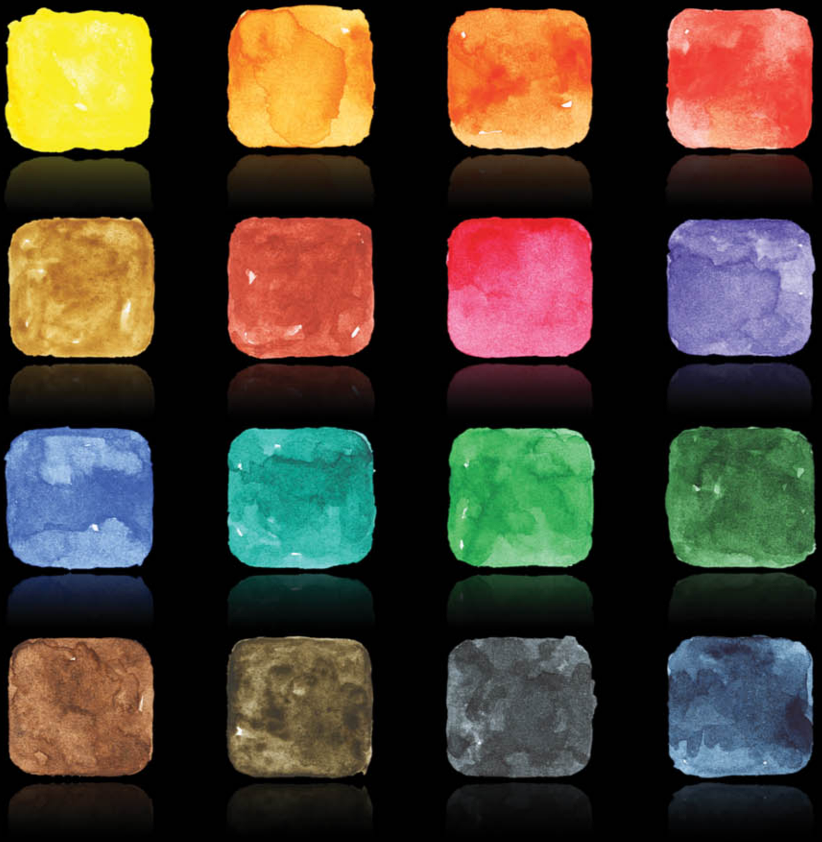
Data Structures and Abstractions with JAVA™

4th Edition



Frank M. Carrano • Timothy M. Henry

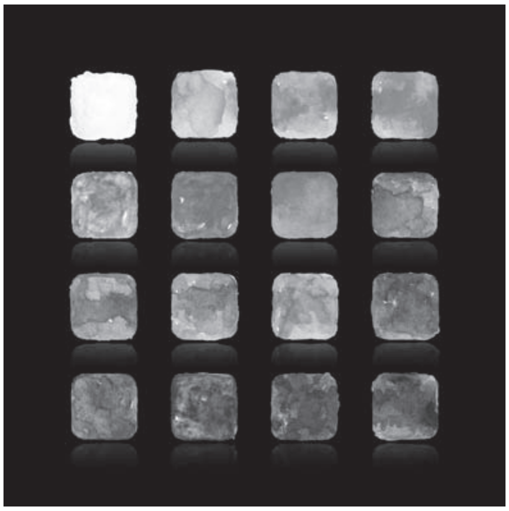
Reserved Words

Reserved words are also called **keywords**. You may not redefine any of these reserved words. Their meanings are determined by the Java language and cannot be changed. In particular, you cannot use any of these reserved words for variable names, method names, or class names.

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

Data Structures and Abstractions with Java™

Fourth Edition



Frank M. Carrano
University of Rhode Island
Timothy M. Henry
New England Institute of Technology

PEARSON

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

Editorial Director, ECS: Marcia Horton
Executive Editor: Tracy Johnson (Dunkelberger)
Editorial Assistant: Kelsey Loanes
Director of Marketing: Christy Lesko
Product Marketing Manager: Bram van Kempen
Field Marketing Manager: Demetrius Hall
Marketing Assistant: Jon Bryant
Director of Program Management: Erin Gregg
Program Management-Team Lead: Scott Disanno
Program Manager: Carole Snyder
Project Manager: Robert Engelhardt
Procurement Specialist: Maura Zaldivar-Garcia
Senior Art Director: Kathryn Foot
Cover Designer: Black Horse Designs
Cover Art: © Vector Illustration/Shutterstock
Permissions Supervisor: Rachel Youdelman
Permissions Administrator: William Opaluch
Web Development, Senior Manager: Steve Wright
Associate Web Developer: Barry Offringa
Full-Service Project Management: GEX Publishing Services

Library of Congress Cataloging-in-Publication Data

Carrano, Frank M.
Data structures and abstractions with Java / Frank M. Carrano, University of Rhode Island. — Fourth edition.
pages cm
Includes bibliographical references and index.
ISBN 978-0-13-374405-7 (alk. paper)
1. Data structures (Computer science) 2. Java (Computer program language) I. Title.
QA76.9.D33C37 2015
005.13'3--dc23
2014025945

10 9 8 7 6 5 4 3 2 1

PEARSON
www.pearsonhighered.com

ISBN 10: 0-13-374405-1
ISBN 13: 978-0-13-374405-7

Welcome to the fourth edition of *Data Structures and Abstractions with Java*, a book for an introductory course in data structures, typically known as CS-2.

I wrote this book with you in mind—whether you are an instructor or a student—based upon my experiences during more than three decades of teaching undergraduate computer science. I wanted my book to be reader friendly so that students could learn more easily and instructors could teach more effectively. To this end, you will find the material covered in small pieces—I call them "segments"—that are easy to digest and facilitate learning. Numerous examples that mimic real-world situations provide a context for the new material and help to make it easier for students to learn and retain abstract concepts. Many simple figures illustrate and clarify complicated ideas. Included are over 60 video tutorials to supplement the instruction and help students when their instructor is unavailable.

I am pleased and excited to welcome my co-author and colleague, Dr. Timothy Henry, to this edition. Together we have given a fresh update to this work, while retaining the topics and order of the previous edition. You will find a greater emphasis on our design decisions for both specifications and implementations of the various data structures, as well as a new introduction to safe and secure programming practices. The new features in this edition are given on the next page.

We hope that you enjoy reading this book. Like many others before you, you can learn—or teach—data structures in an effective and sustainable way.

Warm regards,
Frank M. Carrano

We are always available to connect with instructors and students who use our books. Your comments, suggestions, and corrections will be greatly appreciated. Here are a few ways to reach us:

E-mail:	`carrano@acm.org` or `thenry@neit.edu`
Facebook:	`www.facebook.com/makingitreal`
Twitter:	`twitter.com/Frank_M_Carrano`
Website:	`frank-m-carrano.com/makingitreal`

making it real
Frank M. Carrano

Organization and Structure

This book's organization, sequencing, and pace of topic coverage make learning and teaching easier by focusing your attention on one concept at a time, by providing flexibility in the order in which you can cover topics, and by clearly distinguishing between the specification and implementation of abstract data types, or ADTs. To accomplish these goals, we have organized the material into 29 chapters, composed of small, numbered segments that deal with one concept at a time. Each chapter focuses on either the specification and use of an ADT or its various implementations. You can choose to cover the specification of an ADT followed by its implementations, or you can treat the specification and use of several ADTs before you consider any implementation issues. The book's organization makes it easy for you to choose the topic order that you prefer.

Table of Contents at a Glance

The following brief table of contents shows the overall composition of the book. Notice the new Prelude and nine Java Interludes. Further details—including a chapter-by-chapter description—are given later in this preface. Note that some of the appendixes and the glossary are available online.

What's New?

While the chapters are in the same order and cover the same topics as in the previous edition, reader feedback convinced us to move some material from the appendixes or online into the main portion of the book. Other changes are motivated by reader suggestions and our own desire to improve the presentation. Here are the significant changes in this edition:

- A new Prelude follows the Introduction and precedes Chapter 1 to discuss how to design classes. This material was in Appendix D of the previous edition.
- Relevant aspects of Java have been extracted from either the appendixes or the chapters themselves and placed into new Java Interludes that occur throughout the book and as needed. By doing so, we increase the distinction and separation between concepts and Java-specific issues. The titles of these interludes follow, and you can see their placement between chapters on the previous page:

Java Interlude 1	Generics
Java Interlude 2	Exceptions
Java Interlude 3	More About Generics
Java Interlude 4	More About Exceptions
Java Interlude 5	Iterators
Java Interlude 6	Mutable and Immutable Objects
Java Interlude 7	Inheritance
Java Interlude 8	Generics Once Again
Java Interlude 9	Cloning

- Safe and secure programming is a new topic that is introduced in Chapter 2, discussed in new Security Notes, and reflected in the Java code that implements the ADTs.
- Beginning with stacks in Chapter 5, most ADT methods now indicate failure by throwing an exception. Methods only return `null` when it cannot be a data value within a collection.
- Expanded coverage of generics treats generic methods and bounded types.
- Immutable, mutable, and cloneable objects are covered in Java Interludes instead of the online Chapter 30 of the previous edition.
- Additional Design Decisions continue to present the options one has when specifying and implementing particular ADTs and provide the rationale behind our choices.
- Illustrations have been revised to show objects specifically instead of as values within nodes or array elements.
- Vector-based implementations of the ADT list and queue are no longer covered, but are left as programming projects.
- Line numbers appear in program listings.
- Java code is Java 8 compliant.
- Supplements now include a test bank.

Here are the significant changes to specific chapters:

- Chapter 1 introduces the ADT set in addition to the bag.
- Chapter 2 introduces safe and secure programming. The code changes suggested here are integrated into all ADT implementations in subsequent chapters.
- Chapters 5 and 6 use exceptions in the specification and implementations of the ADT stack.
- Chapters 8 and 9 replace some Java code for sorting methods with pseudocode.

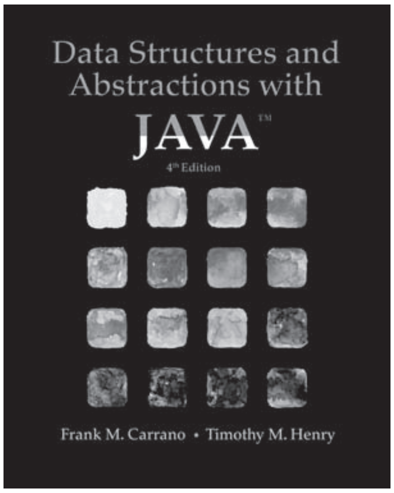
- Chapters 10 and 11 use exceptions in the specification and implementations of the ADTs queue, deque, and priority queue.
- Chapter 11 no longer covers the vector-based implementation of the ADT queue; it is left as a programming project.
- Chapters 12, 13, and 14 use exceptions in the specification and implementations of the ADT list.
- Chapter 13 changes the array-based implementation of the ADT list by ignoring the array element at index 0. The vector-based implementation of the ADT list is no longer covered, but is left as a programming project.
- Chapter 15 covers only iterators for the ADT list. The concepts of an iterator in Java are treated in the preceding Java Interlude 5 instead of in this chapter.
- Chapter 20 no longer covers the vector-based implementation of the ADT dictionary; it is left as a programming project.
- Chapter 23 defines balanced binary trees, which previously was in Chapter 25.
- Chapter 24 no longer defines an interface for a binary node, and the class `BinaryNode` no longer implements one.

A Note to Students

The topics that we cover in this book deal with the various ways of organizing data so that a given application can access and manipulate data in an efficient way. These topics are fundamental to your future study of computer science, as they provide you with the foundation of knowledge required to create complex and reliable software. Whether you are interested in designing video games or software for robotic controlled surgery, the study of data structures is vital to your success. Even if you do not study all of the topics in this book now, you are likely to encounter them later. We hope that you will enjoy reading the book, and that it will serve as a useful reference tool for your future courses.

After looking over this preface, you should read the Introduction. There you will quickly see what this book is about and what you need to know about Java before you begin. The Prelude discusses class design and the use of Java interfaces. We use interfaces throughout the book. Appendixes A through E review `javadoc` comments, Java basics, classes, inheritance, and files. New Java Interludes occur throughout the book and cover advanced aspects of Java as they are needed. Note that inside the front and back covers you will find Java's reserved words, its primitive data types, the precedence of its operators, and a list of Unicode characters.

Please be sure to browse the rest of this preface to see the features that will help you in your studies.


Data Structures and Abstractions with
JAVA™
4th Edition
Frank M. Carrano • Timothy M. Henry

Features to Enhance Learning

Each chapter begins with a table of contents, a list of prerequisite portions of the book that you should have read, and the learning objectives for the material to be covered. Other pedagogical elements appear throughout the book, as follows:

Notes Important ideas are presented or summarized in highlighted paragraphs and are meant to be read in line with the surrounding text.

Security Notes Aspects of safe and secure programming are introduced and highlighted in this new feature.

A Problem Solved Large examples are presented in the form of "A Problem Solved," in which a problem is posed and its solution is discussed, designed, and implemented.

Design Decisions To give readers insight into the design choices that one could make when formulating a solution, "Design Decision" elements lay out such options, along with the rationale behind the choice made for a particular example. These discussions are often in the context of one of the "A Problem Solved" examples.

Examples Numerous examples illuminate new concepts.

Programming Tips Suggestions to improve or facilitate programming are presented as soon as they become relevant.

SELF-TEST

Self-Test Questions Questions are posed throughout each chapter, integrated within the text, that reinforce the concept just presented. These "self-test" questions help readers to understand the material, since answering them requires pause and reflection. Solutions to these questions are provided at the end of each chapter.

VideoNote

VideoNotes Online tutorials are a Pearson feature that provides visual and audio support to the presentation given throughout the book. They offer students another way to recap and reinforce key concepts. VideoNotes allow for self-paced instruction with easy navigation, including the ability to select, play, rewind, fast-forward, and stop within each video. Unique VideoNote icons appear throughout this book whenever a video is available for a particular concept or problem. A detailed list of the VideoNotes for this text and their associated locations in the book can be found on page xxvi. VideoNotes are free with the purchase of a new textbook. To purchase access to VideoNotes, please go to

```
pearsonhighered.com/carrano
```

Exercises and Programming Projects Further practice is available by solving the exercises and programming projects at the end of each chapter. Unfortunately, we cannot give readers the answers to these exercises and programming projects, even if they are not enrolled in a class. Only instructors who adopt the book can receive selected answers from the publisher. For help with these exercises and projects, you will have to contact your instructor.

Accessing Instructor and Student Resource Materials

The following items are available on the publisher's website at `pearsonhighered.com/carrano`:

- Java code as it appears in the book
- A link to any misprints that have been discovered since the book was published
- Links to additional online content, which is described next

Instructor Resources

The following protected material is available to instructors who adopt this book by logging onto Pearson's Instructor Resource Center, accessible from `pearsonhighered.com/carrano`:

- PowerPoint lecture slides
- Solutions to exercises and projects
- Test bank
- Instructor source code
- Figures from the book

Additionally, instructors can access the book's Companion Website for the following online premium content, also accessible from `pearsonhighered.com/carrano`:

- Instructional VideoNotes
- Appendixes B, C, and E
- A glossary of terms

Please contact your Pearson sales representative for an instructor access code. Contact information is available at `pearsonhighered.com/replocator`.

Student Resources

The following material is available to students by logging onto the Companion Website accessible from `pearsonhighered.com/carrano`:

- Instructional VideoNotes
- Appendixes B, C, and E
- A glossary of terms

Students must use the access card located in the front of the book to register for and then enter the Companion Website. Students without an access code can purchase access from the Companion Website by following the instructions listed there.

Note that the Java Class Library is available at `docs.oracle.com/javase/8/docs/api/`.

Content Overview

Readers of this book should have completed a programming course, preferably in Java. The appendixes cover the essentials of Java that we assume readers will know. You can use these appendixes as a review or as the basis for making the transition to Java from another programming language. The book itself begins with the Introduction, which sets the stage for the data organizations that we will study.

- **Prelude:** At the request of readers of the previous edition, we have moved the introduction to class design from the appendix to the beginning of the book. Most of the material that was in Appendix D of the third edition is now in the Prelude, which follows the Introduction.
- **Chapters 1 through 3:** We introduce the bag as an abstract data type (ADT). By dividing the material across several chapters, we clearly separate the specification, use, and implementation of the bag. For example, Chapter 1 specifies the bag and provides several examples of its use. This chapter also introduces the ADT set. Chapter 2 covers implementations that use arrays, while Chapter 3 introduces chains of linked nodes and uses one in the definition of a class of bags.

  In a similar fashion, we separate specification from implementation throughout the book when we discuss various other ADTs. You can choose to cover the chapters that specify and use the ADTs and then later cover the chapters that implement them. Or you can cover the chapters as they appear, implementing each ADT right after studying its specification and use. A list of chapter prerequisites appears later in this preface to help you plan your path through the book.

  Chapter 2 does more than simply implement the ADT bag. It shows how to approach the implementation of a class by initially focusing on core methods. When defining a class, it is often useful to implement and test these core methods first and to leave definitions of the other methods for later. Chapter 2 also introduces the concept of safe and secure programming, and shows how to add this protection to your code.
- **Java Interludes 1 and 2:** The first Java interlude introduces generics, so that we can use it with our first ADT, the bag. This interlude immediately follows Chapter 1. Java Interlude 2 introduces exceptions and follows Chapter 2. We apply this material, which was formerly in an appendix, to the implementations of the ADT bag.
- **Chapter 4:** Here we introduce the complexity of algorithms, a topic that we integrate into future chapters.
- **Chapters 5 and 6:** Chapter 5 discusses stacks, giving examples of their use, and Chapter 6 implements the stack using an array, a vector, and a chain.
- **Chapter 7:** Next, we present recursion as a problem-solving tool and its relationship to stacks. Recursion, along with algorithm efficiency, is a topic that is revisited throughout the book.
- **Java Interlude 3:** This interlude provides the Java concepts needed for the sorting methods that we are about to present. It introduces the standard interface `Comparable`, generic methods, bounded type parameters, and wildcards.
- **Chapters 8 and 9:** The next two chapters discuss various sorting techniques and their relative complexities. We consider both iterative and recursive versions of these algorithms.
- **Java Interlude 4:** This Java interlude shows how the programmer can write new exception classes. In doing so, it shows how to extend an existing class of exceptions. It also introduces the `finally` block.
- **Chapters 10 and 11:** Chapter 10 discusses queues, deques, and priority queues, and Chapter 11 considers their implementations. It is in this latter chapter that we introduce circularly linked and doubly linked chains. Chapter 11 also uses the programmer-defined class `EmptyQueueException`.
- **Chapters 12, 13, and 14**: The next three chapters introduce the ADT list. We discuss this collection abstractly and then implement it by using an array and a chain of linked nodes.
- **Java Interlude 5 and Chapter 15:** The coverage of Java iterators that was formerly in Chapter 15 now appears before the chapter in Java Interlude 5. Included are the standard interfaces `Iterator`,

`Iterable`, and `ListIterator`. Chapter 15 then shows ways to implement an iterator for the ADT list. It considers and implements Java's iterator interfaces `Iterator` and `ListIterator`.

- **Java Interlude 6**: This interlude discusses mutable and immutable objects, material that previously was in the online Chapter 30.
- **Chapters 16 and 17 and Java Interlude 7:** Continuing the discussion of a list, Chapter 16 introduces the sorted list, looking at two possible implementations and their efficiencies. Chapter 17 shows how to use the list as a superclass for the sorted list and discusses the general design of a superclass. Although inheritance is reviewed in Appendix D, the relevant particulars of inheritance—including protected access, abstract classes, and abstract methods—are presented in Java Interlude 7 just before Chapter 17.
- **Chapter 18:** We then examine some strategies for searching an array or a chain in the context of a list or a sorted list. This discussion is a good basis for the sequence of chapters that follows.
- **Java Interlude 8:** Before we get to the next chapter, we quickly cover in this interlude situations where more than one generic data type is necessary.
- **Chapters 19 through 22:** Chapter 19 covers the specification and use of the ADT dictionary. Chapter 20 presents implementations of the dictionary that are linked or that use arrays. Chapter 21 introduces hashing, and Chapter 22 uses hashing as a dictionary implementation.
- **Chapters 23 and 24 and Java Interlude 9:** Chapter 23 discusses trees and their possible uses. Included among the several examples of trees is an introduction to the binary search tree and the heap. Chapter 24 considers implementations of the binary tree and the general tree. Java Interlude 9 discusses cloning, a topic that was previously online. We clone an array, a chain of linked nodes, and a binary node. We also investigate a sorted list of clones. Although this material is important, you can treat it as optional, as it is not required in the following chapters.
- **Chapters 25 through 27:** Chapter 25 focuses on the implementation of the binary search tree. Chapter 26 shows how to use an array to implement the heap. Chapter 27 introduces balanced search trees. Included in this chapter are the AVL, 2-3, 2-4, and red-black trees, as well as B-trees.
- **Chapters 28 and 29:** Finally, we discuss graphs and look at several applications and two implementations.
- **Appendixes A through E:** The appendixes provide supplemental coverage of Java. As we mentioned earlier. Appendix A considers programming style and comments. It introduces `javadoc` comments and defines the tags that we use in this book. Appendix B reviews Java up to but not including classes. However, this appendix also covers the `Scanner` class, enumerations, boxing and unboxing, and the for-each loop. Appendix C discusses Java classes, Appendix D expands this topic by looking at composition and inheritance, and Appendix E discusses files.

Acknowledgments

Our sincere appreciation and thanks go to the following reviewers for carefully reading the previous edition and making candid comments and suggestions that greatly improved the work:

Tony Allevato—*Virginia Polytechnic Institute and State University*
Mary Boelk—*Marquette University*
Suzanne Buchele—*Southwestern University*
Kevin Buffardi—*Virginia Polytechnic Institute and State University*
Jose Cordova—*University of Louisiana at Monroe*
Greg Gagne—*Westminster College*
Victoria Hilford—*University of Houston*
Jim Huggins—*Kettering University*
Shamim Kahn—*Columbus State University*
Kathy Liszka—*University of Akron*
Eli Tilevich—*Virginia Polytechnic Institute and State University*
Jianhua Yang—*Columbus State University*
Michelle Zhu—*Southern Illinois University*

Special thanks go to our support team at Pearson Education Computer Science during the lengthy process of revising this book: Executive Editor Tracy Dunkelberger, Program Manager Carole Snyder, Program Management-Team Leader Scott Disanno, and Project Manager Bob Engelhardt have always be a great help to us in completing our projects. Our long-time copy editor, Rebecca Pepper, ensured that the presentation is clear, correct, and grammatical. Thank you so much!

Our gratitude for the previously mentioned people does not diminish our appreciation for the help provided by many others. Steve Armstrong produced the lecture slides for this edition and previous editions of the book. Professor Charles Hoot of the Oklahoma City University created the lab manual, Professor Kathy Liszka from the University of Akron created the new collection of test questions, and Jesse Grabowski provided the solutions to many of the programming projects. Thank you again to the reviewers of the previous editions of the book:

Reviewers for the third edition:

Steven Andrianoff—*St. Bonaventure University*
Brent Baas—*LeTourneau University*
Timothy Henry—*New England Institute of Technology*
Ken Martin—*University of North Florida*
Bill Siever—*Northwest Missouri State University*
Lydia Sinapova—*Simpson College*
Lubomir Stanchev—*Indiana University*
Judy Walters—*North Central College*
Xiaohui Yuan—*University of North Texas*

Reviewers for the second edition:

Harold Anderson—*Marist College*
Razvan Andonie—*Central Washington University*
Tom Blough—*Rensselaer Polytechnic Institute*
Chris Brooks—*University of San Francisco*
Adrienne Decker—*University at Buffalo, SUNY*

Henry Etlinger—*Rochester Institute of Technology*
Derek Harter—*Texas A&M University*
Timothy Henry—*New England Institute of Technology*
Robert Holloway—*University of Wisconsin, Madison*
Charles Hoot—*Oklahoma City University*
Teresa Leyk—*Texas A&M University*
Robert McGlinn—*Southern Illinois University, Carbondale*
Edward Medvid—*Marymount University*
Charles Metzler—*City College of San Francisco*
Daniel Zeng—*University of Arizona*

Reviewers for the first edition:

David Boyd—*Valdosta State University*
Dennis Brylow—*Purdue University*
Michael Croswell—*Industry trainer/consultant*
Matthew Dickerson—*Middlebury College*
Robert Holloway—*University of Wisconsin, Madison*
John Motil—*California State University, Northridge*
Bina Ramamurthy—*University at Buffalo, SUNY*
David Surma—*Valparaiso University*

We continue to appreciate the many others who helped during previous editions. They include Alan Apt, James Blanding, Lianne Dunn, Mike Giacobbe, Toni Holm, Charles Hoot, Brian Jepson, Rose Kernan, Christianna Lee, Patrick Lindner, John Lovell, Vince O'Brien, Patty Roy, Walt Savitch, Ben Schomp, Heather Scott, Carole Snyder, Chirag Thakkar, Camille Trentacoste, Nate Walker, and Xiaohong Zhu.

Finally, we thank our families and friends—Doug, Joanne, Tita, Bobby, Ted, Nancy, Sue, Tom, Maybeth, Marge, and Lorraine—for giving us lives away from computers.

Thank you, everyone, for your expertise and good cheer.

Frank M. Carrano
Timothy M. Henry

This page intentionally left blank

Contents

VideoNotes Directory

VideoNote

This table lists the VideoNotes that are available online. The page numbers indicate where in the book each VideoNote has relevance.

Chapter Prerequisites

Each chapter and appendix assumes that the reader has studied certain previous material. This list indicates those prerequisites. Numbers represent chapter numbers, letters reference appendixes, and "JI" precedes each interlude number. You can use this information to plan a path through the book.

		Prerequisites
Prelude	Designing Classes	A, B, C, D
Chapter 1	Bags	Prelude, D
Java Interlude 1	Generics	Prelude
Chapter 2	Bag Implementations That Use Arrays	Prelude, 1
Java Interlude 2	Exceptions	B, C, D
Chapter 3	A Bag Implementation That Links Data	1, 2, JI2
Chapter 4	The Efficiency of Algorithms	2, 3, C
Chapter 5	Stacks	Prelude, 1, JI2
Chapter 6	Stack Implementations	2, 3, 4, 5
Chapter 7	Recursion	2, 3, 4, 5, C
Java Interlude 3	More About Generics	JI1
Chapter 8	An Introduction to Sorting	3, 4, 7, JI3
Chapter 9	Faster Sorting Methods	4, 7, 8, JI3
Java Interlude 4	More About Exception	D, JI2
Chapter 10	Queues, Deques, and Priority Queues	Prelude, 5, 8
Chapter 11	Queue, Deque, and Priority Queue Implementations	2, 3, 6, 10
Chapter 12	Lists	Prelude, 6, C, JI2, JI3
Chapter 13	List Implementations That Use Arrays	Prelude, 2, 4, 12
Chapter 14	A List Implementation That Links Data	3, 11, 12, 13
Java Interlude 5	Iterators	12, JI2
Chapter 15	Iterators	13, 14, JI5
Java Interlude 6	Mutable and Immutable Objects	12, D
Chapter 16	Sorted Lists	4, 7, 12, 14
Java Interlude 7	Inheritance and Polymorphism	Prelude, 6, D
Chapter 17	Inheritance and Lists	12, 13, 14, 16, D, JI7
Chapter 18	Searching	4, 7, 12, 13, 14, 16
Java Interlude 8	Generics Once Again	C, JI3
Chapter 19	Dictionaries	12, 15, 18, JI5, JI8
Chapter 20	Dictionary Implementations	3, 4, 12, 13, 14, 18, 19, JI5
Chapter 21	Introducing Hashing	19, 20

		Prerequisites
Chapter 22	Hashing as a Dictionary Implementation	4, 13, 14, 19, 20, 21, JI5
Chapter 23	Trees	5, 7, 14, 18, JI5
Chapter 24	Tree Implementations	5, 10, 14, 23, D, JI2
Java Interlude 9	Cloning	16, 24, C, D, JI3, JI6
Chapter 25	A Binary Search Tree Implementation	7, 19, 23, 24, D
Chapter 26	A Heap Implementation	2, 13, 23
Chapter 27	Balanced Search Trees	23, 24, 25
Chapter 28	Graphs	5, 10, 23
Chapter 29	Graph Implementations	5, 10, 12, 15, 19, 23, 28, JI5
Appendix A	Documentation and Programming Style	Some knowledge of Java
Appendix B	Java Essentials	Programming knowledge
Appendix C	Java Classes	B
Appendix D	Creating Classes from Other Classes	C
Appendix E	File Input and Output	Prelude, B, JI2

Introduction

Organizing Data

Look around and you will see ways that people organize things. When you stopped at the store this morning, you went to the back of a line to wait for the cashier. The line organized people chronologically. The first person in the line was the first to be served and to leave the line. Eventually, you reached the front of the line and left the store with a bag containing your purchases. The items in the bag were in no particular order, and some of them were the same.

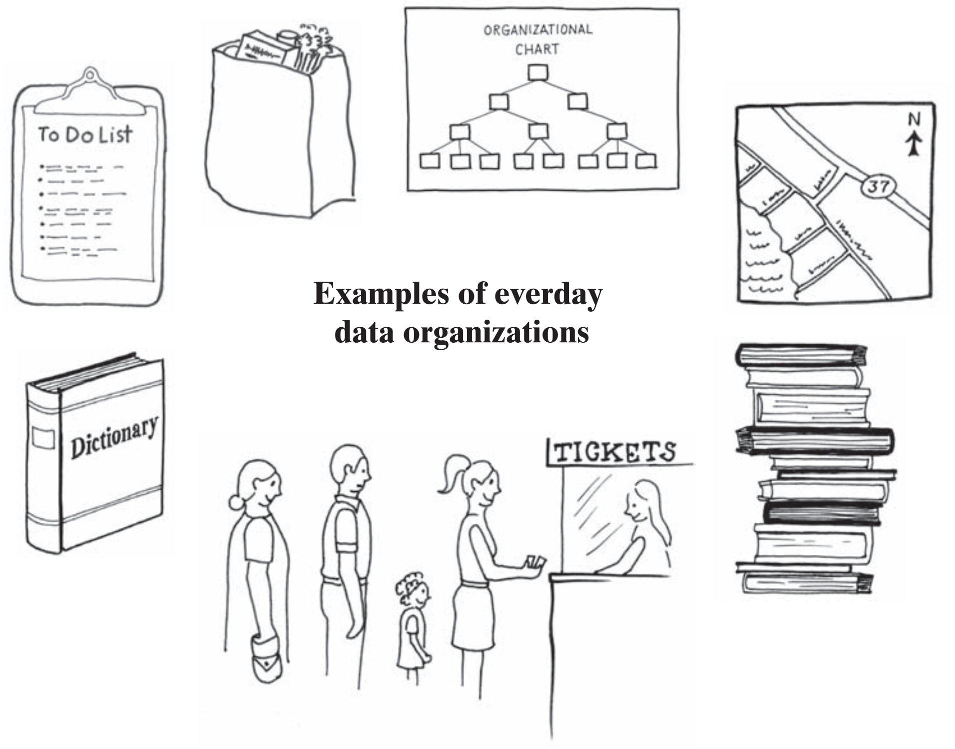
Do you see a stack of books or a pile of papers on your desk? It's easy to look at or remove the top item of the stack or to add a new item to the top of the stack. The items in a stack also are organized chronologically, with the item added most recently on top and the item added first on the bottom.

At your desk, you see your to-do list. Each entry in the list has a position that might or might not be important to you. You may have written them either as you thought of them, in their order of importance, or in alphabetical order. You decide the order; the list simply provides places for your entries.

Your dictionary is an alphabetical list of words and their definitions. You search for a word and get its definition. If your dictionary is printed, the alphabetical organization helps you to locate a word quickly. If your dictionary is computerized, its alphabetical organization is hidden, but it still speeds the search.

Speaking of your computer, you have organized your files into folders, or directories. Each folder contains several other folders or files. This type of organization is hierarchical. If you drew a picture of it, you would get something like a family tree or a chart of a company's internal departments. These data organizations are similar and are called trees.

Finally, notice the road map that you are using to plan your weekend trip. The diagram of roads and towns shows you how to get from one place to another. Often, several ways are possible. One way might be shorter, another faster. The map has an organization known as a graph.



Examples of everday data organizations

Computer programs also need to organize their data. They do so in ways that parallel the examples we just cited. That is, programs can use a stack, a list, a dictionary, and so on. These ways of organizing data are represented by abstract data types. An **abstract data type**, or **ADT**, is a specification that describes a data set and the operations on that data. Each ADT specifies what data is stored and what the operations on the data do. Since an ADT does not indicate how to store the data or how to implement the operations, we can talk about ADTs independently of any programming language. In contrast, a **data structure** is an implementation of an ADT within a programming language.

A **collection** is a general term for an ADT that contains a group of objects. Some collections allow duplicate items, some do not. Some collections arrange their contents in a certain order, while others do not.

We might create an ADT **bag** consisting of an unordered collection that allows duplicates. It is like a grocery bag, a lunch bag, or a bag of potato chips. Suppose you remove one chip from a bag of chips. You don't know when the chip was placed into the bag. You don't know whether the bag contains another chip shaped exactly like the one you just removed. But you don't really care. If you did, you wouldn't store your chips in a bag!

A bag does not order its contents, but sometimes you do want to order things. ADTs can order their items in a variety of ways. The ADT **list**, for example, simply numbers its items. A list, then, has a first item, a second item, and so on. Although you can add an item to the end of a list, you can also insert an item at the beginning of the list or between existing items. Doing so renumbers the items after the new item. Additionally, you can remove an item at a particular position within a list. Thus, the position of an item in the list does not necessarily indicate when it was added. Notice that the list does not decide where an item is placed; you make this decision.

In contrast, the ADTs **stack** and **queue** order their items chronologically. When you remove an item from a stack, you remove the one that was added most recently. When you remove an item from a queue, you remove the one that was added the earliest. Thus, a stack is like a pile of books. You can remove the top book or add another book to the top of the pile. A queue is like a line of people. People leave a line from its front and join it at its end.

Some ADTs maintain their entries in sorted order, if the items can be compared. For instance, strings can be organized in alphabetical order. When you add an item to the ADT **sorted list**, for example, the ADT determines where to place the item in the list. You do not indicate a position for the item, as you would with the ADT list.

The ADT **dictionary** contains pairs of items, much as a language dictionary contains a word and its definition. In this example, the word serves as a **key** that is used to locate the entries. Some dictionaries sort their entries and some do not.

The ADT **tree** organizes its entries according to some hierarchy. For example, in a family tree, people are associated with their children and their parents. The ADT **binary search tree** has a combined hierarchical and sorted organization that makes locating a particular entry easier.

The ADT **graph** is a generalization of the ADT tree that focuses on the relationship among its entries instead of any hierarchical organization. For example, a road map is a graph that shows the existing roads and distances between towns.

This book shows you how to use and implement these data organizations. Throughout the book, we've assumed that you already know Java. If you need a refresher, you will find the appendixes helpful. Appendix A gives an overview of writing comments suitable for `javadoc`. Appendix B reviews the basic statements in Java. Appendix C discusses the fundamental construction of classes and methods, and Appendix D covers the essentials of composition and inheritance. Finally, Appendix E presents reading and writing external files. Appendixes B, C, and E are on the book's website (see page ix of the Preface). You can download them and refer to the material as needed. Special sections throughout the book, called Java Interludes, focus on relevant aspects of Java that might be new to you, including how to handle exceptions. The Prelude, which follows, discusses how to design classes, specify methods, and write Java interfaces. Using interfaces and writing comments to specify methods are essential to our presentation of ADTs.

This page intentionally left blank

Prelude

Designing Classes

Contents

Prerequisites

Appendix A	Documentation and Programming Style
Appendix B	Java Basics (Online)
Appendix C	Java Classes (Online)
Appendix D	Creating Classes from Other Classes

Object-oriented programming embodies three design concepts: encapsulation, inheritance, and polymorphism. If you are not familiar with inheritance and polymorphism, please review Appendixes B, C, and D. Here we will discuss encapsulation as a way to hide the details of an implementation during the design of a class. We emphasize the importance both of specifying how a method should behave before you implement it and of expressing your specifications as comments in your program.

We introduce Java interfaces as a way to separate the declarations of a class's behavior from its implementation. Finally, we present, at an elementary level, some techniques for identifying the classes necessary for a particular solution.

Encapsulation

P.1 What is the most useful description of an automobile, if you want to learn to drive one? It clearly is not a description of how its engine goes through a cycle of taking in air and gasoline, igniting the gasoline/air mixture, and expelling exhaust. Such details are unnecessary when you want to learn to drive. In fact, such details can get in your way. If you want to learn to drive an automobile, the most useful description of an automobile has such features as the following:

- If you press your foot on the accelerator pedal, the automobile will move faster.
- If you press your foot on the brake pedal, the automobile will slow down and eventually stop.
- If you turn the steering wheel to the right, the automobile will turn to the right.
- If you turn the steering wheel to the left, the automobile will turn to the left.

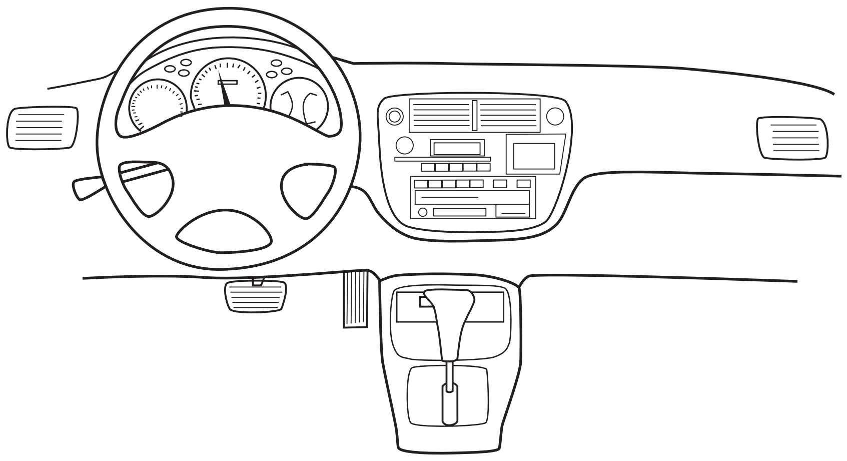
Just as you need not tell somebody who wants to drive a car how the engine works, you need not tell somebody who uses a piece of software all the fine details of its Java implementation. Likewise, suppose that you create a software component for another programmer to use in a program. You should describe the component in a way that tells the other programmer how to use it but that spares the programmer all the details of how you wrote the software.

P.2 **Encapsulation** is one of the design principles of object-oriented programming. The word "encapsulation" sounds as though it means putting things into a capsule, and that image is indeed correct. Encapsulation hides the fine detail of what is inside the "capsule." For this reason, encapsulation is often called **information hiding**. But not everything should be hidden. In an automobile, certain things are visible—like the pedals and steering wheel—and others are hidden under the hood. In other words, the automobile is encapsulated so that the details are hidden, and only the controls needed to drive the automobile are visible, as Figure P-1 shows. Similarly, you should encapsulate your Java code so that details are hidden and only the necessary controls are visible.

Encapsulation encloses data and methods within a class and hides the implementation details that are not necessary for using the class. If a class is well designed, its use does not require an understanding of its implementation. A programmer can use the class's methods without knowing the details of how they are coded. The programmer must know only how to provide a method with appropriate arguments, leaving the method to perform the right action. Stated simply, the programmer need not worry about the internal details of the class definition. The programmer who uses encapsulated software to write more software has a simpler task. As a result, software is produced more quickly and with fewer errors.

Note: **Encapsulation** is a design principle of object-oriented programming that encloses data and methods within a class, thereby hiding the details of a class's implementation. A programmer receives only enough information to be able to use the class. A well-designed class can be used as though the body of every method was hidden from view.

FIGURE P-1 An automobile's controls are visible to the driver, but its inner workings are hidden



P.3 **Abstraction** is a process that asks you to focus on *what* instead of *how*. When you design a class, you practice **data abstraction**. You focus on what you want to do with or to the data without worrying about how you will accomplish these tasks and how you will represent the data. Abstraction asks you to focus on what data and operations are important. When you abstract something, you identify the central ideas. For example, an abstract of a book is a brief description of the book, as opposed to the entire book.

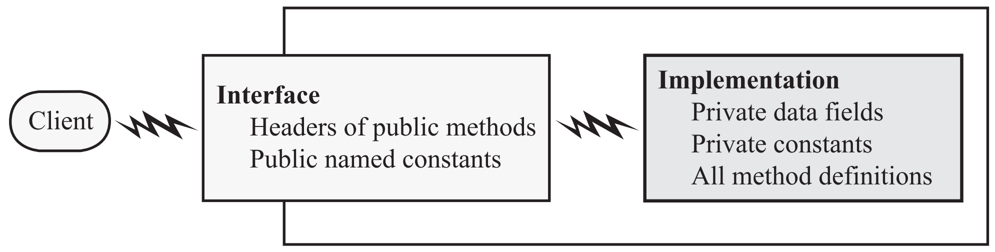
When designing a class, you should not think about any method's implementation. That is, you should not worry about *how* the class's methods will accomplish their goals. This separation of specification from implementation allows you to concentrate on fewer details, thereby making your task easier and less error-prone. Detailed, well-planned specifications facilitate an implementation that is more likely to be successful.

Note: The process of abstraction asks you to focus on *what* instead of *how*.

P.4 When done correctly, encapsulation divides a class definition into two parts, which we will call the **client interface** and the **implementation**. The client interface describes everything a programmer needs to know to use the class. It consists of the headers for the public methods of the class, the comments that tell a programmer how to use these public methods, and any publicly defined constants of the class. The client interface part of the class definition should be all you need to know to use the class in your program.

The implementation consists of all data fields and the definitions of all methods, including those that are public, private, and protected. Although you need the implementation to run a client (a program that uses the class), you should not need to know anything about the implementation to write the client. Figure P-2 illustrates an encapsulated implementation of a class and the client interface. Although the implementation is hidden from the client, the interface is visible and provides a well-regulated means for the client to communicate with the implementation.

FIGURE P-2 An interface provides well-regulated communication between a hidden implementation and a client



The client interface and implementation are not separated in the definition of a Java class. They are mixed together. You can, however, create a separate Java interface as a companion to your class. A later section of this prelude describes how to write such an interface, and we will write several of them in this book.

? SELF-TEST

Question 1 How does a client interface differ from a class implementation?

Question 2 Think of an example, other than an automobile, that illustrates encapsulation. What part of your example corresponds to a client interface and what part to an implementation?

Specifying Methods

Separating the purpose of a class and its methods from their implementations is vital to a successful software project. You should specify what each class and method does without concern for its implementation. Writing descriptions enables you to capture your ideas initially and to develop them so that they are clear enough to implement. Your written descriptions should reach the point where they are useful as comments in your program. You need to go beyond a view that sees comments as something you add after you write the program to satisfy an instructor or boss.

Comments

Let's focus on comments that you write for a class's methods. Although organizations tend to have their own style for comments, the developers of Java have specified a commenting style that you should follow. If you include comments written in this style in your program, you can run a utility program called `javadoc` to produce documents that describe your classes. This documentation tells people what they need to know to use your class but omits all the implementation details, including the bodies of all method definitions.

The program `javadoc` extracts the header for your class, the headers for all public methods, and comments that are written in a certain form. Each such comment must appear immediately before a public class definition or the header of a public method and must begin with `/**` and end with `*/`. Certain **tags** that begin with the symbol `@` appear within the comments to identify various aspects of the method. For example, you use `@param` to identify a parameter, `@return` to identify a return value, and `@throws` to indicate an exception that the method throws. You will see some examples of these tags within the comments in this prelude. Appendix A provides the details for writing comments acceptable to `javadoc`.

Rather than talk further about the rules for `javadoc` here, we want to discuss some important aspects of specifying a method. First, you need to write a concise statement of the method's purpose or task. Beginning this statement with a verb will help you to avoid many extra words that you really do not need.

In thinking about a method's purpose, you should consider its input parameters, if any, and describe them. You also need to describe the method's results. Does it return a value, does it cause some action, or does it affect the state of an argument? In writing such descriptions, you should keep in mind the following ideas.

Preconditions and Postconditions

P.5 A **precondition** is a statement of the conditions that must be true before a method begins execution. The method should not be used, and cannot be expected to perform correctly, unless the precondition is satisfied. A precondition can be related to the description of a method's parameters. For example, a method that computes the square root of x can have $x \geq 0$ as a precondition.

A **postcondition** is a statement of what is true after a method completes its execution, assuming that the precondition was met. For a valued method, the postcondition will describe the value returned by the method. For a void method, the postcondition will describe actions taken and any changes to the calling object. In general, the postcondition describes all the effects produced by a method invocation.

Thinking in terms of a postcondition can help you to clarify a method's purpose. Notice that going from precondition to postcondition leaves out the *how*—that is, we separate the method's specification from its implementation.

Programming Tip: A method that cannot satisfy its postcondition, even though its precondition is met, can throw an exception. (See Java Interludes 2 and 4 for a discussion of exceptions.)

P.6 **Responsibility.** A precondition implies responsibility for guaranteeing that certain conditions are met. If the client is responsible for meeting the conditions before calling the method, the method need not check the conditions. On the other hand, if the method is responsible for enforcing the conditions, the client does not check them. A clear statement of who must check a given set of conditions increases the probability that someone will do so and avoids duplication of effort.

For example, you could specify the square root method that we mentioned in the previous segment by writing the following comments before its header:

```
/** Computes the square root of a number.
    @param x  A real number >= 0.
    @return  The square root of x.
*/
```

In this case, the method assumes that the client will provide a nonnegative number as an argument.

A safer technique is to make the method assume responsibility for checking the argument. In that case, its comments could read as follows:

```
/** Computes the square root of a number.
    @param x  A real number.
    @return  The square root of x if x >= 0.
    @throws  ArithmeticException if x < 0.
*/
```

Although we've integrated the precondition and postcondition into the previous comments, we could instead identify them separately.