1.compareTo(k2)` to compare two keys `k1` and `k2`.

#### TreeMap from java.util

Java's `TreeMap<K,V>` implements the `Map` interface in an efficient way in which the keys are required to come from a class (such as `String`) that implements the `Comparable` interface.

The most important `TreeMap` operations are specified in Figure 5.8.

To illustrate the `TreeMap` operations, we will write a program that uses a `TreeMap` to keep track of the number of times various words appear in a text file. We will use the words as the keys, and the value for each word will be an integer. For example, if the word "starship" appears in the text file 42 times, then the key/value pair of "starship"/42 will be stored in the map. With this in mind, we will write some examples of `TreeMap` operations using these three variables:

```
TreeMap<String, Integer> frequencyData;  
String word; // A word from our text file, to be used as a key  
Integer count; // The number of times that the word appeared  
// in our text file, to be used as a value in the TreeMap
```

Notice that keys will be strings, which allows us to use a `TreeMap` since the Java `String` class does implement the `Comparable<String>` interface. Also, the variable `count` is an `Integer` rather than a simple `int`. This is because the values in a map must be objects rather than primitive values.

**FIGURE 5.8**

Partial Specification for the `TreeMap` Class, Which Implements the `Map` Interface

### Generic Class `TreeMap<K,V>`

#### ❖ **public class TreeMap from the package java.util**

A `TreeMap<K,V>` implements Java's `Map` interface for a collection of key/value pairs. The keys (of type `K`) in a `TreeMap` are required to implement the `Comparable<K>` interface so that for any two keys `x` and `y`, the return value of `x.compareTo(y)` is an integer value that is:

- negative if `x` is less than `y`
- zero if `x` and `y` are equal
- positive if `x` is greater than `y`

(continued)

(FIGURE 5.8 continued)

Partial Specification (see the API documentation for complete specification)

◆ **Constructor for the TreeMap<K,V> (see the API documentation for more constructors)**

public TreeMap()

Initialize a TreeMap with no keys and values.

◆ **clear**

public void clear()

Remove all keys and values from this TreeMap.

**Postcondition:**

This TreeMap is now empty.

◆ **containsKey**

public boolean containsKey(K key)

Determine whether the TreeMap has a particular key.

**Parameter:**

key – the key to be searched for

**Precondition:**

The key can be compared to other keys in the TreeMap using the comparison operation.

**Returns:**

The return value is true if this TreeMap has a key of the specified value; otherwise, it's false.

**Postcondition:**

This TreeMap is now empty.

**Throws:** ClassCastException or NullPointerException

Indicates that the specified key cannot be compared to other keys currently in the TreeMap.

(The NullPointerException means that the specified key is null, and the comparison operation does not permit null.)

◆ **get**

public V get(K key)

Gets the value that is currently associated with the specified key.

**Parameter:**

key – the key whose associated value is to be returned

**Precondition:**

The key can be compared to other keys in the TreeMap using the comparison operation.

**Returns:**

The value for the specified key within this TreeMap; if there is no such value, then the return value is null.

**Throws:** ClassCastException or NullPointerException

Indicates that the specified key cannot be compared to other keys currently in the TreeMap.

(The NullPointerException means that the specified key is null, and the comparison operation does not permit null.)

(continued)

**304 Chapter 5 / Generic Programming**

(FIGURE 5.8 continued)

**◆ keySet**

```
public Set<K> keySet()
```

Obtain a Set that contains all the current keys of this TreeMap.

**Returns:**

The return value is a Java Set from the class `java.util.Set`. This Set is a container that contains all of the keys currently in this TreeMap.

**Note:**

Format for a loop that steps through every key in a TreeMap `t` (assuming the keys are strings):

```
String key;
while (K nextKey : t.keySet())
{
    ...process the next key, which is stored in nextKey...
}
```

**◆ put**

```
public V put(K key, V value)
```

Put a new key and its associated value into this TreeMap.

**Parameters:**

`key` and `value` – the key and its associated value to put into this TreeMap

**Precondition:**

The key can be compared to other keys in the TreeMap using the comparison operation.

**Postcondition:**

The specified key and its associated value have been inserted into this TreeMap. The return value is the value that was previously associated with the specified key (or null if there was no such key previously in the TreeMap).

**Throws: ClassCastException or NullPointerException**

Indicates that the specified key cannot be compared to other keys currently in the TreeMap.

(The `NullPointerException` means that the specified key is null, and the comparison operation does not permit null.)

**Note:**

The return value does not need to be used. For example, `t.put(k, v)` can be a statement on its own.

**◆ size**

```
public int size()
```

Obtain the number of key/value pairs currently in this TreeMap.

**Returns:**

The number of key/value pairs currently in this TreeMap.

Here are the common tasks we'll need to do with our TreeMap:

**1. Putting a Key/Value Pair into a TreeMap.** A key and its associated value are put into a TreeMap with the put method. For our example, we will read an English word into the variable `word` and compute the `count` of how many times the word occurs. Then we can put the `word` and its `count` into the `frequencyData` TreeMap with the statement:

```
frequencyData.put(word, count);
```

This adds a new key (`word`) with its value (`count`) to the `frequencyData`. If the `word` was already present in the TreeMap, then its old value is replaced by the new `count`.

**2. Checking Whether a Specified Key Is Already in a TreeMap.** The boolean method `containsKey` is used for this task. For example, the expression `frequencyData.containsKey(word)` will be true if the map already has a key that is equal to `word`.

**3. Retrieving the Value That Is Associated with a Specified Key.** The `get` method retrieves a value for a specified key. For example, the return value of `frequencyData.get(word)` is the `Integer` value associated with the key `word`. For our program, this return value is a Java `Integer` object.

**4. Stepping Through All the Keys of a TreeMap.** For any TreeMap, we can use the enhanced form of the for-loop to step through all the different keys currently in the map. The pattern for doing this uses the `keySet` method, as shown here for our word counting program:

```
for(String word : wordMap.keySet( ))  
{  
    ... do processing for this key, which is in the variable word ...
```

This programming pattern works because the return value of `keySet` is a collection class that implements the `Iterable` interface.

## The Word Counting Program

Using a TreeMap and the four operations we have just described, we can write a small program that counts the number of occurrences of every word in a text file. The program we write will just read the words (which are expected to be separated by spaces) and then print a table of this sort:

**306 Chapter 5 / Generic Programming**

| Occurrences | Word     |
|-------------|----------|
| 2           | aardvark |
| 10          | dog      |
| 1           | not      |
| 1           | shower   |

In this example, the file contained four different words (“aardvark,” “dog,” “not,” and “shower”). The word “aardvark” appeared twice, “dog” appeared 10 times, and the other two words appeared once each.

One of the key tasks in our program is to open the input file (which will be called `words.txt`) and read all the words in the file, compute the correct counts as we go, and store these counts in a `TreeMap` called `frequencyData`. The pseudocode for this task follows these steps:

A. *Open the `words.txt` file for reading. We will use a `Scanner` object to do this (see Appendix B).*

B. *while there is still input in the file*  
{  
    *word = the next word (read from the file)*  
  
    *Get the current count (from `frequencyData`) of how many times the word has appeared in the file.*  
  
    *Add one to that current count and store the result back in the count variable.*  
  
    *`frequencyData.put(word, count);`*  
}

The implementation of this pseudocode is given in the `readWordFile` method of Figure 5.9, along with the implementations of three other methods for the application. The `getCount` method is needed to get the current count (from `frequencyData`) of a word. In addition to getting the count, it converts from an `Integer` to an ordinary `int`. The `printAllCounts` method is particularly interesting because it uses an enhanced for-loop, as discussed in Section 5.6.

**Self-Test Exercises for Section 5.7**

34. Write a Java statement that will put a key `k` into a `TreeMap` `t` with an associated value `v`. What will this statement do if `t` already has the key `k`?
35. Write an expression that will be true if a `TreeMap` `t` has a specific key `k`.

36. Suppose that a TreeMap *t* has a key *k* and that the value associated with *k* is an array of double numbers. Write a Java statement that will retrieve the value associated with *k* and assign it to an array variable called *v*.
37. Suppose *t* is a TreeMap with keys that are strings. Write a few Java statements that will print a list of all the keys in *t*, one per line.

**FIGURE 5.9** Implementation of the WordCounter Program to Illustrate the Use of a TreeMap

### Java Application Program

```
// File: WordCounter.java
// Program from Section 5.7 to illustrate the use of TreeMaps and Iterators.
// The program opens and reads a file called words.txt.
// Each line in this file should consist of one or more English words separated by spaces.
// The end of each line must not have any extra spaces after the last word.
// The program reads the file, and then a table is printed of all words and their counts.

import java.util.*; // Provides TreeMap, Iterator, and Scanner
import java.io.*; // Provides FileReader and FileNotFoundException

public class WordCounter
{
    private static void main(String[ ] args)
    {
        TreeMap<String, Integer> frequencyData =
            new TreeMap<String, Integer>();

        readWordFile(frequencyData);
        printAllCounts(frequencyData);
    }

    private static int getCount
    (String word, TreeMap<String, Integer> frequencyData)
    {
        if (frequencyData.containsKey(word))
        { // The word has occurred before, so get its count from the map.
            return frequencyData.get(word); // Auto-unboxed
        }
        else
        { // No occurrences of this word
            return 0;
        }
    }
}
```

(continued)

**308 Chapter 5 / Generic Programming**

(FIGURE 5.9 continued)

```
private static void printAllCounts
(TreeMap<String, Integer> frequencyData)
{
    System.out.println("-----");
    System.out.println("    Occurrences    Word");

    for(String word : frequencyData.keySet( ))
    {
        System.out.printf("%15d    %s\n", frequencyData.get(word), word);
    }

    System.out.println("-----");
}

private static void readWordFile
(TreeMap<String, Integer> frequencyData)
{
    Scanner wordFile;
    String word;      // A word read from the file
    Integer count;   // The number of occurrences of the word

    try
    { // Try to open the words.txt file:
        wordFile = new Scanner(new FileReader("words.txt"));
    }
    catch (FileNotFoundException e)
    { // If the file failed, then print an error message and return without counting words:
        System.err.println(e);
        return;
    }

    while (wordFile.hasNext( ))
    {
        // Read the next word and get rid of the end-of-line marker if needed:
        word = wordFile.next( );

        // Get the current count of this word, add 1, and then store the new count:
        count = getCount(word, frequencyData) + 1; // Autobox
        frequencyData.put(word, count);
    }
}
```

---

## CHAPTER SUMMARY

- A Java variable can be one of the eight primitive data types. Anything that's not one of the eight primitive types is a reference to a Java Object.
- An assignment `x = y` is a **widening conversion** if the data type of `x` is capable of referring to a wider variety of things than the type of `y`. It is a **narrowing conversion** if the data type of `x` is capable of referring to a smaller variety of things than the type of `y`. Java always permits widening conversions, but narrowing conversions require a typecast.
- A **wrapper class** is a class in which each object holds a primitive value. Java provides wrapper classes for each of the eight primitive types. In many situations, Java will carry out automatic conversions from a primitive value to a wrapper object (**autoboxing**) or vice versa (**auto-unboxing**).
- A **generic method** is similar to an ordinary method with one important difference: The definition of a generic method can depend on an underlying data type. The underlying data type is given a name, such as `T`, but `T` is not pinned down to a specific type anywhere in the method's implementation.
- When a class depends on an underlying data type, the class can be implemented as a **generic class**. Converting a collection class to a generic class that holds objects is usually a small task. For example, we converted the `IntArrayBag` to an `ArrayBag` by following the steps on page 269.
- An interface provides a list of methods for a class to implement. By writing a class that implements one of the standard Java interfaces, you make it easier for other programmers to use your class. There may also be existing programs already written that work with some of the standard interfaces.
- Java's `Iterator<E>` generic interface provides an easy way to step through all the elements of a collection class. A class that implements Java's `Iterator` interface must provide two methods:

```
public boolean hasNext( )
public E next( )
```

An `Iterator` must also have a `remove` method, although if removal is not supported, then the `remove` method can simply throw an exception.

- Two classes in this chapter have wide applicability, and you'll find them useful in the future: (1) the `Node` class from Section 5.4, which is a node from a linked list of objects; and (2) the `LinkedBag` class from Section 5.6, which includes a method to generate an `Iterator` for its elements.
- Java provides several different standard collection classes that implement the `Collection` interface (such as `Vector`) and the `Map` interface (such as `TreeMap`).



## Solutions to Self-Test Exercises

1. Yes, yes, yes.
2. This code has both a widening conversion (marked with the circle) and a narrowing conversion (marked with a triangle):

```
String s = new String("Liberty!");
Object obj;
obj = s; ●
s = (String) obj; △
```
3. Here is one example that causes a `ClassCastException` at run time:

```
String s = new String("Liberty!");
Integer i;
Object obj;
obj = s;
i = (Integer) obj;
```
4. Character example

```
= new Character('w');
```
5. Advantage: When a primitive value is placed in a wrapper object, it can be treated just like any other Java object. Disadvantage: A wrapper object can no longer use the primitive operations, such as the arithmetic operations.
6. First, `x` and `y` are unboxed. Then the double numbers are added, resulting in a double answer. This answer is boxed and assigned to `z`.
7. A generic method is a type-safe way to write a single method that can be used with a variety of different types of parameters.
8. The compiler can discover more type errors with a generic method than with an object method.
9. The generic type parameter first appears in angle brackets before the return type of the generic method. It may later appear within the return type, the parameter list, or the implementation of the generic method. Almost all situations require it to appear at least once in the method's parameter list.
10. Create a new object of that type, or create an array with that type of element.
11. 

```
public static <E> int
count(E[] data, E target)
{
    int answer = 0;
    if (target == null)
    {
        for (E next : data)
        {
            if (next == null)
                answer++;
        }
    }
    else
    {
        for (E next : data)
        {
            if (target.equals(next))
                answer++;
        }
    }
    return answer;
}
```
12. We use the `count` method from the previous exercise:

```
static <S,T> bool most(
    S[] sa, S starget,
    T[] ta, T tttarget
)
{
    return count(sa, starget)
        > count(ta, tttarget);
}
```
13. No. The only conversions from `int` to `E` are the places where the data type refers to an element in the bag. For example, the return value from `size` remains an `int`.
14. Yes, we allow `add(null)`, but we also need special code in `countOccurrences` and `remove` to handle the null reference.

15. Here is the code:
- ```
ArrayBag numbers;
int i;
numbers = new ArrayBag();
for (i = 1; i <= 10; i++)
    numbers.add(new Integer(i));
```
16. In the new `countOccurrences`, we use this for non-null targets:  
`target.equals(data[index])`
17. It used `stdin.stringInput`, where `stdin` is an `EasyReader` from Appendix B.
18. A reference to the `moving` object was put in the bag. When the `moving` object changes its position, the original location is no longer in the bag. So, the first `countOccurrences` prints 0, and the second prints 1.
19. To handle searching for a null target.
20. Suppose the data in the first node was equal to the data in the end node. Then the boolean expression will be false right at the start, and only one node will be copied.
21. The expression `(x == y)` is true if `x` and `y` refer to exactly the same node. The expression `(x.data.equals(y.data))` is true if the data in the two nodes is the same.
22. The methods are `charAt`, `length`, `subSequence`, and `toString`.
23. `public static <T> int countNull(Node<T> head)`
- ```
{
    int answer = 0;
    Lister<T> it =
        new Lister<T>(head);
    while (it.hasNext())
    {
        if (it.next() == null)
            count++;
    }
    return count;
}
```
24. `public static <T> T find(Node<T> head, Comparable x)`
- ```
{
    Lister<T> it =
        new Lister<T>(head);
    T d;
    while (it.hasNext())
    {
        d = it.next();
        if (x.compareTo(d) == 0)
            return d;
    }
    return null;
}
```
25. `public Lister(Node<T> head)`
- ```
{
    current = Node<T>.listCopy(head);
}
```
26. `(x instanceof Comparable<Integer>)`
27. None. The only purpose of the `Cloneable` interface is to allow a method to check whether an object is an instance of `Cloneable`.
28. `import java.util.Iterator;`
- ```
public class ArrayIterator<E>
implements Iterator<E>
{
    private E[ ] array;
    private int index;

    public ArrayIterator(E[ ] things)
    {
        array = things;
        index = 0;
    }
    public boolean hasNext()
    {
        return (index < array.length);
    }
    public E next()
    {
        return array[index++];
    }
    ... See page 290 for remove ...
}
```

### 312 Chapter 5 / Generic Programming

29. The elements in the new bag are Java objects rather than integers. Also, the new bag has an `iterator` method to return a `Lister`.
30. The bag from this section stores its elements in a linked list rather than in an array. Also, the new bag has an `iterator` method to return a `Lister`.
31. This uses three import statements:

```
import java.util.Scanner;
import
edu.colorado.collections.LinkedBag;
import edu.colorado.nodes.Lister;
```

The code is:

```
Scanner stdin =
    new Scanner(System.in);
LinkedBag<String> b =
    new LinkedBag<String>();
Lister<String> list;
String s;
int i;
for (i = 1; i <= 10; i++)
{
    System.out.print("Next: ");
    s = stdin.next();
    b.add(s);
}
```

```
list = b.iterator();
while (list.hasNext())
{
    s = list.next();
    System.out.println(s);
}
```

32. An internal iterator is quick to implement and use, but an external iterator provides more flexibility, such as the ability to have two or more iterators active at once.
33. Any collection that implements the `Iterable` interface.
34. `t.put(k,v)`; If `k` is already a key in `t`, then the `put` method will replace the old value with the new value `v`, and the return value of `put` will be the old value.
35. `t.containsKey(k)`
36. `v = (double [ ]) t.get(k);`
37. `Iterator it = t.keySet().iterator;`  
`while (it.hasNext())`  
`{`  
 `System.out.println(it.next());`  
`}`



## PROGRAMMING PROJECTS

**1** Implement a generic class for a sequence of Java objects. You can store the objects in an array (as in Section 3.3) or in a linked list (as in Section 4.5). The class should also implement the `Iterable` interface.

**2** Write a program that uses a bag of strings to keep track of a list of chores you have to accomplish today. The user of the program can request several services: (1) Add an item to the list of chores; (2) ask how many chores are in the list; (3) print the list of chores to the screen; (4)

delete an item from the list; (5) exit the program.

If you know how to read and write strings from a file, then have the program obtain its initial list of chores from a file. When the program ends, it should write all unfinished chores back to this file.

**3** For this project, you will use the bag class from Section 5.6, including the `grab` method that returns a randomly selected element. Use this ADT in a program that does the following:



**314** Chapter 5 / Generic Programming

cannot call `data[i].clone()` directly to make a copy (since the `clone` method is not public for the `Object` class). The solution to this problem uses two classes, `java.lang.Class` and `java.lang.reflect.Method`, as shown in Figure 5.10.

- 6** Modify the bag from the previous exercise so that all of the add methods attempt to make a clone of any item that is added to the bag.

These clones are then put in the bag (rather than just putting a reference to the original item into the bag). Because you don't know whether the type of items in the bag have a public `clone` method, you'll need to attempt to clone in a manner that is similar to Figure 5.10.

- 7** Rewrite the word counting program from Section 5.7 so that the user can specify the name of the input file. Also, modify the program so that all non-letters are removed from each word as it is read, and all uppercase letters are converted to the corresponding lowercase letter.

- 8** Write an interactive program that uses a `TreeMap` to store the names and phone numbers of your friends.

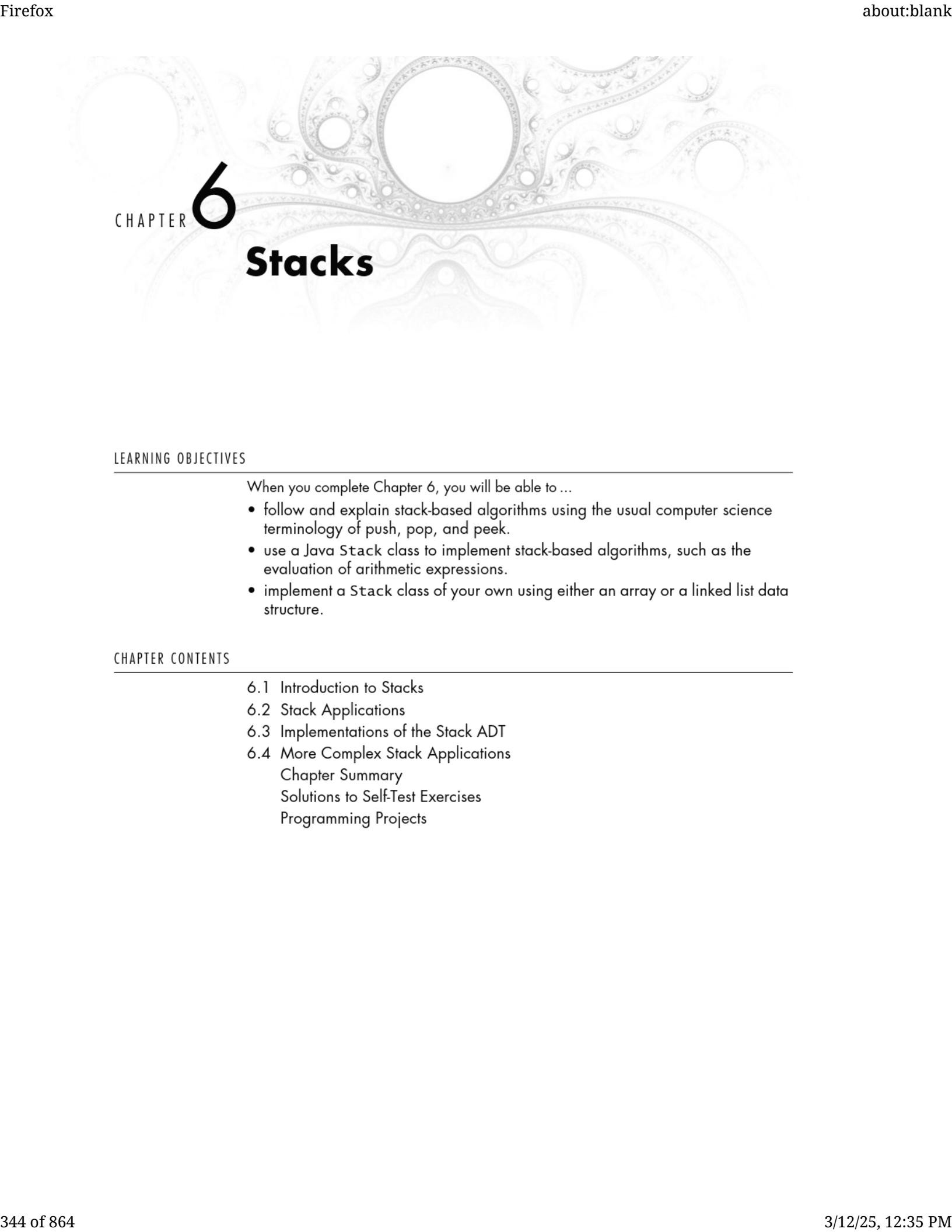
- 9** Write a group of generic static methods for examining and manipulating a collection of items via the collection's iterator.

For example, one of the methods might have this heading:

```
public static <E> E find(  
    Iterator<E> it,  
    E target  
>;
```

When the `find` method is activated, the iterator, `it`, is already in some collection. The method searches the collection via the iterator. If one of the collection's elements is equal to the target (using the `equals` method), then a reference to that element is returned; if the target is never found in the range, then the null reference is returned (and the iterator is left at a position that is just after the target).

Discuss and design other methods for your toolkit that manipulate a collection. As a starting point, please use the linked-list methods from the previous chapter.



# CHAPTER 6

# Stacks

---

## LEARNING OBJECTIVES

When you complete Chapter 6, you will be able to ...

- follow and explain stack-based algorithms using the usual computer science terminology of push, pop, and peek.
- use a Java Stack class to implement stack-based algorithms, such as the evaluation of arithmetic expressions.
- implement a Stack class of your own using either an array or a linked list data structure.

---

## CHAPTER CONTENTS

- 6.1 Introduction to Stacks
- 6.2 Stack Applications
- 6.3 Implementations of the Stack ADT
- 6.4 More Complex Stack Applications
  - Chapter Summary
  - Solutions to Self-Test Exercises
  - Programming Projects

CHAPTER

## 6

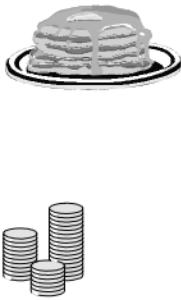
## Stacks

*The pushdown store is a “first in–last out” list. That is, symbols may be entered or removed only at the top of the list.*

JOHN E. HOPCROFT AND JEFFREY D. ULLMAN  
*Formal Languages and Their Relation to Automata*

This chapter introduces a data structure known as a *stack* or, as it is sometimes called, a *pushdown store*. It is a simple structure, even simpler than a linked list. Yet it turns out to be one of the most useful data structures known to computer science.

## 6.1 INTRODUCTION TO STACKS



The drawings in the margin depict some stacks. There is a stack of pancakes, some stacks of coins, and a stack of books. A *stack* is an ordered collection of items that can be accessed only at one end. That may not sound like what you see in these drawings, but think for a moment. Each of the stacks is ordered from top to bottom; you can identify any item in a stack by saying it is the top item, second from the top, third from the top, and so on. Unless you mess up one of the neat stacks, you can access only the top item. To remove the bottom book from the stack, you must first remove the two books on top of it. The abstract definition of a stack reflects this intuition.



### Stack Definition

A **stack** is a data structure of ordered items such that items can be inserted and removed only at one end (called the **top**).

When we say that the items in a stack are *ordered*, all we mean is that there is one that can be accessed first (the one on top), one that can be accessed second (just below the top), a third one, and so forth. We do *not* require that the items can be compared using the < operator. The items can be of any type.

Stack items must be removed in the reverse order of that in which they are placed on the stack. For example, you can create a stack of books by first placing a dictionary, placing a thesaurus on top of the dictionary, and placing a novel on top of those so that the stack has the novel on top. When the books are removed, the novel must come off first (since it is on top), then the thesaurus, and finally the dictionary. Because of this property, a stack is called a “last-in/first-out” data structure (abbreviated LIFO).

Of course, a stack that is used in a program stores information rather than physical items such as books or pancakes. Therefore, it may help to visualize a stack as a pile of papers on which information is written. To place some information on the stack, you write the information on a new sheet of paper and place this sheet of paper on top of the stack. Getting information out of the stack is also accomplished by a simple operation since the top sheet of paper can be removed and read. There is just one restriction: Only the top sheet of paper is accessible. To read the third sheet from the top, for example, the top two sheets must be removed from the stack.

A stack is analogous to a mechanism that is used in a popular candy holder called a *Pez® dispenser*, shown in the margin. The dispenser stores candy in a slot underneath an animal head figurine. Candy is loaded into the dispenser by pushing each piece into the hole. There is a spring under the candy with the tension adjusted so that when the animal head is tipped backward, one piece of candy pops out. If this sort of mechanism were used as a stack data structure, the data would be written on the candy (which may violate some health laws, but it still makes a good analogy). Using this analogy, you can understand why adding an item to a stack is called a **push** operation and removing an item from a stack is called a **pop** operation.

LIFO
A stack is a last-in/first-out data structure. Items are taken out of the stack in the reverse order of their insertion.



Pushing



Popping

### The Stack Class—Specification

The key methods of a stack class are specified in Figure 6.1. Our specification lists a stack constructor and five methods. The most important methods are **push** (to add an item at the top of the stack) and **pop** (to remove the top item). Another method, called **peek**, allows a programmer to examine the top item without actually removing it. There are no methods that allow a program to access items other than the top. To access any item other than the top one, the program must remove items one at a time from the top until the desired item is reached.

**FIGURE 6.1** Specification of the Key Methods of a Generic Stack Class

#### Specification

These are the key methods of a stack class, which we will implement in several different ways. Although this is a specification for a generic stack, we could also implement a stack that contains primitive values (such as `int`) directly. The Java Class Libraries also provide a generic stack class called `java.util.Stack`, which has these same key methods.

##### ◆ Constructor for the Generic Stack<E>

```
public Stack( )
```

Initialize an empty stack.

##### Postcondition:

This stack is empty.

(continued)

**318 Chapter 6 / Stacks***(FIGURE 6.1 continued)***◆ isEmpty**

```
public boolean isEmpty( )
```

Determine whether this stack is empty.

**Returns:**

true if this stack is empty; otherwise, false

**◆ peek**

```
public E peek( )
```

Get the top item of this stack without removing the item.

**Precondition:**

This stack is not empty.

**Returns:**

the top item of the stack

**Throws: EmptyStackException**

Indicates that this stack is empty.

**◆ pop**

```
public E pop( )
```

Get the top item, removing it from this stack.

**Precondition:**

This stack is not empty.

**Postcondition:**

The return value is the top item of this stack, and the item has been removed.

**Throws: EmptyStackException**

Indicates that this stack is empty.

**◆ push**

```
public void push(E item)
```

Push a new item onto this stack. The new item may be the null reference.

**Parameter:**

item – the item to be pushed onto this stack

**Postcondition:**

The item has been pushed onto this stack.

**Throws: OutOfMemoryException**

Indicates insufficient memory for pushing a new item onto this stack.

**◆ size**

```
public int size( )
```

Accessor method to determine the number of items in this stack.

**Returns:**

the number of items in this stack

If a program attempts to pop an item off an empty stack, it is asking for the impossible; this error is called **stack underflow**. The pop method indicates a stack underflow by throwing an `EmptyStackException`. This exception is defined in `java.util.EmptyStackException`. To help you avoid a stack underflow, the class provides a method to determine whether a stack is empty. There is also a method to obtain the stack's current size.

### We Will Implement a Generic Stack

Later we will implement the stack in several different ways. Some of our implementations will have extra methods beyond the five given in the figure. Also, each of our implementations will be a *generic* stack that depends on an unspecified data type for the stack's elements. We'll call the class `Stack`, but just like any generic class, it can be used only with Java objects. For example, we can't put primitive `int` values into our `Stack` without a boxing conversion. Because of this, it might sometimes be useful to have a stack that contains primitive values directly. Although we won't show those simpler stacks in this chapter, you can find source code for them in this book's online resources at <http://www.cs.colorado.edu/~main/dsoj.html>.

The Java Class Libraries also provide a stack of objects called `java.util.Stack`, with the same five methods from Figure 6.1.

*The stacks implemented in this chapter are generic stacks*

### PROGRAMMING EXAMPLE: Reversing a Word

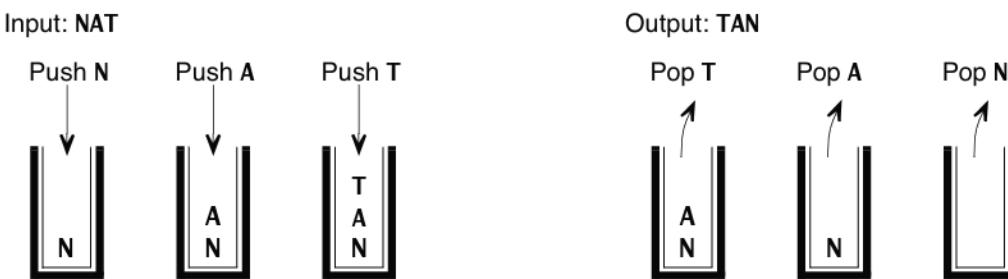
Stacks are very intuitive—even cute—but are they good for anything? Surprisingly, they have many applications. Most compilers use stacks to analyze the syntax of a program. Stacks are used to keep track of local variables when a program is run. Stacks can be used to search a maze or a family tree or other types of branching structures. In this book, we will discuss examples related to each of these applications. But before we present any complicated applications of the stack ADT, let us first practice with a simple problem so that we can see how a stack is used.

*uses for stacks*

Suppose you want a program to read in a word and then write it out backward. If the program reads in NAT, then it will output TAN. If it reads in TAPS, it will output SPAT. The author Roald Dahl wrote a book called ESIOTROT, which our program converts to TORTOISE. One way to accomplish this task is to read the input one letter at a time and place each letter in a stack of characters. After the word is read, the letters in the stack are written out, but because of the way a stack works, they are written out in reverse order. The outline is shown here:

```
// Reversing the spelling of a word
Declare a stack of characters.

while (there are more characters of the word to read)
    Read a character and push the character onto the stack.
while (the stack is not empty)
    Pop a character off the stack and write that character to the screen.
```

**FIGURE 6.2** Using a Stack to Reverse Spelling

This computation is illustrated in Figure 6.2. At all times in the computation, the only available item is on “top.” Figure 6.2 suggests another intuition for thinking about a stack. You can view a stack as a hole in the ground and view the items as being placed in the hole one on top of the other. To retrieve an item, you must first remove the items on top of it.

### Self-Test Exercises for Section 6.1

1. Suppose a program uses a stack of characters to read in a word and then write the word out backward as described in this section. Now suppose the input word is DAHL. List all the activations of the push and pop methods. List them in the order in which they will be executed, and indicate the character that is pushed or popped. What is the output?
2. Consider the stack class given in Figure 6.1 on page 317. The `peek` method lets you look at the top item in the stack without changing the stack. Describe how you can define a new method that returns the second item from the top of the stack without permanently changing the stack. (If you temporarily change the stack, then change it back before the method ends.) Your description will be in terms of `peek`, `pop`, and `push`. Give your solution in pseudocode, not in Java.

## 6.2 STACK APPLICATIONS

As an exercise to learn about data structures, we will eventually implement the stack class ourselves. For now, though, we’ll look at some example applications that use Java’s generic stack from `java.util.Stack`. Our first example uses a stack that contains characters to analyze a string of parentheses.

### PROGRAMMING EXAMPLE: Balanced Parentheses

Later in this chapter, we will describe how stacks can be used to evaluate arithmetic expressions. At the moment, we will describe a simpler method called `isBalanced` that does a closely related task. The algorithm checks an expres-

sion to see when the parentheses match correctly. It allows three kinds of parentheses: ( ), [ ], or { }. Any symbol other than one of these parentheses is ignored.

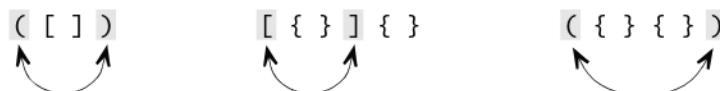
For example, consider the string " $\{[X + Y^*(Z + 7)]^*(A + B)\}$ ". Each of the left parentheses has a corresponding right parenthesis. Also, as the string is read from left to right, there is never an occurrence of a right parenthesis that cannot be matched with a corresponding left parenthesis. Therefore, the activation of `isBalanced("{{X + Y*(Z + 7)}*(A + B)}")` returns `true`.

On the other hand, consider the string " $((X + Y^*{Z + 7})^*[A + B])$ ". The parentheses around the subexpression  $Z + 7$  match each other, as do the parentheses around  $A + B$ . And one of the left parentheses in the expression matches the final right parenthesis. But the other left parenthesis has no matching right parenthesis. Hence, `isBalanced("((X + Y^*{Z + 7})^*[A + B])")` returns `false`.

The technique used is simple: The algorithm scans the characters of the string from left to right. Every time a left parenthesis occurs, it is pushed onto the stack. Every time a right parenthesis occurs, a matching left parenthesis is popped off the stack. If the correct kind of left parenthesis is not on top of the stack, then the string is unbalanced. For example, when a curly right parenthesis '}' occurs in the string, the matching curly left parenthesis '{' should be on top of the stack.

All symbols other than parentheses are ignored. If all goes smoothly and the stack is empty at the end of the expression, then the parentheses match. On the other hand, three things might go wrong: (1) The stack is empty when the algorithm needs to pop a symbol; (2) the wrong kind of parenthesis appears at some point; or (3) symbols are still in the stack after all the input has been read. In each of these cases, the parentheses are not balanced.

Let's look at some examples. Since no symbols other than parentheses can affect the results, we will use expressions of just parentheses symbols. All of the following are balanced (shading and arrows help find matching parentheses):



If you think about these examples, you can begin to understand the algorithm. In the first example, the parentheses match because they have the same number of left and right parentheses, but the algorithm does more than just count parentheses. The algorithm actually matches parentheses. Every time it encounters a ')', the symbol it pops off the stack is the matching '('. When it encounters a ']', the symbol it pops off the stack is the matching '['. When it encounters a '}', the symbol it pops off the stack is the matching '{'.

The complete sequences of stack configurations from two executions of the algorithm are shown in Figure 6.3. The stacks shown in the figure show the configuration after processing each character of the expression.

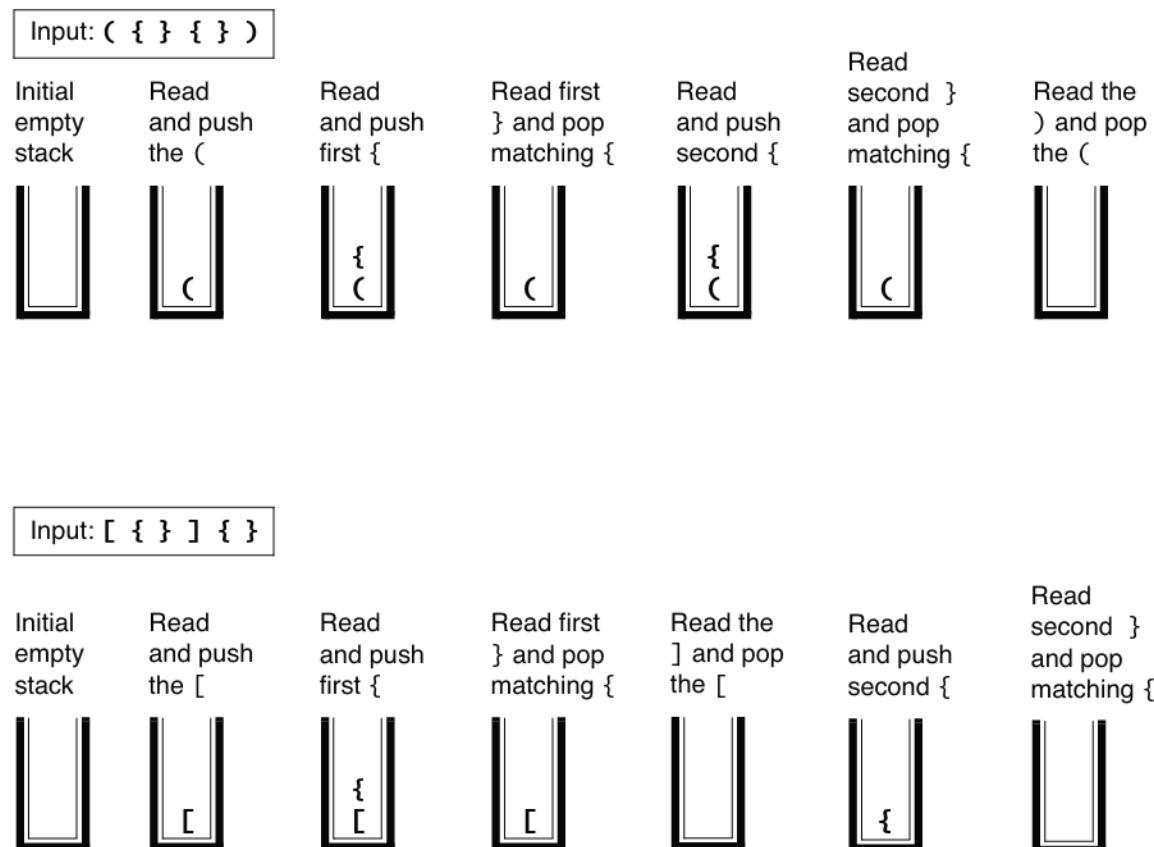
In general, the stack works by keeping a stack of the unmatched left parentheses. Every time the algorithm encounters a right parenthesis, the corresponding

## 322 Chapter 6 / Stacks

left parenthesis is deleted (popped) from the stack. If the parentheses in the input match correctly, things work out perfectly, and the stack is empty at the end of the input line.

The balancing algorithm is implemented by the `isBalanced` method of Figure 6.4. Notice that the method uses a stack of `Character` objects. We can push ordinary `char` values onto the stack (through autoboxing), and we can use the popped values as if they were ordinary `char` values (through auto-unboxing). One technique in the implementation may be new to you: acting on the string's next character via a *switch* statement. We'll discuss this technique after you've looked through the program.

**FIGURE 6.3** Stack Configurations for the Parentheses Balancing Algorithm



**FIGURE 6.4** A Method to Check for Balanced ParenthesesA Method Implementation

```
public static boolean isBalanced(String expression)
// Postcondition: A true return value indicates that the parentheses in the
// given expression are balanced. Otherwise, the return value is false.
// Note that characters other than ( ), { }, and [ ] are ignored.
{
    // Meaningful names for characters
    final char LEFT_NORMAL = '(';
    final char RIGHT_NORMAL = ')';
    final char LEFT_CURLY = '{';
    final char RIGHT_CURLY = '}';
    final char LEFT_SQUARE = '[';
    final char RIGHT_SQUARE = ']';

    Stack<Character> store = new Stack<Character>; // From java.util.Stack
    int i; // An index into the string
    boolean failed = false; // Change to true for a mismatch

    for (i = 0; !failed && (i < expression.length()); i++)
    {
        switch (expression.charAt(i))
        {
            case LEFT_NORMAL:
            case LEFT_CURLY:
            case LEFT_SQUARE:
                store.push(expression.charAt(i));
                break;
            case RIGHT_NORMAL:
                if (store.isEmpty() || (store.pop() != LEFT_NORMAL))
                    failed = true;
                break;
            case RIGHT_CURLY:
                if (store.isEmpty() || (store.pop() != LEFT_CURLY))
                    failed = true;
                break;
            case RIGHT_SQUARE:
                if (store.isEmpty() || (store.pop() != LEFT_SQUARE))
                    failed = true;
                break;
        }
    }
    return (store.isEmpty() && !failed);
}
```

## PROGRAMMING TIP

### THE SWITCH STATEMENT

The for-loop in Figure 6.4 processes character number *i* of the expression during each iteration. There are several possible actions, depending on what kind of character appears at `expression.charAt(i)`. An effective statement to select among many possible actions is the switch statement, with the general form:

```
switch (<Control value>)
{
    <Body of the switch statement>
}
```

When the switch statement is reached, the control value is evaluated. The program then looks through the body of the switch statement for a matching case label. For example, if the control value is the character 'A', then the program looks for a case label of the form `case 'A':`. If a matching case label is found, then the program goes to that label and begins executing statements. Statements are executed one after another, but if a **break** statement (of the form `break;`) occurs, then the program skips to the end of the body of the switch statement.

If the control value has no matching case label, then the program will look for a **default label** of the form `default:`. This label handles any control values that don't have their own case label.

If there is no matching case label and no default label, then the whole body of the switch statement is skipped.

The control value may be an integer, character, short integer, byte, enumerated value, or (starting with Java SE 7) a string value.

For the `isBalanced` method of Figure 6.4, the switch statement has one case label for each of the six possible kinds of parentheses. The three kinds of left parentheses are all handled together by putting their case statements one after another. Each of the right parentheses is handled with its own case statement. For example, one of the right parentheses is the character `RIGHT_NORMAL`, which is an ordinary right parenthesis `')'`. The `RIGHT_NORMAL` character is handled as shown here:

```
switch (expression.charAt(i))
{
    ...
    case RIGHT_NORMAL:
        if (store.isEmpty() || (store.pop() != LEFT_NORMAL))
            failure = true;
        break;
    ...
}
```

## Evaluating Arithmetic Expressions

In this next programming example, we will design and write a calculator program. This will be an example of a program that uses two stacks: a stack of characters and a stack of double numbers.

### Evaluating Arithmetic Expressions—Specification

The program takes as input a fully parenthesized numeric expression such as the following:

$$(((12 + 9)/3) + 7.2)*((6 - 4)/8))$$

*input to the calculator program*

The expression consists of integers or double numbers, together with the operators  $+$ ,  $-$ ,  $*$ , and  $/$ . To focus on the use of the stack (rather than on input details), we require that each input number be non-negative. (Otherwise, it is hard to distinguish the subtraction operator from a minus sign that is part of a negative number.) We will assume that the expression is formed correctly so that each operation has two arguments. Finally, we will also assume that the expression is fully parenthesized with ordinary parentheses ' $($ ' and ' $)$ ', meaning that each operation has a pair of matched parentheses surrounding its arguments. We can later enhance our program so that these assumptions are no longer needed.

The output will simply be the value of the arithmetic expression.

*output of the calculator program*

### Evaluating Arithmetic Expressions—Design

Most of the program's work will be carried out by a method that reads one line of input and evaluates that line as an arithmetic expression. To get a feel for the problem, let's start by doing a simple example by hand. Consider the following expression:

$$(((6 + 9)/3)*(6 - 4))$$

*do an example by hand*

If we were to evaluate this expression by hand, we might first evaluate the innermost expressions,  $(6 + 9)$  and  $(6 - 4)$ , to produce the smaller expression:

$$((15/3)*2)$$

Next, we would evaluate the expression  $(15/3)$  and replace this expression with its value of 5. That would leave us with the expression  $(5 * 2)$ . Finally, we would evaluate this last operation to get the answer of 10.

To convert this intuitive approach into a fully specified algorithm that can be implemented, we need to do things in a more systematic way: We need a specific way to find the expression to be evaluated next and a way to remember the results of our intermediate calculations.

First let's find a systematic way of choosing the next expression to be evaluated. (After that, we can worry about how we will keep track of the intermediate

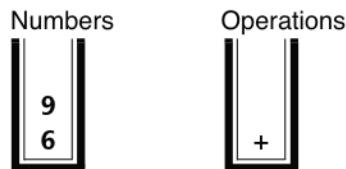


expressions, there is no need to back up; to find the next right parenthesis, we can just keep reading left to right from where we left off. The next right parenthesis will indicate the end of the next expression to be evaluated.

Now we know how to find the expression to be evaluated next, but how do we keep track of our intermediate values? For this, we use two stacks. One stack will contain numbers; there will be numbers from the input as well as numbers that were computed when subexpressions were evaluated. The other stack will hold symbols for the operations that still need to be evaluated. Because a stack processes data in a last-in/first-out manner, it will turn out that the correct two numbers are on the top of the numbers stack at the same time that the appropriate operation is at the top of the stack of operations. To better understand how the process works, let's evaluate our sample expression one more time, this time using the two stacks.

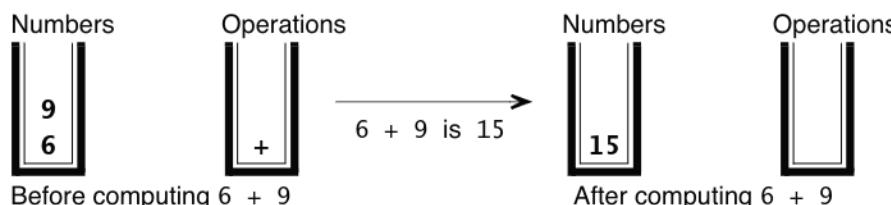
We begin by reading up to the first right parenthesis; the numbers we encounter along the way are pushed onto the numbers stack, and the operations we encounter along the way are pushed onto the operations stack. When we reach the first right parenthesis, our two stacks look like this:

Characters read so far (shaded):  
 $((6 + 9) / 3) * (6 - 4))$



Whenever we reach a right parenthesis, we combine the top two numbers (on the numbers stack) using the topmost operation (on the character stack). In our example, we compute  $6 + 9$ , yielding 15, and this number 15 is pushed back onto the numbers stack:

Characters read so far (shaded):  
 $((6 + 9) / 3) * (6 - 4))$

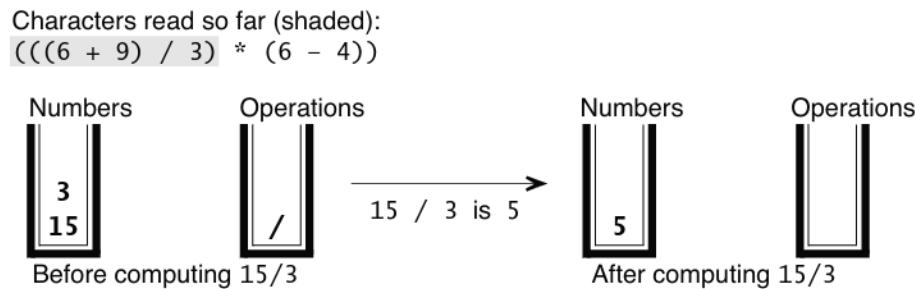


Notice that the leftmost operand (6 in this example) is the *second* number popped off the stack. For addition, this does not matter—who cares whether we have added  $6 + 9$  or  $9 + 6$ ? But the order of the operands does matter for subtraction and division.

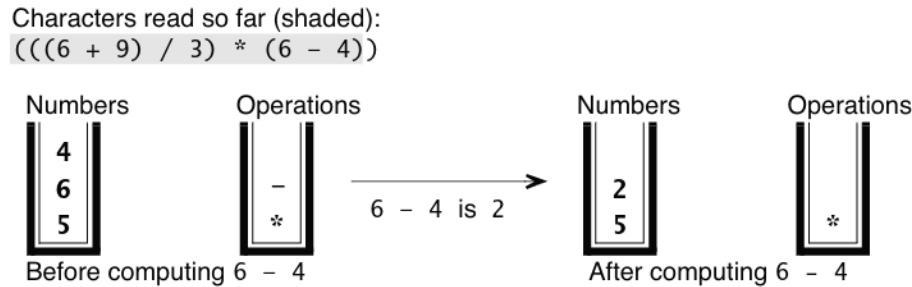
Next, we simply continue the process by reading up to the next right parenthesis, pushing the numbers we encounter onto the numbers stack, and pushing

## 328 Chapter 6 / Stacks

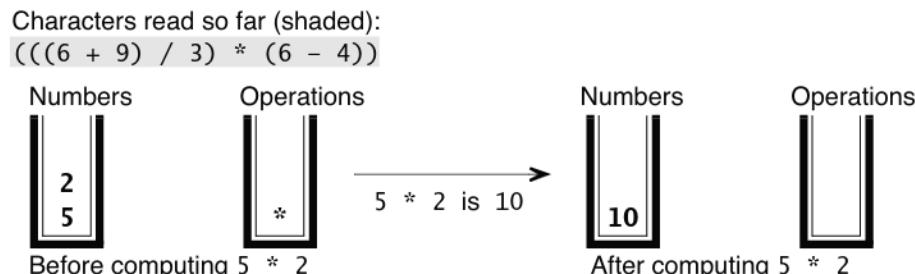
the operations we encounter onto the operations stack. When we reach the next right parenthesis, we combine the top two numbers using the topmost operation. Here's what happens in our example when we reach the second right parenthesis:



Again, the leftmost operand (15) is the second number popped off the stack, so the correct evaluation is  $15/3$ , not  $3/15$ . Continuing the process, we obtain:



Finally, continuing the process one more time does not add anything to the stacks, but it does read the last right parenthesis and does combine two numbers from the numbers stack with an operation from the operations stack:



At this point, there is no more input, and there is exactly one number in the number stack, namely 10. That number is the answer. Notice that when we used the two stacks, we performed the exact same evaluations as we did when we first evaluated this expression in a simple pencil-and-paper fashion.

To evaluate our expression, we only need to repeatedly handle the input items according to the following cases:

*cases for evaluating an arithmetic expression*

**Numbers.** When a number is encountered in the input, the number is read and pushed onto the numbers stack. We allow the numbers to be any double number, but we do not allow a + or - sign at the front of the number. So, we can have numbers such as 42.8 and 0.34, but not -42.8 or -0.34. This restriction is because we haven't yet developed the algorithm for distinguishing between a + or - sign that is part of a number and a + or - sign that is an arithmetic addition or subtraction.

**Operation Characters.** When one of the four operation characters is encountered in the input, the character is read and pushed onto the operations stack.

**Right Parenthesis.** When a right parenthesis is read from the input, an "evaluation step" takes place. The step pops the top two numbers from the number stack and pops the top operation from the operation stack. The two numbers are combined using the operation (with the second number popped as the left operand). The result of the operation is pushed back onto the numbers stack.

**Left Parenthesis or Blank.** The only other characters that appear in the input are left parentheses and blanks. These are read and thrown away, not affecting the computation. A more complete algorithm would need to process the left parentheses in some way to ensure that each left parenthesis is balanced by a right parenthesis, but for now we are assuming that the input is completely parenthesized in a proper manner.

The processing of input items halts when the end of the input line occurs, indicated by '\n' in the input. At this point, the answer is the single number that remains in the number stack.

We now have our algorithm, which we plan to implement as a method called `evaluate`. The parameter to `evaluate` is a string that provides the arithmetic expression. The return value is the value of the arithmetic expression as a double number. For example, `evaluate("(((60 + 40)/50) * (16 - 4))")` returns 24.0.

### Implementation of the `evaluate` Method

Our implementation of `evaluate` requires some understanding of the `Scanner` class from Appendix B. This class is often attached to keyboard input, but in our `evaluate` implementation, we create a `Scanner` called `input` that contains all the characters from the arithmetic expression. This allows us to more easily read the expression, detecting which parts are numbers and which parts are operations. In particular, we use these items:

- The method `input.hasNext()` returns `true` if there are still more parts of the expression to be processed.

**330 Chapter 6 / Stacks**

- The expression `input.hasNext(UNSIGNED_DOUBLE)` is true if the next part of the input expression is a double number (with no + or - sign in front). This uses a constant `UNSIGNED_DOUBLE` that is defined and discussed in Appendix B.
- The statement `next = input.findInLine(UNSIGNED_DOUBLE)` sets the string `next` equal to the next part of the input expression, which must be a double number. Once the double number is in the string `next`, we can convert it to a `Double` value and push it onto the stack with:

```
numbers.push(new Double(next));
```

- Similarly, `next = input.findInLine(CHARACTER)` sets `next` equal to the next single character (skipping spaces) in the input expression.

Our implementation also uses one other method, `evaluateStackTops`, which appears along with the `evaluate` method in Figure 6.5. In order to compile these methods, you will need to include the definitions of `UNSIGNED_DOUBLE` and `CHARACTER` from Appendix B; the code also requires these import statements:

```
import java.util.Stack;      // Provides the generic Stack class
import java.util.Scanner;    // Provides the Scanner class
import java.util.Pattern;    // Provides the Pattern class
```

---

**FIGURE 6.5** A Method to Evaluate a Fully Parenthesized Arithmetic Expression**Method Specification and Implementation****◆ evaluate**

```
public static double evaluate(String expression)
```

The `evaluate` method evaluates the arithmetic expression.

**Parameter:**

`expression`—a fully parenthesized arithmetic expression

**Precondition:**

The expression must be a fully parenthesized arithmetic expression formed from double numbers (with no + or – sign in front), any of the four arithmetic operations (+, –, \*, or /), and spaces.

**Returns:**

the value of the arithmetic expression

**Throws: IllegalArgumentException**

Indicates that the expression had the wrong format.

(continued)

(FIGURE 6.5 continued)

```
public static double evaluate(String expression)
{
    // Two generic stacks to hold the expression's numbers and operations:
    Stack<Double> numbers = new Stack<Double>();
    Stack<Character> operations = new Stack<Character>();

    // Convert the expression to a Scanner for easier processing. The next String holds the
    // next piece of the expression: a number, operation, or parenthesis.
    Scanner input = new Scanner(expression);
    String next;

    while (input.hasNext())
    {
        if (input.hasNext(UNSIGNED_DOUBLE))
        { // The next piece of the expression is a number
            next = input.findInLine(UNSIGNED_DOUBLE);
            numbers.push(new Double(next));
        }
        else
        { // The next piece of the input is an operation (+, -, *, or /) or a parenthesis.
            next = input.findInLine(CHARACTER);
            switch (next.charAt(0))
            {
                case '+': // Addition
                case '-': // Subtraction
                case '*': // Multiplication
                case '/': // Division
                    operations.push(next.charAt(0));
                    break;
                case ')': // Right parenthesis (the evaluateStackTops function is on the next page)
                    evaluateStackTops(numbers, operations);
                    break;
                case '(': // Left parenthesis
                    break;
                default : // Illegal character
                    throw new IllegalArgumentException("Illegal character");
            }
        }
    }
    if (numbers.size() != 1)
        throw new IllegalArgumentException("Illegal input expression");
    return numbers.pop();
}
```

This code requires  
java.util.Stack,  
java.util.Scanner, and  
java.util.Pattern.

See Appendix B for  
UNSIGNED\_DOUBLE and  
CHARACTER, which we use  
here to simplify reading from  
a Scanner.

(continued)

**332 Chapter 6 / Stacks**

(FIGURE 6.5 continued)

**Method Specification and Implementation****◆ evaluateStackTops**

```
public static void evaluateStackTops  
    (Stack<Double> numbers, Stack<Character> operations)
```

This method applies an operation to two numbers taken from the numbers stack.

**Precondition:**

There must be at least two numbers on the numbers stack, and the top character on the operations stack must be the character '+', '-', '\*', or '/'.

**Postcondition:**

The top two numbers have been popped from the numbers stack, and the top operation has been popped from the operations stack. The two numbers have been combined using the operation (with the second number popped as the left operand).

**Throws: IllegalArgumentException**

Indicates that the stacks fail the precondition.

```
public static void evaluateStackTops  
    (Stack<Double> numbers, Stack<Character> operations)  
{  
    double operand1, operand2;  
  
    // Check that the stacks have enough items, and get the two operands.  
    if ((numbers.size( ) < 2) || (operations.isEmpty( )))  
        throw new IllegalArgumentException("Illegal expression");  
    operand2 = numbers.pop( );  
    operand1 = numbers.pop( );  
  
    // Carry out an action based on the operation on the top of the stack.  
    switch (operations.pop( ))  
    {  
        case '+': numbers.push(operand1 + operand2);  
                    break;  
        case '-': numbers.push(operand1 - operand2);  
                    break;  
        case '*': numbers.push(operand1 * operand2);  
                    break;  
        case '/': // Note: A division by zero is possible. The result would be one of the  
                   // constants Double.POSITIVE_INFINITY or Double.NEGATIVE_INFINITY.  
                   numbers.push(operand1 / operand2);  
                   break;  
        default : throw new IllegalArgumentException("Illegal operation");  
    }  
}
```

## Evaluating Arithmetic Expressions—Testing and Analysis

As usual, you should test your program on boundary values that are most likely to cause problems. For this program, one kind of boundary value consists of the simplest kind of expressions: those that combine only two numbers. To test that operations are performed correctly, you should test simple expressions for each of the operations  $+$ ,  $-$ ,  $*$ , and  $/$ . These simple expressions should have only one operation. Be sure to test the division and subtraction operations carefully to ensure that the operations are performed in the correct order. After all,  $3/15$  is not the same as  $15/3$ , and  $3 - 15$  is not the same as  $15 - 3$ . These are perhaps the only boundary values. But it is important also to test some cases with nested parentheses, and you can test an illegal division such as  $15/0$ . What does the program do? It throws an `IllegalArgumentException` in the `evaluateStackTops` method.

Let's estimate the number of operations that our program will use on an expression of length  $n$ . We will count each of the following as one program operation: reading or peeking at a symbol, performing one of the arithmetic operations ( $+$ ,  $-$ ,  $*$ , or  $/$ ), pushing an item onto one of the stacks, and popping an item off of one of the stacks. We consider each kind of operation separately.

*testing*

*time analysis*

**Time Spent Reading Characters.** There are only  $n$  symbols in the input, so the program can read at most  $n$  symbols. No character is “peeked” at more than once either, so this aspect of the program has no more than  $2n$  operations.

**Time Spent Evaluating Arithmetic Operations.** Each arithmetic operation performed by the program is the evaluation of an operation symbol in the input. Because there are no more than  $n$  arithmetic operations in the input, there are at most  $n$  arithmetic operations performed. In actual fact, there are far fewer than  $n$  operations since many of the input symbols are digits or parentheses. But there are certainly no more than  $n$  arithmetic operation symbols, so it is safe to say that there are no more than  $n$  arithmetic operations performed.

**Number of Push Operations.** Since there are no more than  $n$  arithmetic operation symbols, we know that there are at most  $n$  operation symbols pushed onto the operations stack. The numbers stack may contain input numbers and numbers obtained from evaluating arithmetic expressions. Again, an upper bound will suffice: There are at most  $n$  input numbers and at most  $n$  arithmetic operations evaluated. Thus, at most,  $2n$  numbers are pushed onto the numbers stack. This gives an upper bound of  $3n$  total push operations onto the stacks.

**Number of Pop Operations.** Once we know the total number of items that are pushed onto the two stacks, we have a bound on how many things can be popped off of the two stacks. After all, you cannot pop off an item unless it was first pushed onto the stack. Thus, there is an upper bound of  $3n$  pop operations from any of the stacks.

**Total Number of Operations.** Now let's total things up. The total number of operations is no more than  $2n$  reads/peeks, plus  $n$  arithmetic operations performed, plus  $3n$  items pushed onto a stack, plus  $3n$  items popped off of a stack—for a grand total of  $9n$ . The actual number of operations will be less than this because we have used generous upper bounds in several estimates, but  $9n$  is enough to conclude that the algorithm for this program is  $O(n)$ ; this is a linear algorithm in the number of stack operations.

### Evaluating Arithmetic Expressions—Enhancements

The program in Figure 6.5 on page 330 is a fine example of how to use stacks. As a computer scientist, you will find yourself using stacks in this manner in many different situations. However, the program is not a fine example of a finished program. Before we can consider it to be a finished product, we need to add a number of enhancements to make the program more robust and friendly.

Some enhancements are easy. It is useful (and easy) to write a main program that repeatedly gets and evaluates arithmetic expressions. Another nice enhancement would be to permit expressions that are not fully parenthesized and to use the Java precedence rules to decide the order of operations when parentheses are missing. We will discuss topics related to this enhancement in Section 6.4, where (surprise!) we'll see that a stack is useful for this purpose, too.

### Self-Test Exercises for Section 6.2

3. How would you modify the calculator program in Figure 6.5 on page 330 to allow the symbol  $\wedge$  to be used for exponentiation? Describe the changes; do not write out the code.
4. How would you modify the calculator program in Figure 6.5 on page 330 to allow for comments in the calculator input? Comments appear at the end of the expression, starting with a double slash // and continuing to the end of the line. Describe the changes; do not write out the code.
5. Write some illegal expressions that are caught by the calculator program in Figure 6.5 on page 330, resulting in an `IllegalArgumentException`.
6. Write some illegal expressions that are not caught by the calculator program in Figure 6.5 on page 330.
7. Carry out the operations of `evaluate` by hand on the input expression `((60 + 40)/50) * (16 - 4)`. Draw the two stacks after each push or pop.
8. What kind of expressions cause the `evaluation` stacks to grow large?
9. What is the time analysis, in big- $O$  notation, of the `evaluation` method? Explain your reasoning.

## 6.3 IMPLEMENTATIONS OF THE STACK ADT

We will give two implementations of our generic stack class: an implementation using an array and an implementation using a linked list. Each implementation will be for a generic class, but as we discussed earlier, we could just as easily implement the stack to hold one of Java's primitive types.

### Array Implementation of a Stack

Figure 6.6 gives the specification and implementation of a generic `ArrayStack` class that includes all the earlier methods we mentioned (from Figure 6.1 on page 317). The class stores its items in an array, so the class also has extra methods for explicitly dealing with capacity. The class implementation uses two instance variables described here:

#### Invariant of the ArrayStack Class

1. The number of items in the stack is stored in the instance variable `manyItems`.
2. The items in the stack are stored in a partially filled array called `data`, with the bottom of the stack at `data[0]`, the next item at `data[1]`, and so on, to the top of the stack at `data[manyItems-1]`.

In other words, our stack implementation is simply a partially filled array implemented in the usual way: an array and a variable to indicate how much of the array is being used. The stack bottom is at `data[0]`, and the top position is the last array position used. Each method (except the constructor) can assume that the stack is represented in this way when the operation is activated. Each method has the responsibility of ensuring that the stack is still represented in this manner when the method finishes.

The methods that operate on our stack are now straightforward. To initialize the stack, set the instance variable `manyItems` to zero, indicating an empty array and hence an empty stack. The constructor also makes the data array, which (as discussed in Section 5.3) is an array of Java objects.

To add an item to the stack (in the `push` method), we store the new item in `data[manyItems]`, and then we increment `manyItems` by 1. To look at the top item in the stack (the `peek` method), we simply look at the item in array position `data[manyItems-1]`. To remove an item from the stack (in the `pop` method), we decrement `manyItems` and then return the value of `data[manyItems]` (which is the value that was on top prior to the `pop` operation). The methods to test for emptiness and to return the size of the stack work by examining the value of `manyItems`.

*the stack items  
are stored in a  
partially filled  
array*

*implementing  
the stack  
operations*

---

**FIGURE 6.6 Specification and Implementation of the Array Version of the Generic Stack Class****Generic Class ArrayStack****◆ public class ArrayStack<E> from the package edu.colorado.collections**

An ArrayStack<E> is a stack of references to E objects.

**Limitations:**

- (1) The capacity of one of these stacks can change after it's created, but the maximum capacity is limited by the amount of free memory on the machine. The constructors, `clone`, `ensureCapacity`, `push`, and `trimToSize` will result in an `OutOfMemoryError` when free memory is exhausted.
- (2) A stack's capacity cannot exceed the largest integer, 2,147,483,647 (`Integer.MAX_VALUE`). Any attempt to create a larger capacity results in failure due to an arithmetic overflow.

**Specification****◆ Constructor for the ArrayStack<E>**

`public ArrayStack()`

Initialize an empty stack with an initial capacity of 10. Note that the `push` method works efficiently (without needing more memory) until this capacity is reached.

**Postcondition:**

This stack is empty and has an initial capacity of 10.

**Throws: OutOfMemoryError**

Indicates insufficient memory for new `Object[10]`.

**◆ Second Constructor for the ArrayStack<E>**

`public ArrayStack(int initialCapacity)`

Initialize an empty stack with a specified initial capacity. Note that the `push` method works efficiently (without needing more memory) until this capacity is reached.

**Parameter:**

`initialCapacity` – the initial capacity of this stack

**Precondition:**

`initialCapacity` is non-negative.

**Postcondition:**

This stack is empty and has the given initial capacity.

**Throws: IllegalArgumentException**

Indicates that `initialCapacity` is negative.

**Throws: OutOfMemoryError**

Indicates insufficient memory for new `Object[initialCapacity]`.

(continued)

(FIGURE 6.6 continued)

◆ **clone**

```
public ArrayStack<E> clone()
```

Generate a copy of this stack.

**Returns:**

The return value is a copy of this stack. Subsequent changes to the copy will not affect the original, nor vice versa.

**Throws:** OutOfMemoryError

Indicates insufficient memory for creating the clone.

◆ **ensureCapacity**

```
public void ensureCapacity(int minimumCapacity)
```

Change the current capacity of this stack.

**Parameter:**

minimumCapacity – the new capacity for this stack

**Postcondition:**

This stack's capacity has been changed to at least minimumCapacity. If the capacity was already at or greater than minimumCapacity, then the capacity is left unchanged.

**Throws:** OutOfMemoryError

Indicates insufficient memory for new Object[minimumCapacity].

◆ **getCapacity**

```
public int getCapacity()
```

Accessor method to determine the current capacity of this stack. The push method works efficiently (without needing more memory) until this capacity is reached.

**Returns:**

the current capacity of this stack

◆ **isEmpty—peek—pop—push—size**

```
public boolean isEmpty()
```

```
public E peek()
```

```
public E pop()
```

```
public void push(E item)
```

```
public int size()
```

These are the standard stack specifications from Figure 6.1 on page 318.

◆ **trimToSize**

```
public void trimToSize()
```

Reduce the current capacity of this stack to its actual size (i.e., the number of items it contains).

**Postcondition:**

This stack's capacity has been changed to its current size.

**Throws:** OutOfMemoryError

Indicates insufficient memory for altering the capacity.

(continued)



(FIGURE 6.6 continued)

```
@SuppressWarnings("unchecked") // See the warnings discussion on page 265.
public ArrayStack<E> clone( )
{ // Clone an ArrayStack.
    ArrayStack<E> answer;

    try
    {
        answer = (ArrayStack<E>) super.clone( );
    }
    catch (CloneNotSupportedException e)
    {
        // This exception should not occur. But if it does, it would probably indicate a
        // programming error that made super.clone unavailable.
        // The most common error would be forgetting the "implements Cloneable"
        // clause at the start of this class.
        throw new RuntimeException
            ("This class does not implement Cloneable.");
    }

    answer.data = data.clone( );

    return answer;
}

public void ensureCapacity(int minimumCapacity)
{
    Object[ ]biggerArray;

    if (data.length < minimumCapacity)
    {
        biggerArray = new Object[minimumCapacity];
        System.arraycopy(data, 0, biggerArray, 0, manyItems);
        data = biggerArray;
    }
}

public int getCapacity( )
{
    return data.length;
}

public boolean isEmpty( )
{
    return (manyItems == 0);
```

(continued)

**340 Chapter 6 / Stacks**

(FIGURE 6.6 continued)

```
@SuppressWarnings("unchecked") // See the warnings discussion on page 265.
public E peek( )
{
    if (manyItems == 0)
        // EmptyStackException is from java.util, and its constructor has no argument.
        throw new EmptyStackException( );
    return (E) data[manyItems-1];
}

@SuppressWarnings("unchecked") // See the warnings discussion on page 265.
public E pop( )
{
    E answer;
    if (manyItems == 0)
        // EmptyStackException is from java.util, and its constructor has no argument.
        throw new EmptyStackException( );
    answer = (E) data[--manyItems];
    data[manyItems] = null; // For the garbage collector
    return answer;
}

public void push(E item)
{
    if (manyItems == data.length)
    {
        // Double the capacity and add 1; this works even if manyItems is 0. However, in
        // case that manyItems*2 + 1 is beyond Integer.MAX_VALUE, there will be an
        // arithmetic overflow and the stack will fail.
        ensureCapacity(manyItems * 2 + 1);
    }
    data[manyItems] = item;
    manyItems++;
}

public int size( )
{
    return manyItems;
}

public void trimToSize( )
{
    Object[ ] trimmedArray;

    if (data.length != manyItems)
    {
        trimmedArray = new Object[manyItems];
        System.arraycopy(data, 0, trimmedArray, 0, manyItems);
        data = trimmedArray;
    }
}
```

---

### Linked List Implementation of a Stack

A linked list is a natural way to implement a stack as a dynamic structure whose size can grow and shrink one item at a time. The head of the linked list serves as the top of the stack. Figure 6.7 contains the specification and implementation for a stack class that is implemented with a linked list. The class is called `LinkedStack` to distinguish it from our earlier `ObjectStack`. Here is a precise statement of the invariant of this version of the new stack ADT:

#### Invariant of the Generic `LinkedStack` Class

1. The items in the stack are stored in a linked list, with the top of the stack stored at the head node, down to the bottom of the stack at the final node.
2. The instance variable `top` is the head reference of the linked list of items.

As usual, all methods (except the constructors) assume that the stack is represented in this way when the method is activated, and all methods ensure that the stack continues to be represented in this way when the method finishes.

Because we are using a linked list, there are no capacity worries. Thus, there are no methods that deal with capacity. A program could build a stack with more than `Integer.MAX_VALUE` items, limited only by its amount of memory. However, beyond `Integer.MAX_VALUE`, the return value of the `size` method will be wrong because of arithmetic overflow.

As a further consequence of using a linked list, it makes sense to utilize the generic `Node<E>` class from <http://www.cs.colorado.edu/~main/edu/colorado/nodes/Node.java>. Thus, in Figure 6.7 you will find this import statement:

```
import edu.colorado.nodes.Node;
```

By using the `Node<E>` class, many of the stack methods can be implemented with just a line or two of code.

### Discussion of the Linked List Implementation of the Stack

The constructor, `size`, and `isEmpty` each require just one line of code. The `size` method actually uses the node's `listLength` method to do its work since we are not maintaining an instance variable that keeps track of the number of nodes. Because the head node of the list is the top of the stack, the implementation of `peek` is easy: `peek` just returns the data from the head node. The operations `push` and `pop` work by adding and removing nodes, always working at the head of the linked list. Adding and removing nodes at the head of the linked list is straightforward using the techniques of Section 4.2. The `clone` method makes use of the node's `listCopy` method to copy the original stack to the clone.

**FIGURE 6.7** Specification and Implementation for the Linked List Version of the Generic Stack Class

### Generic Class LinkedStack

❖ **public class LinkedStack<E> from the package edu.colorado.collections**

A LinkedStack is a stack of references to E objects.

**Limitations:**

Beyond Int.MAX\_VALUE items, size is wrong.

#### Specification

♦ **Constructor for the LinkedStack<E>**

public LinkedStack()

Initialize an empty stack.

**Postcondition:**

This stack is empty.

♦ **clone**

public LinkedStack<E> clone()

Generate a copy of this stack.

**Returns:**

The return value is a copy of this stack. Subsequent changes to the copy will not affect the original, nor vice versa.

**Throws:** OutOfMemoryError

Indicates insufficient memory for creating the clone.

♦ **isEmpty—peek—pop—push—size**

public boolean isEmpty()

public E peek()

public E pop()

public void push(E item)

public int size()

These are the standard stack specifications from Figure 6.1 on page 318.

#### Implementation

```
// File: LinkedStack.java from the package edu.colorado.collections
// Complete documentation is available above or from the LinkedStack link at
// http://www.cs.colorado.edu/~main/docs/,

package edu.colorado.collections;
import java.util.EmptyStackException;
import edu.colorado.nodes.Node;
```

(continued)

(FIGURE 6.7 continued)

```
public class LinkedStack<E> implements Cloneable
{
    // Invariant of the LinkedStack class:
    // 1. The items in the stack are stored in a linked list, with the top of the stack stored
    //     at the head node, down to the bottom of the stack at the final node.
    // 2. The instance variable top is the head reference of the linked list of items.
    private Node<E> top;

    public LinkedStack( )
    {
        top = null;
    }

    public LinkedStack<E> clone( )
    { // Clone a LinkedStack.
        LinkedStack<E> answer;

        try
        {
            answer = (LinkedStack<E>) super.clone( );
        }
        catch (CloneNotSupportedException e)
        {
            // This exception should not occur. But if it does, it would probably indicate a
            // programming error that made super.clone unavailable. The most common error
            // is forgetting the "implements Cloneable" clause at the start of this class.
            throw new RuntimeException
                ("This class does not implement Cloneable.");
        }

        answer.top = Node.listCopy(top); // Generic listCopy method
        return answer;
    }

    public boolean isEmpty( )
    {
        return (top == null);
    }

    public E peek( )
    {
        if (top == null)
            // EmptyStackException is from java.util, and its constructor has no argument.
            throw new EmptyStackException( );
        return top.getData( );
    }
```

(continued)

**344 Chapter 6 / Stacks**

(FIGURE 6.7 continued)

```
public E pop( )
{
    E answer;

    if (top == null)
        // EmptyStackException is from java.util, and its constructor has no argument.
        throw new EmptyStackException( );

    answer = top.getData( );
    top = top.getLink( );
    return answer;
}

public void push(E item)
{
    top = new Node<E>(item, top);
}

public int size( )
{
    return Node.listLength(top); // Generic listLength method
}
```

---

**Self-Test Exercises for Section 6.3**

10. For the array version of the stack, which element of the array contains the top of the stack? In the linked list version, where is the stack's top?
11. For the array version of the stack, write a new member function that returns the maximum number of items that can be added to the stack without stack overflow.
12. Give the full implementation of an accessor method that returns the second item from the top of the stack without actually changing the stack. Write separate solutions for the two different stack versions.
13. For the linked list version of the stack, do we maintain references to both the head and the tail?
14. Is the constructor really needed for the linked list version of the stack? What would happen if we omitted the constructor?
15. Do a time analysis of the `size` method for the linked list version of the stack. If the method is not constant time, then can you think of a different approach that is constant time?
16. What kind of exception is thrown if you try to pop an empty stack? Which Java Class Library defines this exception?

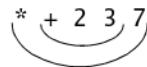
## 6.4 MORE COMPLEX STACK APPLICATIONS

### Evaluating Postfix Expressions

We normally write an arithmetic operation between its two arguments; for example, the  $+$  operation occurs between the 2 and the 3 in the arithmetic expression  $2 + 3$ . This is called *infix notation*. There is another way of writing arithmetic operations that places the operation in front of the two arguments; for example,  $+ 2 3$  evaluates to 5. This is called **Polish prefix notation**, or simply **prefix notation**.

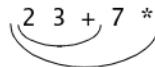
A **prefix** is something attached to the front of an expression. You may have heard about similar prefixes for words, such as the prefix *un* in *unbelievable*. Thus, it makes sense to call this notation *prefix notation*. But why *Polish*? It is called Polish because it was devised by the Polish mathematician Jan Łukasiewicz. It would be more proper to call it *Łukasiewicz notation*, but apparently non-Polish-speaking people have trouble pronouncing Łukasiewicz (lü-kä-sha-vēch).

Using prefix notation, parentheses are completely unneeded. For example, the expression  $(2 + 3) * 7$  written in Polish prefix notation is:



The curved lines under the expression indicate groupings of subexpressions (but the lines are not actually part of the prefix notation).

If we prefer, we can write the operations after the two numbers being combined. This is called **Polish postfix notation**, or more simply **postfix notation** (or sometimes **reverse Polish notation**). For example, the expression  $(2 + 3) * 7$  written in Polish postfix notation is:



Once again, the curves merely clarify the groupings of subexpressions, and these curves are not actually part of the postfix notation.

Here's a longer example. The postfix expression  $7 3 5 * + 4 -$  is equivalent to the infix expression  $(7 + (3 * 5)) - 4$ . Notice that an operation is applied to the two numbers that are immediately before it, so the multiplication is  $3 * 5$ . Sometimes one or both numbers for an operation are computed from a subexpression; for example, the  $+$  operation in the example is applied to the "number"  $3 * 5$  and the number before that (the 7), resulting in  $7 + (3 * 5)$ .

Do not intermix prefix and postfix notation. You should consistently use one or the other and not mix them together in a single expression.

Postfix notation is handy because it does not require parentheses and because it is particularly easy to evaluate (once you learn to use the notation). In fact, postfix notation often is used internally for computers because of the ease of

*infix versus  
prefix notation*

*the origin of the  
notation*

*postfix notation*

**346 Chapter 6 / Stacks**

*our goal:  
evaluation of  
postfix  
expressions*

evaluation. We will describe an algorithm to evaluate a postfix expression. When converted to a Java program, the postfix evaluation is similar to the calculator program (Figure 6.5 on page 330)—although, from our comments, you might guess that the postfix evaluation is actually simpler than the infix evaluation required in the calculator program.

There are two input format issues that we must handle. When entering postfix notation, we will require a space between two consecutive numbers so that you can tell where one number ends and another begins. For example, the input

35 6

consists of two numbers, 35 and 6, with a space in between. This is different from the input

356

*postfix  
evaluation  
algorithm*

which is just a single number, 356. A second input issue: You probably want to restrict the input to non-negative numbers to avoid the complication of distinguishing the negative sign of a number from a binary subtraction operation.

Our algorithm for evaluating a postfix expression uses only one stack, which is a stack of numbers. There is no need for a second stack of operation symbols because *each operation is used as soon as it is read*. In fact, the reason why postfix evaluation is easy is precisely because each operation symbol is used as soon as it is read. In the algorithm, we assume that each input entry is either a number or an operation. For simplicity, we will assume that all the operations take two arguments. The complete evaluation algorithm is given in Figure 6.8, along with an example computation.

Let's study the example to see how the algorithm works. Each time an operation appears in the input, the operands for the operation are the two most recently seen numbers. For example, in Figure 6.8(c), we are about to read the \* symbol. Since we have just pushed 3 and 2 onto the stack, the \* causes a multiplication of  $3 * 2$ , resulting in 6. The result of 6 is then pushed onto the stack, as shown in Figure 6.8(d).

Sometimes the “most recently seen number” is not actually an input number; instead, it is a number that we computed and pushed back onto the stack. For example, in Figure 6.8(d), we are about to read the first +. At this point, 6 is on top of the stack (as a result of multiplying  $3 * 2$ ). Below the 6 is the number 5. So the “two most recently seen numbers” are the 6 (that we computed) and the 5 (underneath the 6). We add these two numbers, resulting in 11 (which we push onto the stack, as shown in Figure 6.8(e)).

And so the process continues: Each time we encounter an operation, the operation is immediately applied to the two most recently seen numbers, which always reside in the top two positions of the stack. When the input is exhausted, the number remaining in the stack is the value of the entire expression.

**FIGURE 6.8** Evaluating a Postfix ExpressionPseudocode

1. Initialize a stack of double numbers.
2. do
  - if** (the next input is a number)
    - Read the next input and push it onto the stack.
  - else**
    - {
    - Read the next character, which is an operation symbol.
    - Pop two numbers off the stack.
    - Combine the two numbers with the operation (using the *second* number popped as the *left* operand) and push the result onto the stack.
    - }
  - while** (there is more of the expression to read);
3. At this point, the stack contains one number, which is the value of the expression.

Example

Evaluate the postfix expression

5 3 2 \* + 4 - 5 +

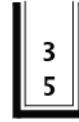
(a) Input so far (shaded):

5 3 2 \* + 4 - 5 +



(b) Input so far (shaded):

5 3 2 \* + 4 - 5 +



(c) Input so far (shaded):

5 3 2 \* + 4 - 5 +



(d) Input so far (shaded):

5 3 2 \* + 4 - 5 +



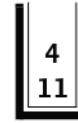
(e) Input so far (shaded):

5 3 2 \* + 4 - 5 +



(f) Input so far (shaded):

5 3 2 \* + 4 - 5 +



(g) Input so far (shaded):

5 3 2 \* + 4 - 5 +



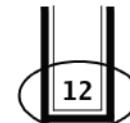
(h) Input so far (shaded):

5 3 2 \* + 4 - 5 +



(i) Input so far (shaded):

5 3 2 \* + 4 - 5 +



















































**BooleanSource.** During each simulated second, the BooleanSource provides us with a single boolean value indicating whether a new customer has arrived during that second. A `true` value indicates that a customer has arrived; `false` indicates that no customer arrived. With this in mind, we propose two methods: a constructor and a method called `query`.

The constructor for the BooleanSource has one argument, which is the probability that the BooleanSource returns `true` to a query. The probability is expressed as a decimal value between 0 and 1. For example, suppose our program uses the name `arrival` for its BooleanSource, and we want to simulate the situation in which a new customer arrives during 1% of the simulated seconds. Then our program would have the following declaration:

```
BooleanSource arrival = new BooleanSource(0.01);
```

The second method of the BooleanSource can be called to obtain the next value in the BooleanSource's sequence of values. Here is the specification:

◆ **query (method of the BooleanSource)**

```
public boolean query()
```

Get the next value from this BooleanSource.

**Returns:**

The return value is either `true` or `false`; the probability of a `true` value is determined by the argument that was given to the constructor.

There are several ways of generating random boolean values, but at this specification stage, we don't need to worry about such implementation details.

**Averager.** The averager has a constructor that initializes the averager so that it is ready to accept numbers. The numbers will be given to the averager one at a time through a method called `addNumber`. For example, suppose our averager is named `waitTimes`, and the next number in the sequence is 10. Then we will activate `waitTimes.addNumber(10)`; the averager also has two methods to retrieve its results: `average` and `howManyNumbers`, as specified here:

◆ **Constructor for the Averager**

```
public Averager()
```

Initialize an Averager so that it is ready to accept numbers.

◆ **addNumber**

```
public void addNumber(double value)
```

Give another number to this Averager.

◆ **average**

```
public double average()
```

The return value is the average of all numbers given to this Averager.

◆ **howManyNumbers**

```
public int howManyNumbers()
```

Provide a count of how many numbers have been given to this Averager.



























































































































































































































































































































































**546 Chapter 10 / Tree Projects**

The combination of removing the largest element from `subset[i]` and placing a copy of the largest element into `data[i]` can be accomplished by activating another new private method, `removeBiggest`, with a specification shown here:

```
private int removeBiggest( )
// Precondition: (dataCount > 0) and this entire B-tree is valid.
// Postcondition: The largest element in this set has been removed, and the
// return value is this removed element. The entire B-tree is still valid,
// EXCEPT that the number of elements in the root of this
// set might be one less than the allowed minimum.
```

By using `removeBiggest`, most of Step 2d is done with one statement:

```
data[i] = subset[i].removeBiggest();
```

After this statement finishes, we have deleted the largest element from `subset[i]` and placed a copy of this element into `data[i]` (replacing the target). The work that remains is to fix the possible shortage that may occur in the root of `subset[i]` (since the postcondition of `subset[i].removeBiggest` allows for the possibility that the root of `subset[i]` ends up with `MINIMUM - 1` elements). How do we fix such a shortage? We can use the same `fixShortage` method that we used at the end of Step 2c. Thus, the entire code for Step 2d is the following:

```
data[i] = subset[i].removeBiggest();
if (subset[i].dataCount < MINIMUM)
    fixShortage(i);
return true; // To indicate that we removed the target.
```

We have two more issues to deal with: the designs of `fixShortage` and `removeBiggest`.

*four situations  
for the  
fixShortage  
pseudocode*

### A Private Method to Fix a Shortage in a Child

When `fixShortage(i)` is activated, we know that `subset[i]` has `MINIMUM - 1` elements. How can we correct this problem? There are four situations that you can consider:

**Case 1 of fixShortage: Transfer an Extra Element from `subset[i-1]`.**  
Suppose `subset[i-1]` has more than the minimum number of elements. Then we can carry out these transfers:

- a. Transfer `data[i-1]` down to the front of `subset[i].data`.  
Remember to shift over the existing elements to make room and add one to `subset[i].dataCount`.
- b. Transfer the final element of `subset[i-1].data` up to replace `data[i-1]` and subtract one from `subset[i-1].dataCount`.





































































































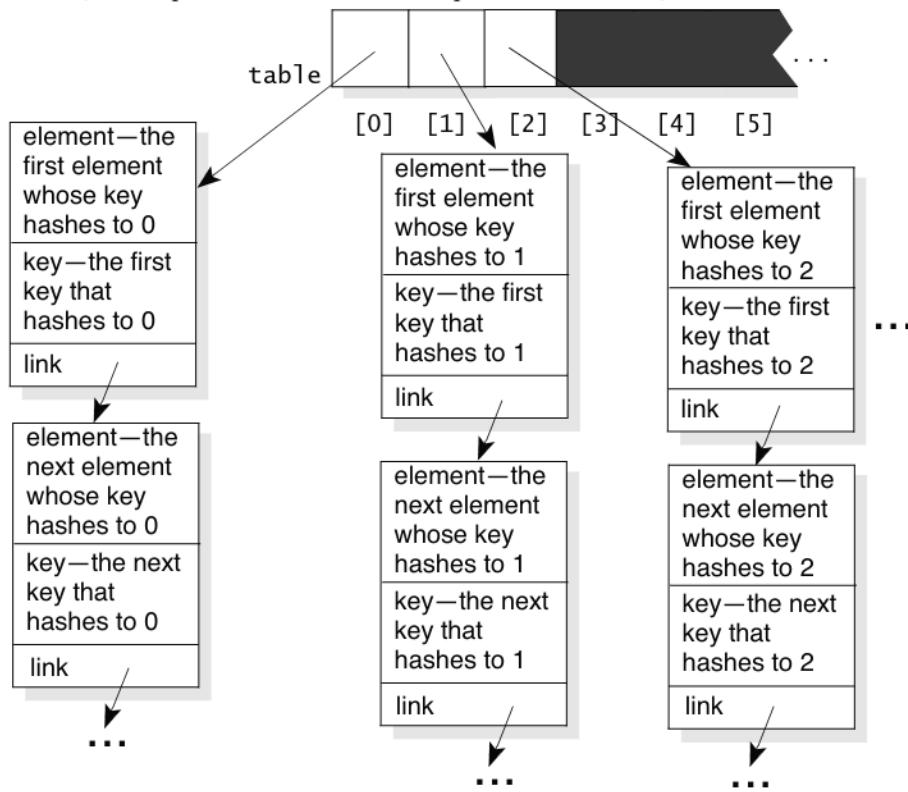








In effect, the implementation has 811 separate linked lists, as shown here:



All the elements that hash to location  $i$  are placed on the linked list, which has its head reference stored in `table[i]`. Figure 11.6 shows an outline of the `Table` class declaration is using this scheme. In this outline, the table's size is determined by the constructor. The separate class, `ChainedHashNode<K, E>`, can be placed in the same file as the `ChainedTable<K, E>` class.

#### Self-Test Exercises for Section 11.4

11. Consider the chaining version of the table we have described. What value will be in each component of `table` after the constructor finishes?
12. Write a private method that can be used to carry out the first two shaded lines in the outline of Figure 11.6 on page 602.
13. Use your `put` method to place six items in a hash table with a table size of 811. Use the keys 811, 0, 1623, 2435, 3247, and 2.
14. In our new `ChainedTable` class, the array is still a fixed size. It never grows after the constructor allocates it. So, is there a limit to the number of elements in the chained hash table?
15. Suppose the keys can be compared using a `compareTo` method (such as in Programming Project 11 on page 519). Can you think of some advantage to keeping each linked list of the table sorted from smallest key to largest key?











































































































Here is the pseudocode for the heapsort algorithm we have been describing:

```
// Heapsort for the array called data with n elements
1. Convert the array of n elements into a heap.
2. unsorted = n; // The number of elements in the unsorted side
3. while (unsorted > 1)
{
    // Reduce the unsorted side by 1.
    unsorted--;
    Swap data[0] with data[unsorted].
    The unsorted side of the array is now a heap with the root out of place.
    Do a reheapification downward to turn the unsorted side back into
    a heap.
}
```

The implementation of this pseudocode is shown in Figure 12.6. In the implementation we use two methods, `makeHeap` and `reheapifyDown`, which we discuss next.

---

**FIGURE 12.6** Heapsort

### Implementation

```
public static void heapsort(int[ ] data, int n)
{
    int unsorted; // Size of the unsorted part of the array
    int temp;     // Used during the swapping of two array locations

    makeHeap(data, n);

    unsorted = n;

    while (unsorted > 1)
    {
        unsorted--;

        // Swap the largest element (data[0]) with the final element of unsorted part
        temp = data[0];
        data[0] = data[unsorted];
        data[unsorted] = temp;

        reheapifyDown(data, unsorted);
    }
}
```

---







































































































































































































































































































































































