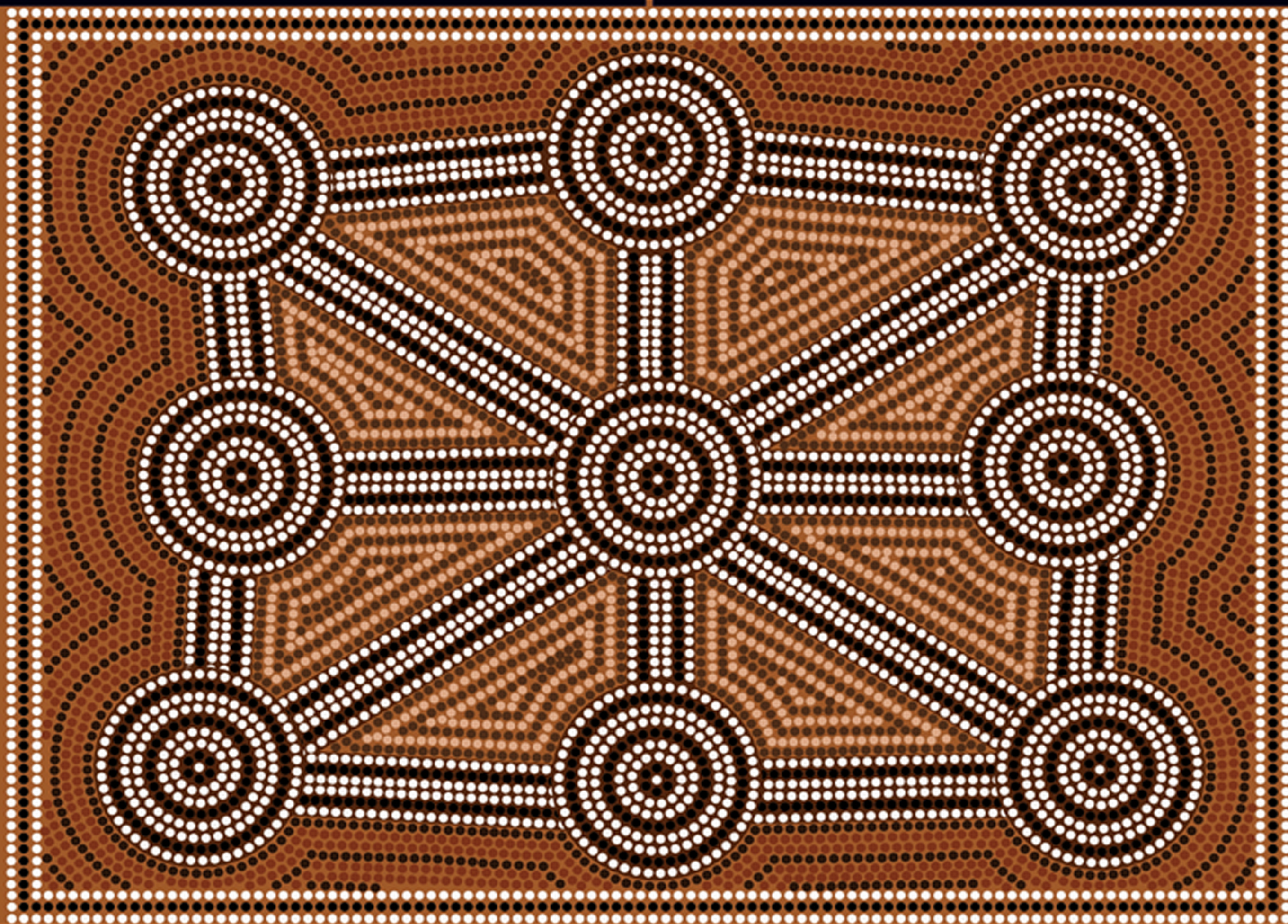
SIXTH EDITION

DATA STRUCTURES & ALGORITHMS



MICHAEL T. GOODRICH
ROBERTO TAMASSIA
MICHAEL H. GOLDWASSER

in JAVA™

WILEY

Data Structures and Algorithms in Java™

Sixth Edition

Michael T. Goodrich
Department of Computer Science
University of California, Irvine

Roberto Tamassia
Department of Computer Science
Brown University

Michael H. Goldwasser
Department of Mathematics and Computer Science
Saint Louis University

WILEY

Vice President and Executive Publisher	Don Fowley
Executive Editor	Beth Lang Golub
Assistant Marketing Manager	Debbie Martin
Sponsoring Editor	Mary O'Sullivan
Project Editor	Ellen Keohane
Associate Production Manager	Joyce Poh
Cover Designer	Kenji Ngieng

This book was set in LaTeX by the authors, and printed and bound by RR Donnelley. The cover was printed by RR Donnelley.

Trademark Acknowledgments: *Java is a trademark of Oracle Corporation. Unix® is a registered trademark in the United States and other countries, licensed through X/Open Company, Ltd. PowerPoint® is a trademark of Microsoft Corporation. All other product names mentioned herein are the trademarks of their respective owners.*

This book is printed on acid free paper.

Founded in 1807, John Wiley & Sons, Inc. has been a valued source of knowledge and understanding for more than 200 years, helping people around the world meet their needs and fulfill their aspirations. Our company is built on a foundation of principles that include responsibility to the communities we serve and where we live and work. In 2008, we launched a Corporate Citizenship Initiative, a global effort to address the environmental, social, economic, and ethical challenges we face in our business. Among the issues we are addressing are carbon impact, paper specifications and procurement, ethical conduct within our business and among our vendors, and community and charitable support. For more information, please visit our website: www.wiley.com/go/citizenship.

Copyright © 2014, 2010 John Wiley & Sons, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc. 222 Rosewood Drive, Danvers, MA 01923, website www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030-5774, (201) 748-6011, fax (201) 748-6008, website http://www.wiley.com/go/ permissions.

Evaluation copies are provided to qualified academics and professionals for review purposes only, for use in their courses during the next academic year. These copies are licensed and may not be sold or transferred to a third party. Upon completion of the review period, please return the evaluation copy to Wiley. Return instructions and a free of charge return mailing label are available at www.wiley.com/go/returnlabel. If you have chosen to adopt this textbook for use in your course, please accept this book as your complimentary desk copy. Outside of the United States, please contact your local sales representative.

ISBN: 978-1-118-77133-4 (paperback)

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

To Karen, Paul, Anna, and Jack
– *Michael T. Goodrich*

To Isabel
– *Roberto Tamassia*

To Susan, Calista, and Maya
– *Michael H. Goldwasser*

Preface to the Sixth Edition

Data Structures and Algorithms in Java provides an introduction to data structures and algorithms, including their design, analysis, and implementation. The major changes in this sixth edition include the following:

- We redesigned the entire code base to increase clarity of presentation and consistency in style and convention, including reliance on *type inference*, as introduced in Java 7, to reduce clutter when instantiating generic types.
- We added 38 new figures, and redesigned 144 existing figures.
- We revised and expanded exercises, bringing the grand total to 794 exercises! We continue our approach of dividing them into reinforcement, creativity, and project exercises. However, we have chosen not to reset the numbering scheme with each new category, thereby avoiding possible ambiguity between exercises such as R-7.5, C-7.5, P-7.5.
- The introductory chapters contain additional examples of classes and inheritance, increased discussion of Java's generics framework, and expanded coverage of cloning and equivalence testing in the context of data structures.
- A new chapter, dedicated to the topic of recursion, provides comprehensive coverage of material that was previously divided within Chapters 3, 4, and 9 of the fifth edition, while newly introducing the use of recursion when processing file systems.
- We provide a new empirical study of the efficiency of Java's StringBuilder class relative to the repeated concatenation of strings, and then discuss the theoretical underpinnings of its amortized performance.
- We provide increased discussion of iterators, contrasting between so-called *lazy iterators* and *snapshot iterators*, with examples of both styles of implementation for several data structures.
- We have increased the use of abstract base classes to reduce redundancy when providing multiple implementations of a common interface, and the use of nested classes to provide greater encapsulation for our data structures.
- We have included complete Java implementations for many data structures and algorithms that were only described with pseudocode in earlier editions. These new implementations include both array-based and linked-list-based queue implementations, a heap-based *adaptable* priority queue, a bottom-up heap construction, hash tables with either separate chaining or linear probing, splay trees, dynamic programming for the least-common subsequence problem, a union-find data structure with path compression, breadth-first search of a graph, the Floyd-Warshall algorithm for computing a graph's transitive closure, topological sorting of a DAG, and both the Prim-Jarník and Kruskal algorithms for computing a minimum spanning tree.

Prerequisites

We assume that the reader is at least vaguely familiar with a high-level programming language, such as C, C++, Python, or Java, and that he or she understands the main constructs from such a high-level language, including:

- Variables and expressions
- Methods (also known as functions or procedures)
- Decision structures (such as if-statements and switch-statements)
- Iteration structures (for-loops and while-loops)

For readers who are familiar with these concepts, but not with how they are expressed in Java, we provide a primer on the Java language in Chapter 1. Still, this book is primarily a data structures book, not a Java book; hence, it does not provide a comprehensive treatment of Java. Nevertheless, we do not assume that the reader is necessarily familiar with object-oriented design or with linked structures, such as linked lists, for these topics are covered in the core chapters of this book.

In terms of mathematical background, we assume the reader is somewhat familiar with topics from high-school mathematics. Even so, in Chapter 4, we discuss the seven most-important functions for algorithm analysis. In fact, sections that use something other than one of these seven functions are considered optional, and are indicated with a star (★).

Online Resources

This book is accompanied by an extensive set of online resources, which can be found at the following website:

www.wiley.com/college/goodrich

Included on this website is a collection of educational aids that augment the topics of this book, for both students and instructors. For all readers, and especially for students, we include the following resources:

- All Java source code presented in this book
- An appendix of useful mathematical facts
- PDF handouts of PowerPoint slides (four-per-page)
- A study guide with hints to exercises, indexed by problem number

For instructors using this book, we include the following additional teaching aids:

- Solutions to hundreds of the book's exercises
- Color versions of all figures and illustrations from the book
- Slides in PowerPoint and PDF (one-per-page) format

The slides are fully editable, so as to allow an instructor using this book full freedom in customizing his or her presentations.

Use as a Textbook

The design and analysis of efficient data structures has long been recognized as a core subject in computing. We feel that the central role of data structure design and analysis in the curriculum is fully justified, given the importance of efficient data structures and algorithms in most software systems, including the Web, operating systems, databases, compilers, and scientific simulation systems.

This book is designed for use in a beginning-level data structures course, or in an intermediate-level introduction to algorithms course. The chapters for this book are organized to provide a pedagogical path that starts with the basics of Java programming and object-oriented design. We then discuss concrete structures including arrays and linked lists, and foundational techniques like algorithm analysis and recursion. In the main portion of the book we present fundamental data structures and algorithms, concluding with a discussion of memory management. A detailed table of contents follows this preface, beginning on page x.

To assist instructors in designing a course in the context of the IEEE/ACM 2013 Computing Curriculum, the following table describes curricular knowledge units that are covered within this book.

Knowledge Unit	**Relevant Material**
AL/Basic Analysis	Chapter 4 and Sections 5.2 & 12.1.4
AL/Algorithmic Strategies	Sections 5.3.3, 12.1.1, 13.2.1, 13.4.2, 13.5, 14.6.2 & 14.7
AL/Fundamental Data Structures and Algorithms	Sections 3.1.2, 5.1.3, 9.3, 9.4.1, 10.2, 11.1, 13.2, and Chapters 12 & 14
AL/Advanced Data Structures	Sections 7.2.1, 10.4, 11.2–11.6, 12.2.1, 13.3, 14.5.1 & 15.3
AR/Memory System Organization and Architecture	Chapter 15
DS/Sets, Relations, and Functions	Sections 9.2.2 & 10.5
DS/Proof Techniques	Sections 4.4, 5.2, 7.2.3, 9.3.4 & 12.3.1
DS/Basics of Counting	Sections 2.2.3, 6.2.2, 8.2.2 & 12.1.4.
DS/Graphs and Trees	Chapters 8 and 14
DS/Discrete Probability	Sections 3.1.3, 10.2, 10.4.2 & 12.2.1
PL/Object-Oriented Programming	Chapter 2 and Sections 7.3, 9.5.1 & 11.2.1
SDF/Algorithms and Design	Sections 2.1, 4.3 & 12.1.1
SDF/Fundamental Programming Concepts	Chapters 1 & 5
SDF/Fundamental Data Structures	Chapters 3 & 6, and Sections 1.3, 9.1 & 10.1
SDF/Developmental Methods	Sections 1.9 & 2.4
SE/Software Design	Section 2.1

Mapping the *IEEE/ACM 2013 Computing Curriculum* knowledge units to coverage within this book.

About the Authors

Michael Goodrich received his Ph.D. in Computer Science from Purdue University in 1987. He is currently a Chancellor's Professor in the Department of Computer Science at University of California, Irvine. Previously, he was a professor at Johns Hopkins University. He is a Fulbright Scholar and a Fellow of the American Association for the Advancement of Science (AAAS), Association for Computing Machinery (ACM), and Institute of Electrical and Electronics Engineers (IEEE). He is a recipient of the IEEE Computer Society Technical Achievement Award, the ACM Recognition of Service Award, and the Pond Award for Excellence in Undergraduate Teaching.

Roberto Tamassia received his Ph.D. in Electrical and Computer Engineering from the University of Illinois at Urbana–Champaign in 1988. He is the Plastech Professor of Computer Science and the Chair of the Department of Computer Science at Brown University. He is also the Director of Brown's Center for Geometric Computing. His research interests include information security, cryptography, analysis, design, and implementation of algorithms, graph drawing, and computational geometry. He is a Fellow of the American Association for the Advancement of Science (AAAS), Association for Computing Machinery (ACM) and Institute for Electrical and Electronic Engineers (IEEE). He is a recipient of the IEEE Computer Society Technical Achievement Award.

Michael Goldwasser received his Ph.D. in Computer Science from Stanford University in 1997. He is currently Professor and Director of the Computer Science program in the Department of Mathematics and Computer Science at Saint Louis University. He was previously a faculty member in the Department of Computer Science at Loyola University Chicago. His research interests focus on the design and implementation of algorithms, having published work involving approximation algorithms, online computation, computational biology, and computational geometry. He is also active in the computer science education community.

Additional Books by These Authors

- Di Battista, Eades, Tamassia, and Tollis, *Graph Drawing*, Prentice Hall
- Goodrich, Tamassia, and Goldwasser, *Data Structures and Algorithms in Python*, Wiley
- Goodrich, Tamassia, and Mount, *Data Structures and Algorithms in C++*, Wiley
- Goodrich and Tamassia, *Algorithm Design: Foundations, Analysis, and Internet Examples*, Wiley
- Goodrich and Tamassia, *Introduction to Computer Security*, Addison-Wesley
- Goldwasser and Letscher, *Object-Oriented Programming in Python*, Prentice Hall

Acknowledgments

There are so many individuals who have made contributions to the development of this book over the past decade, it is difficult to name them all. We wish to reiterate our thanks to the many research collaborators and teaching assistants whose feedback shaped the previous versions of this material. The benefits of those contributions carry forward to this book.

For the sixth edition, we are indebted to the outside reviewers and readers for their copious comments, emails, and constructive criticisms. We therefore thank the following people for their comments and suggestions: Sameer O. Abufardeh (North Dakota State University), Mary Boelk (Marquette University), Frederick Crabbe (United States Naval Academy), Scot Drysdale (Dartmouth College), David Eisner, Henry A. Etlinger (Rochester Institute of Technology), Chun-Hsi Huang (University of Connecticut), John Lasseter (Hobart and William Smith Colleges), Yupeng Lin, Suely Oliveira (University of Iowa), Vincent van Oostrom (Utrecht University), Justus Piater (University of Innsbruck), Victor I. Shtern (Boston University), Tim Soethout, and a number of additional anonymous reviewers.

There have been a number of friends and colleagues whose comments have led to improvements in the text. We are particularly thankful to Erin Chambers, Karen Goodrich, David Letscher, David Mount, and Ioannis Tollis for their insightful comments. In addition, contributions by David Mount to the coverage of recursion and to several figures are gratefully acknowledged.

We appreciate the wonderful team at Wiley, including our editor, Beth Lang Golub, for her enthusiastic support of this project from beginning to end, and the Product Solutions Group editors, Mary O'Sullivan and Ellen Keohane, for carrying the project to its completion. The quality of this book is greatly enhanced as a result of the attention to detail demonstrated by our copyeditor, Julie Kennedy. The final months of the production process were gracefully managed by Joyce Poh.

Finally, we would like to warmly thank Karen Goodrich, Isabel Cruz, Susan Goldwasser, Giuseppe Di Battista, Franco Preparata, Ioannis Tollis, and our parents for providing advice, encouragement, and support at various stages of the preparation of this book, and Calista and Maya Goldwasser for offering their advice regarding the artistic merits of many illustrations. More importantly, we thank all of these people for reminding us that there are things in life beyond writing books.

Michael T. Goodrich
Roberto Tamassia
Michael H. Goldwasser

Contents

1 Java Primer 1

1.1 Getting Started 2
1.1.1 Base Types 4
1.2 Classes and Objects 5
1.2.1 Creating and Using Objects 6
1.2.2 Defining a Class 9
1.3 Strings, Wrappers, Arrays, and Enum Types 17
1.4 Expressions 23
1.4.1 Literals 23
1.4.2 Operators 24
1.4.3 Type Conversions 28
1.5 Control Flow 30
1.5.1 The If and Switch Statements 30
1.5.2 Loops 33
1.5.3 Explicit Control-Flow Statements 37
1.6 Simple Input and Output 38
1.7 An Example Program 41
1.8 Packages and Imports 44
1.9 Software Development 46
1.9.1 Design 46
1.9.2 Pseudocode 48
1.9.3 Coding 49
1.9.4 Documentation and Style 50
1.9.5 Testing and Debugging 53
1.10 Exercises 55

2 Object-Oriented Design 59

2.1 Goals, Principles, and Patterns 60
2.1.1 Object-Oriented Design Goals 60
2.1.2 Object-Oriented Design Principles 61
2.1.3 Design Patterns 63
2.2 Inheritance 64
2.2.1 Extending the CreditCard Class 65
2.2.2 Polymorphism and Dynamic Dispatch 68
2.2.3 Inheritance Hierarchies 69
2.3 Interfaces and Abstract Classes 76
2.3.1 Interfaces in Java 76
2.3.2 Multiple Inheritance for Interfaces 79
2.3.3 Abstract Classes 80
2.4 Exceptions 82
2.4.1 Catching Exceptions 82
2.4.2 Throwing Exceptions 85
2.4.3 Java's Exception Hierarchy 86
2.5 Casting and Generics 88

2.5.1 Casting . . . 88
2.5.2 Generics . . . 91
2.6 Nested Classes . . . 96
2.7 Exercises . . . 97

3 Fundamental Data Structures 103
3.1 Using Arrays . . . 104
3.1.1 Storing Game Entries in an Array . . . 104
3.1.2 Sorting an Array . . . 110
3.1.3 java.util Methods for Arrays and Random Numbers . . . 112
3.1.4 Simple Cryptography with Character Arrays . . . 115
3.1.5 Two-Dimensional Arrays and Positional Games . . . 118
3.2 Singly Linked Lists . . . 122
3.2.1 Implementing a Singly Linked List Class . . . 126
3.3 Circularly Linked Lists . . . 128
3.3.1 Round-Robin Scheduling . . . 128
3.3.2 Designing and Implementing a Circularly Linked List . . . 129
3.4 Doubly Linked Lists . . . 132
3.4.1 Implementing a Doubly Linked List Class . . . 135
3.5 Equivalence Testing . . . 138
3.5.1 Equivalence Testing with Arrays . . . 139
3.5.2 Equivalence Testing with Linked Lists . . . 140
3.6 Cloning Data Structures . . . 141
3.6.1 Cloning Arrays . . . 142
3.6.2 Cloning Linked Lists . . . 144
3.7 Exercises . . . 145

4 Algorithm Analysis 149
4.1 Experimental Studies . . . 151
4.1.1 Moving Beyond Experimental Analysis . . . 154
4.2 The Seven Functions Used in This Book . . . 156
4.2.1 Comparing Growth Rates . . . 163
4.3 Asymptotic Analysis . . . 164
4.3.1 The "Big-Oh" Notation . . . 164
4.3.2 Comparative Analysis . . . 168
4.3.3 Examples of Algorithm Analysis . . . 170
4.4 Simple Justification Techniques . . . 178
4.4.1 By Example . . . 178
4.4.2 The "Contra" Attack . . . 178
4.4.3 Induction and Loop Invariants . . . 179
4.5 Exercises . . . 182

5 Recursion 189
5.1 Illustrative Examples . . . 191
5.1.1 The Factorial Function . . . 191
5.1.2 Drawing an English Ruler . . . 193
5.1.3 Binary Search . . . 196

5.1.4 File Systems . . . 198
5.2 Analyzing Recursive Algorithms . . . 202
5.3 Further Examples of Recursion . . . 206
5.3.1 Linear Recursion . . . 206
5.3.2 Binary Recursion . . . 211
5.3.3 Multiple Recursion . . . 212
5.4 Designing Recursive Algorithms . . . 214
5.5 Recursion Run Amok . . . 215
5.5.1 Maximum Recursive Depth in Java . . . 218
5.6 Eliminating Tail Recursion . . . 219
5.7 Exercises . . . 221

6 Stacks, Queues, and Deques 225
6.1 Stacks . . . 226
6.1.1 The Stack Abstract Data Type . . . 227
6.1.2 A Simple Array-Based Stack Implementation . . . 230
6.1.3 Implementing a Stack with a Singly Linked List . . . 233
6.1.4 Reversing an Array Using a Stack . . . 234
6.1.5 Matching Parentheses and HTML Tags . . . 235
6.2 Queues . . . 238
6.2.1 The Queue Abstract Data Type . . . 239
6.2.2 Array-Based Queue Implementation . . . 241
6.2.3 Implementing a Queue with a Singly Linked List . . . 245
6.2.4 A Circular Queue . . . 246
6.3 Double-Ended Queues . . . 248
6.3.1 The Deque Abstract Data Type . . . 248
6.3.2 Implementing a Deque . . . 250
6.3.3 Deques in the Java Collections Framework . . . 251
6.4 Exercises . . . 252

7 List and Iterator ADTs 257
7.1 The List ADT . . . 258
7.2 Array Lists . . . 260
7.2.1 Dynamic Arrays . . . 263
7.2.2 Implementing a Dynamic Array . . . 264
7.2.3 Amortized Analysis of Dynamic Arrays . . . 265
7.2.4 Java's StringBuilder class . . . 269
7.3 Positional Lists . . . 270
7.3.1 Positions . . . 272
7.3.2 The Positional List Abstract Data Type . . . 272
7.3.3 Doubly Linked List Implementation . . . 276
7.4 Iterators . . . 282
7.4.1 The Iterable Interface and Java's For-Each Loop . . . 283
7.4.2 Implementing Iterators . . . 284
7.5 The Java Collections Framework . . . 288
7.5.1 List Iterators in Java . . . 289
7.5.2 Comparison to Our Positional List ADT . . . 290

7.5.3 List-Based Algorithms in the Java Collections Framework 291
7.6 Sorting a Positional List 293
7.7 Case Study: Maintaining Access Frequencies 294
7.7.1 Using a Sorted List 294
7.7.2 Using a List with the Move-to-Front Heuristic 297
7.8 Exercises 300

8 Trees 307
8.1 General Trees 308
8.1.1 Tree Definitions and Properties 309
8.1.2 The Tree Abstract Data Type 312
8.1.3 Computing Depth and Height 314
8.2 Binary Trees 317
8.2.1 The Binary Tree Abstract Data Type 319
8.2.2 Properties of Binary Trees 321
8.3 Implementing Trees 323
8.3.1 Linked Structure for Binary Trees 323
8.3.2 Array-Based Representation of a Binary Tree 331
8.3.3 Linked Structure for General Trees 333
8.4 Tree Traversal Algorithms 334
8.4.1 Preorder and Postorder Traversals of General Trees 334
8.4.2 Breadth-First Tree Traversal 336
8.4.3 Inorder Traversal of a Binary Tree 337
8.4.4 Implementing Tree Traversals in Java 339
8.4.5 Applications of Tree Traversals 343
8.4.6 Euler Tours 348
8.5 Exercises 350

9 Priority Queues 359
9.1 The Priority Queue Abstract Data Type 360
9.1.1 Priorities 360
9.1.2 The Priority Queue ADT 361
9.2 Implementing a Priority Queue 362
9.2.1 The Entry Composite 362
9.2.2 Comparing Keys with Total Orders 363
9.2.3 The AbstractPriorityQueue Base Class 364
9.2.4 Implementing a Priority Queue with an Unsorted List 366
9.2.5 Implementing a Priority Queue with a Sorted List 368
9.3 Heaps 370
9.3.1 The Heap Data Structure 370
9.3.2 Implementing a Priority Queue with a Heap 372
9.3.3 Analysis of a Heap-Based Priority Queue 379
9.3.4 Bottom-Up Heap Construction ★ 380
9.3.5 Using the java.util.PriorityQueue Class 384
9.4 Sorting with a Priority Queue 385
9.4.1 Selection-Sort and Insertion-Sort 386
9.4.2 Heap-Sort 388

9.5 Adaptable Priority Queues . . . 390
9.5.1 Location-Aware Entries . . . 391
9.5.2 Implementing an Adaptable Priority Queue . . . 392
9.6 Exercises . . . 395

10 Maps, Hash Tables, and Skip Lists 401
10.1 Maps . . . 402
10.1.1 The Map ADT . . . 403
10.1.2 Application: Counting Word Frequencies . . . 405
10.1.3 An AbstractMap Base Class . . . 406
10.1.4 A Simple Unsorted Map Implementation . . . 408
10.2 Hash Tables . . . 410
10.2.1 Hash Functions . . . 411
10.2.2 Collision-Handling Schemes . . . 417
10.2.3 Load Factors, Rehashing, and Efficiency . . . 420
10.2.4 Java Hash Table Implementation . . . 422
10.3 Sorted Maps . . . 428
10.3.1 Sorted Search Tables . . . 429
10.3.2 Two Applications of Sorted Maps . . . 433
10.4 Skip Lists . . . 436
10.4.1 Search and Update Operations in a Skip List . . . 438
10.4.2 Probabilistic Analysis of Skip Lists ★ . . . 442
10.5 Sets, Multisets, and Multimaps . . . 445
10.5.1 The Set ADT . . . 445
10.5.2 The Multiset ADT . . . 447
10.5.3 The Multimap ADT . . . 448
10.6 Exercises . . . 451

11 Search Trees 459
11.1 Binary Search Trees . . . 460
11.1.1 Searching Within a Binary Search Tree . . . 461
11.1.2 Insertions and Deletions . . . 463
11.1.3 Java Implementation . . . 466
11.1.4 Performance of a Binary Search Tree . . . 470
11.2 Balanced Search Trees . . . 472
11.2.1 Java Framework for Balancing Search Trees . . . 475
11.3 AVL Trees . . . 479
11.3.1 Update Operations . . . 481
11.3.2 Java Implementation . . . 486
11.4 Splay Trees . . . 488
11.4.1 Splaying . . . 488
11.4.2 When to Splay . . . 492
11.4.3 Java Implementation . . . 494
11.4.4 Amortized Analysis of Splaying ★ . . . 495
11.5 (2,4) Trees . . . 500
11.5.1 Multiway Search Trees . . . 500
11.5.2 (2,4)-Tree Operations . . . 503

11.6 Red-Black Trees . . . **510**
11.6.1 Red-Black Tree Operations . . . 512
11.6.2 Java Implementation . . . 522
11.7 Exercises . . . **525**

12 Sorting and Selection **531**
12.1 Merge-Sort . . . **532**
12.1.1 Divide-and-Conquer . . . 532
12.1.2 Array-Based Implementation of Merge-Sort . . . 537
12.1.3 The Running Time of Merge-Sort . . . 538
12.1.4 Merge-Sort and Recurrence Equations ★ . . . 540
12.1.5 Alternative Implementations of Merge-Sort . . . 541
12.2 Quick-Sort . . . **544**
12.2.1 Randomized Quick-Sort . . . 551
12.2.2 Additional Optimizations for Quick-Sort . . . 553
12.3 Studying Sorting through an Algorithmic Lens . . . **556**
12.3.1 Lower Bound for Sorting . . . 556
12.3.2 Linear-Time Sorting: Bucket-Sort and Radix-Sort . . . 558
12.4 Comparing Sorting Algorithms . . . **561**
12.5 Selection . . . **563**
12.5.1 Prune-and-Search . . . 563
12.5.2 Randomized Quick-Select . . . 564
12.5.3 Analyzing Randomized Quick-Select . . . 565
12.6 Exercises . . . **566**

13 Text Processing **573**
13.1 Abundance of Digitized Text . . . **574**
13.1.1 Notations for Character Strings . . . 575
13.2 Pattern-Matching Algorithms . . . **576**
13.2.1 Brute Force . . . 576
13.2.2 The Boyer-Moore Algorithm . . . 578
13.2.3 The Knuth-Morris-Pratt Algorithm . . . 582
13.3 Tries . . . **586**
13.3.1 Standard Tries . . . 586
13.3.2 Compressed Tries . . . 590
13.3.3 Suffix Tries . . . 592
13.3.4 Search Engine Indexing . . . 594
13.4 Text Compression and the Greedy Method . . . **595**
13.4.1 The Huffman Coding Algorithm . . . 596
13.4.2 The Greedy Method . . . 597
13.5 Dynamic Programming . . . **598**
13.5.1 Matrix Chain-Product . . . 598
13.5.2 DNA and Text Sequence Alignment . . . 601
13.6 Exercises . . . **605**

14 Graph Algorithms 611
14.1 Graphs . . . 612
14.1.1 The Graph ADT . . . 618
14.2 Data Structures for Graphs . . . 619
14.2.1 Edge List Structure . . . 620
14.2.2 Adjacency List Structure . . . 622
14.2.3 Adjacency Map Structure . . . 624
14.2.4 Adjacency Matrix Structure . . . 625
14.2.5 Java Implementation . . . 626
14.3 Graph Traversals . . . 630
14.3.1 Depth-First Search . . . 631
14.3.2 DFS Implementation and Extensions . . . 636
14.3.3 Breadth-First Search . . . 640
14.4 Transitive Closure . . . 643
14.5 Directed Acyclic Graphs . . . 647
14.5.1 Topological Ordering . . . 647
14.6 Shortest Paths . . . 651
14.6.1 Weighted Graphs . . . 651
14.6.2 Dijkstra's Algorithm . . . 653
14.7 Minimum Spanning Trees . . . 662
14.7.1 Prim-Jarník Algorithm . . . 664
14.7.2 Kruskal's Algorithm . . . 667
14.7.3 Disjoint Partitions and Union-Find Structures . . . 672
14.8 Exercises . . . 677

15 Memory Management and B-Trees 687
15.1 Memory Management . . . 688
15.1.1 Stacks in the Java Virtual Machine . . . 688
15.1.2 Allocating Space in the Memory Heap . . . 691
15.1.3 Garbage Collection . . . 693
15.2 Memory Hierarchies and Caching . . . 695
15.2.1 Memory Systems . . . 695
15.2.2 Caching Strategies . . . 696
15.3 External Searching and B-Trees . . . 701
15.3.1 (a,b) Trees . . . 702
15.3.2 B-Trees . . . 704
15.4 External-Memory Sorting . . . 705
15.4.1 Multiway Merging . . . 706
15.5 Exercises . . . 707

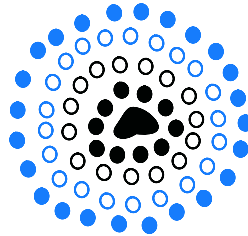
Bibliography 710

Index 714

Useful Mathematical Facts available at www.wiley.com/college/goodrich

Chapter

1 Java Primer



Contents

1.1 Getting Started **2**
1.1.1 Base Types 4
1.2 Classes and Objects **5**
1.2.1 Creating and Using Objects 6
1.2.2 Defining a Class 9
1.3 Strings, Wrappers, Arrays, and Enum Types **17**
1.4 Expressions **23**
1.4.1 Literals 23
1.4.2 Operators 24
1.4.3 Type Conversions 28
1.5 Control Flow **30**
1.5.1 The If and Switch Statements 30
1.5.2 Loops 33
1.5.3 Explicit Control-Flow Statements 37
1.6 Simple Input and Output **38**
1.7 An Example Program **41**
1.8 Packages and Imports **44**
1.9 Software Development **46**
1.9.1 Design 46
1.9.2 Pseudocode 48
1.9.3 Coding 49
1.9.4 Documentation and Style 50
1.9.5 Testing and Debugging 53
1.10 Exercises **55**

1.1 Getting Started

Building data structures and algorithms requires that we communicate detailed instructions to a computer. An excellent way to perform such communication is using a high-level computer language, such as Java. In this chapter, we provide an overview of the Java programming language, and we continue this discussion in the next chapter, focusing on object-oriented design principles. We assume that readers are somewhat familiar with an existing high-level language, although not necessarily Java. This book does not provide a complete description of the Java language (there are numerous language references for that purpose), but it does introduce all aspects of the language that are used in code fragments later in this book.

We begin our Java primer with a program that prints "Hello Universe!" on the screen, which is shown in a dissected form in Figure 1.1.

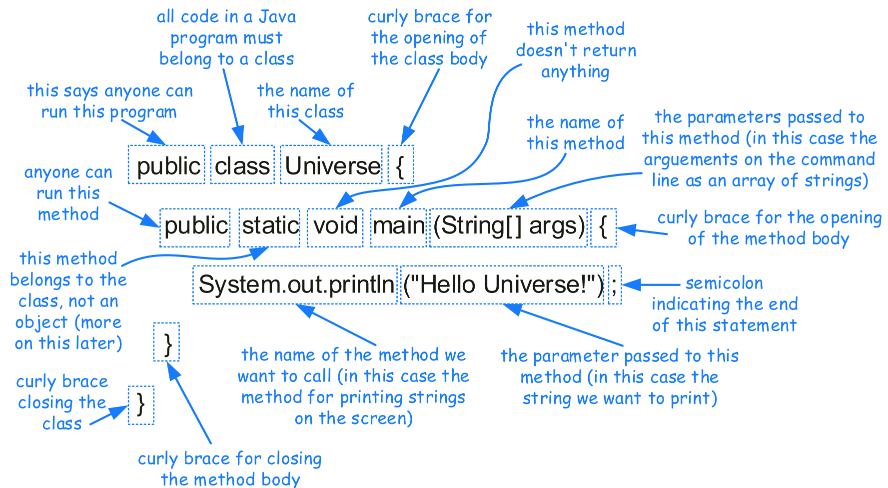


Figure 1.1: A "Hello Universe!" program.

In Java, executable statements are placed in functions, known as ***methods***, that belong to ***class*** definitions. The Universe class, in our first example, is extremely simple; its only method is a static one named main, which is the first method to be executed when running a Java program. Any set of statements between the braces "{" and "}" define a program ***block***. Notice that the entire Universe class definition is delimited by such braces, as is the body of the main method.

The name of a class, method, or variable in Java is called an ***identifier***, which can be any string of characters as long as it begins with a letter and consists of letters, numbers, and underscore characters (where "letter" and "number" can be from any written language defined in the Unicode character set). We list the exceptions to this general rule for Java identifiers in Table 1.1.

Reserved Words				
abstract	default	goto	package	synchronized
assert	do	if	private	this
boolean	double	implements	protected	throw
break	else	import	public	throws
byte	enum	instanceof	return	transient
case	extends	int	short	true
catch	false	interface	static	try
char	final	long	strictfp	void
class	finally	native	super	volatile
const	float	new	switch	while
continue	for	null		

Table 1.1: A listing of the reserved words in Java. These names cannot be used as class, method, or variable names.

Comments

In addition to executable statements and declarations, Java allows a programmer to embed comments, which are annotations provided for human readers that are not processed by the Java compiler. Java allows two kinds of comments: inline comments and block comments. Java uses a "//" to begin an inline comment, ignoring everything subsequently on that line. For example:

```
// This is an inline comment.
```

We will intentionally color all comments in blue in this book, so that they are not confused with executable code.

While inline comments are limited to one line, Java allows multiline comments in the form of block comments. Java uses a "/*" to begin a block comment and a "*/" to close it. For example:

```
/*
 * This is a block comment.
 */
```

Block comments that begin with "/**" (note the second asterisk) have a special purpose, allowing a program, called Javadoc, to read these comments and automatically generate software documentation. We discuss the syntax and interpretation of Javadoc comments in Section 1.9.4.

1.1.1 Base Types

For the most commonly used data types, Java provides the following ***base types*** (also called ***primitive types***):

boolean	a boolean value: true or false
char	16-bit Unicode character
byte	8-bit signed two's complement integer
short	16-bit signed two's complement integer
int	32-bit signed two's complement integer
long	64-bit signed two's complement integer
float	32-bit floating-point number (IEEE 754-1985)
double	64-bit floating-point number (IEEE 754-1985)

A variable having one of these types simply stores a value of that type. Integer constants, like 14 or 195, are of type **int**, unless followed immediately by an 'L' or 'l', in which case they are of type **long**. Floating-point constants, like 3.1416 or 6.022e23, are of type **double**, unless followed immediately by an 'F' or 'f', in which case they are of type **float**. Code Fragment 1.1 demonstrates the declaration, and initialization in some cases, of various base-type variables.

```
1 boolean flag = true;
2 boolean verbose, debug;               // two variables declared, but not yet initialized
3 char grade = 'A';
4 byte b = 12;
5 short s = 24;
6 int i, j, k = 257;                    // three variables declared; only k initialized
7 long l = 890L;                        // note the use of "L" here
8 float pi = 3.1416F;                   // note the use of "F" here
9 double e = 2.71828, a = 6.022e23;     // both variables are initialized
```

Code Fragment 1.1: Declarations and initializations of several base-type variables.

Note that it is possible to declare (and initialize) multiple variables of the same type in a single statement, as done on lines 2, 6, and 9 of this example. In this code fragment, variables verbose, debug, i, and j remain uninitialized. Variables declared locally within a block of code must be initialized before they are first used.

A nice feature of Java is that when base-type variables are declared as instance variables of a class (see next section), Java ensures initial default values if not explicitly initialized. In particular, all numeric types are initialized to zero, a boolean is initialized to false, and a character is initialized to the null character by default.

1.2 Classes and Objects

In more complex Java programs, the primary "actors" are ***objects***. Every object is an ***instance*** of a class, which serves as the ***type*** of the object and as a blueprint, defining the data which the object stores and the methods for accessing and modifying that data. The critical ***members*** of a class in Java are the following:

- ***Instance variables***, which are also called ***fields***, represent the data associated with an object of a class. Instance variables must have a ***type***, which can either be a base type (such as **int**, **float**, or **double**) or any class type (also known as a ***reference type*** for reasons we soon explain).
- ***Methods*** in Java are blocks of code that can be called to perform actions (similar to functions and procedures in other high-level languages). Methods can accept parameters as arguments, and their behavior may depend on the object upon which they are invoked and the values of any parameters that are passed. A method that returns information to the caller without changing any instance variables is known as an ***accessor method***, while an ***update method*** is one that may change one or more instance variables when called.

For the purpose of illustration, Code Fragment 1.2 provides a complete definition of a very simple class named Counter, to which we will refer during the remainder of this section.

```
1 public class Counter {
2   private int count;                                     // a simple integer instance variable
3   public Counter( ) { }                                  // default constructor (count is 0)
4   public Counter(int initial) { count = initial; }       // an alternate constructor
5   public int getCount( ) { return count; }               // an accessor method
6   public void increment( ) { count++; }                  // an update method
7   public void increment(int delta) { count += delta; }   // an update method
8   public void reset( ) { count = 0; }                    // an update method
9 }
```

Code Fragment 1.2: A Counter class for a simple counter, which can be queried, incremented, and reset.

This class includes one instance variable, named count, which is declared at line 2. As noted on the previous page, the count will have a default value of zero, unless we otherwise initialize it.

The class includes two special methods known as constructors (lines 3 and 4), one accessor method (line 5), and three update methods (lines 6–8). Unlike the original Universe class from page 2, our Counter class does not have a main method, and so it cannot be run as a complete program. Instead, the purpose of the Counter class is to create instances that might be used as part of a larger program.

1.2.1 Creating and Using Objects

Before we explore the intricacies of the syntax for our Counter class definition, we prefer to describe how Counter instances can be created and used. To this end, Code Fragment 1.3 presents a new class named CounterDemo.

```
1  public class CounterDemo {
2    public static void main(String[ ] args) {
3      Counter c;                    // declares a variable; no counter yet constructed
4      c = new Counter( );           // constructs a counter; assigns its reference to c
5      c.increment( );               // increases its value by one
6      c.increment(3);               // increases its value by three more
7      int temp = c.getCount( );     // will be 4
8      c.reset( );                   // value becomes 0
9      Counter d = new Counter(5);// declares and constructs a counter having value 5
10     d.increment( );               // value becomes 6
11     Counter e = d;                // assigns e to reference the same object as d
12     temp = e.getCount( );         // will be 6 (as e and d reference the same counter)
13     e.increment(2);               // value of e (also known as d) becomes 8
14   }
15 }
```

Code Fragment 1.3: A demonstration of the use of Counter instances.

There is an important distinction in Java between the treatment of base-type variables and class-type variables. At line 3 of our demonstration, a new variable c is declared with the syntax:

```
Counter c;
```

This establishes the identifier, c, as a variable of type Counter, but it does not create a Counter instance. Classes are known as ***reference types*** in Java, and a variable of that type (such as c in our example) is known as a ***reference variable***. A reference variable is capable of storing the location (i.e., ***memory address***) of an object from the declared class. So we might assign it to reference an existing instance or a newly constructed instance. A reference variable can also store a special value, **null**, that represents the lack of an object.

In Java, a new object is created by using the **new** operator followed by a call to a constructor for the desired class; a constructor is a method that always shares the same name as its class. The **new** operator returns a ***reference*** to the newly created instance; the returned reference is typically assigned to a variable for further use.

In Code Fragment 1.3, a new Counter is constructed at line 4, with its reference assigned to the variable c. That relies on a form of the constructor, Counter( ), that takes no arguments between the parentheses. (Such a zero-parameter constructor is known as a ***default constructor***.) At line 9 we construct another counter using a one-parameter form that allows us to specify a nonzero initial value for the counter.

Three events occur as part of the creation of a new instance of a class:

- A new object is dynamically allocated in memory, and all instance variables are initialized to standard default values. The default values are **null** for reference variables and 0 for all base types except **boolean** variables (which are **false** by default).
- The constructor for the new object is called with the parameters specified. The constructor may assign more meaningful values to any of the instance variables, and perform any additional computations that must be done due to the creation of this object.
- After the constructor returns, the **new** operator returns a reference (that is, a memory address) to the newly created object. If the expression is in the form of an assignment statement, then this address is stored in the object variable, so the object variable ***refers*** to this newly created object.

The Dot Operator

One of the primary uses of an object reference variable is to access the members of the class for this object, an instance of its class. That is, an object reference variable is useful for accessing the methods and instance variables associated with an object. This access is performed with the dot (".") operator. We call a method associated with an object by using the reference variable name, following that by the dot operator and then the method name and its parameters. For example, in Code Fragment 1.3, we call c.increment( ) at line 5, c.increment(3) at line 6, c.getCount( ) at line 7, and c.reset( ) at line 8. If the dot operator is used on a reference that is currently **null**, the Java runtime environment will throw a NullPointerException.

If there are several methods with this same name defined for a class, then the Java runtime system uses the one that matches the actual number of parameters sent as arguments, as well as their respective types. For example, our Counter class supports two methods named increment: a zero-parameter form and a one-parameter form. Java determines which version to call when evaluating commands such as c.increment( ) versus c.increment(3). A method's name combined with the number and types of its parameters is called a method's ***signature***, for it takes all of these parts to determine the actual method to perform for a certain method call. Note, however, that the signature of a method in Java does not include the type that the method returns, so Java does not allow two methods with the same signature to return different types.

A reference variable v can be viewed as a "pointer" to some object o. It is as if the variable is a holder for a remote control that can be used to control the newly created object (the device). That is, the variable has a way of pointing at the object and asking it to do things or give us access to its data. We illustrate this concept in Figure 1.2. Using the remote control analogy, a **null** reference is a remote control holder that is empty.

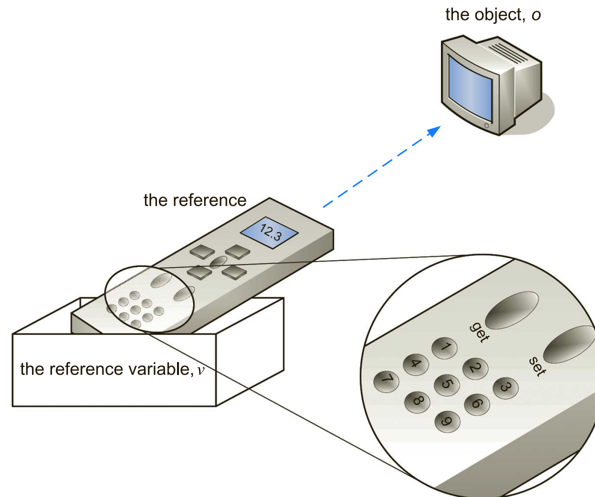


Figure 1.2: Illustrating the relationship between objects and object reference variables. When we assign an object reference (that is, memory address) to a reference variable, it is as if we are storing that object's remote control at that variable.

There can, in fact, be many references to the same object, and each reference to a specific object can be used to call methods on that object. Such a situation would correspond to our having many remote controls that all work on the same device. Any of the remotes can be used to make a change to the device (like changing a channel on a television). Note that if one remote control is used to change the device, then the (single) object pointed to by all the remotes changes. Likewise, if one object reference variable is used to change the state of the object, then its state changes for all the references to it. This behavior comes from the fact that there are many references, but they all point to the same object.

Returning to our CounterDemo example, the instance constructed at line 9 as

```
Counter d = new Counter(5);
```

is a distinct instance from the one identified as c. However, the command at line 11,

```
Counter e = d;
```

does not result in the construction of a new Counter instance. This declares a new *reference variable* named e, and assigns that variable a reference to the existing counter instance currently identified as d. At that point, both variables d and e are aliases for the same object, and so the call to d.getCount( ) behaves just as would e.getCount( ). Similarly, the call to update method e.increment(2) is affecting the same object identified by d.

It is worth noting, however, that the aliasing of two reference variables to the same object is not permanent. At any point in time, we may reassign a reference variable to a new instance, to a different existing instance, or to **null**.

1.2.2 Defining a Class

Thus far, we have provided definitions for two simple classes: the Universe class on page 2 and the Counter class on page 5. At its core, a class definition is a block of code, delimited by braces "{" and "}" , within which is included declarations of instance variables and methods that are the members of the class. In this section, we will undertake a deeper examination of class definitions in Java.

Modifiers

Immediately before the definition of a class, instance variable, or method in Java, keywords known as ***modifiers*** can be placed to convey additional stipulations about that definition.

Access Control Modifiers

The first set of modifiers we discuss are known as ***access control modifiers***, as they control the level of access (also known as ***visibility***) that the defining class grants to other classes in the context of a larger Java program. The ability to limit access among classes supports a key principle of object-orientation known as encapsulation (see Section 2.1). In general, the different access control modifiers and their meaning are as follows:

- The **public** class modifier designates that all classes may access the defined aspect. For example, line 1 of of Code Fragment 1.2 designates

  ```
  public class Counter {
  ```

  and therefore all other classes (such as CounterDemo) are allowed to construct new instances of the Counter class, as well as to declare variables and parameters of type Counter. In Java, each public class must be defined in a separate file named *classname*`.java`, where "*classname*" is the name of the class (for example, file `Counter.java` for the Counter class definition).

  The designation of **public** access for a particular *method* of a class allows any other class to make a call to that method. For example, line 5 of Code Fragment 1.2 designates

  ```
  public int getCount() { return count; }
  ```

  which is why the CounterDemo class may call c.getCount().

  If an instance variable is declared as public, dot notation can be used to directly access the variable by code in any other class that possesses a reference to an instance of this class. For example, were the count variable of Counter to be declared as public (which it is not), then the CounterDemo would be allowed to read or modify that variable using a syntax such as c.count.