Data Structures

Abstraction and Design Using Java

Third Edition

ELLIOT B. KOFFMAN AND PAUL A. T. WOLFGANG

DATA STRUCTURES

Abstraction and Design Using Java

THIRD EDITION

ELLIOT B. KOFFMAN
Temple University

PAUL A. T. WOLFGANG
Temple University

WILEY

VICE PRESIDENT & DIRECTOR	Laurie Rosatone
SENIOR DIRECTOR	Don Fowley
EXECUTIVE EDITOR	Brian Gambrel
DEVELOPMENT EDITOR	Jennifer Lartz
ASSISTANT	Jessy Moor
PROJECT MANAGER	Gladys Soto
PROJECT SPECIALIST	Nichole Urban
PROJECT ASSISTANT	Anna Melhorn
MARKETING MANAGER	Dan Sayre
ASSISTANT MARKETING MANAGER	Puja Katarawala
ASSOCIATE DIRECTOR	Kevin Holm
SENIOR CONTENT SPECIALIST	Nicole Repasky
PRODUCTION EDITOR	Rajeshkumar Nallusamy
PHOTO RESEARCHER	Amanda Bustard
COVER PHOTO CREDIT	© Robert Davies/Shutterstock

This book was set in 10/12 pt SabonLTStd-Roman by SPiGlobal and printed and bound by Lightning Source Inc.

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 © 2016, 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 (Web site: 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, or online at: 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 shipping 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: 978-1-119-23914-7 (PBK)
ISBN: 978-1-119-22307-8 (EVALC)

Library of Congress Cataloging-in-Publication Data

Koffman, Elliot B.
[Objects, abstraction, data structures and design using Java]
Data structures : abstraction and design using Java / Elliot B. Koffman, Temple University, Paul A.T. Wolfgang, Temple University. — Third edition.
pages cm
Original edition published under title: Objects, abstraction, data structures and design using Java.
Includes index.
ISBN 978-1-119-23914-7 (pbk.) 1. Data structures (Computer science) 2. Java (Computer program language) 3. Object-oriented programming (Computer science) 4. Application program interfaces (Computer software) I. Wolfgang, Paul A. T. II. Title.

QA76.9.D35K58 2016
005.7'3—dc23

2015036861

Printing identification and country of origin will either be included on this page and/or the end of the book. In addition, if the ISBN on this page and the back cover do not match, the ISBN on the back cover should be considered the correct ISBN.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Preface

Our goal in writing this book was to combine a strong emphasis on problem solving and software design with the study of data structures. To this end, we discuss applications of each data structure to motivate its study. After providing the specification (interface) and the implementation (a Java class), we then cover case studies that use the data structure to solve a significant problem. Examples include maintaining an ordered list, evaluating arithmetic expressions using a stack, finding the shortest path through a maze, and Huffman coding using a binary tree and a priority queue. In the implementation of each data structure and in the solutions of the case studies, we reinforce the message "Think, then code" by performing a thorough analysis of the problem and then carefully designing a solution (using pseudo-code and UML class diagrams) before the implementation. We also provide a performance analysis when appropriate. Readers gain an understanding of why different data structures are needed, the applications they are suited for, and the advantages and disadvantages of their possible implementations.

Intended Audience

This book was written for anyone with a curiosity or need to know about data structures, those essential elements of good programs and reliable software. We hope that the text will be useful to readers with either professional or educational interests.

It is intended as a textbook for the second programming course in a computing curriculum involving the study of data structures, especially one that emphasizes Object-Oriented Design (OOD). The text could also be used in a more-advanced course in algorithms and data structures. Besides coverage of the basic data structures and algorithms (lists, stacks, queues, trees, recursion, sorting), there are chapters on sets and maps, balanced binary search trees, graphs, and an online appendix on event-oriented programming. Although we expect that most readers will have completed a first programming course in Java, there is an extensive review chapter (included as an appendix) for those who may have taken a first programming course in a different language, or for those who need a refresher in Java.

Emphasis on the Java Collections Framework

The book focuses on the interfaces and classes in the Java Collections Framework. We begin the study of a new data structure by specifying an abstract data type as an interface, which we adapt from the Java API. Readers are encouraged throughout the text to use the Java Collections Framework as a resource for their programming.

Our expectation is that readers who complete this book will be familiar with the data structures available in the Java Collections Framework and will be able to use them in their future programming. However, we also expect that they will want to know how the data structures are implemented, so we provide thorough discussions of classes that implement these data structures. Each class follows the approach taken by the Java designers where appropriate. However, when their industrial-strength solutions appear to be too complicated for beginners to understand, we have provided simpler implementations but have tried to be faithful to their approach.

Think, then Code

To help you "Think, then code" we discuss problem solving and introduce appropriate software design tools throughout the textbook. For example, Chapter 1 focuses on OOD and Class Hierarchies. It introduces the Uniform Modeling Language (also covered in Appendix B) to document an OOD. It introduces the use of interfaces to specify abstract data types and to facilitate contract programming and describes how to document classes using Javadoc-style comments. There is also coverage of exceptions and exception handling. Chapter 2 introduces the Java Collections Framework and focuses on the List interface, and it shows how to use big-O notation to analyze program efficiency. In Chapter 3, we cover different testing strategies in some detail including a discussion of test-driven design and the use of the JUnit program to facilitate testing.

Features of the Third Edition

We had two major goals for the third edition. The first was to bring the coverage of Java up to Java 8 by introducing new features of Java where appropriate. For example, we use the Java 7 diamond operator when creating new **Collection** objects. We use the Java 8 `StringJoiner` in place of the older `StringBuilder` for joining strings.

A rather significant change was to introduce Java 8 lambda expressions and functional interfaces as a way to facilitate functional programming in Java in a new Section 6.4. Using these features significantly improved the code.

The second major goal was to provide additional emphasis on testing and debugging. To facilitate this, we moved our discussion of testing and debugging from an appendix to Chapter 3 and expanded our coverage of testing including more discussion of JUnit. We also added a new section that introduced test-driven development.

A third goal was to ease the transition to Java for Python programmers. When introducing Java data structures (for example, arrays, lists, sets, and maps), we compared them to equivalent Python data structures.

Other changes to the text included reorganizing the chapter on lists and moving the discussion of algorithm analysis to the beginning of the chapter so that big-O notation could be used to compare the efficiency of different List implementations. We also combined the chapters on stacks and queues and increased our emphasis on using Deque as an alternative to the legacy Stack class. We also added a discussion of Timsort, which is used in Java 8, to the chapter on sorting algorithms. Finally, some large case studies and an appendix were moved to online supplements.

Case Studies

We illustrate OOD principles in the design and implementation of new data structures and in the solution of approximately 20 case studies. Case studies follow a five-step process (problem specification, analysis, design, implementation, and testing). As is done in industry, we sometimes perform these steps in an iterative fashion rather than in strict sequence. Several case studies have extensive discussions of testing and include methods that automate the testing process. Some case studies are revisited in later chapters, and solutions involving different data structures are compared. We also provide additional case studies on the Web site for the textbook (www.wiley.com/college/koffman), including one that illustrates a solution to the same problem using several different data structures.

Prerequisites

Our expectation is that the reader will be familiar with the Java primitive data types including `int`, `boolean`, `char`, and `double`; control structures including `if`, case, `while`, `for`, and `try-catch`; the `String` class; the one-dimensional array; input/output using either `JOptionPane` dialog windows or text streams (class `Scanner` or `BufferedReader`) and console input/output. For those readers who lack some of the concepts or who need some review, we provide complete coverage of these topics in Appendix A. Although labeled an Appendix, the review chapter provides full coverage of the background topics and has all the pedagogical features (discussed below) of the other chapters. We expect most readers will have some experience with Java programming, but someone who knows another programming language should be able to undertake the book after careful study of the review chapter. We do not require prior knowledge of inheritance, wrapper classes, or `ArrayLists` as we cover them in the book (Chapters 1 and 2).

Pedagogy

The book contains the following pedagogical features to assist inexperienced programmers in learning the material.

- **Learning objectives** at the beginning of each chapter tell readers what skills they should develop.
- **Introductions** for each chapter help set the stage for what the chapter will cover and tie the chapter contents to other material that they have learned.
- **Case Studies** emphasize problem solving and provide complete and detailed solutions to real-world problems using the data structures studied in the chapter.
- **Chapter Summaries** review the contents of the chapter.
- **Boxed Features** emphasize and call attention to material designed to help readers become better programmers.

  Pitfall boxes help readers identify common problems and how to avoid them.

  Design Concept boxes illuminate programming design decisions and trade-offs.

  Programming Style boxes discuss program features that illustrate good programming style and provide tips for writing clear and effective code.

  Syntax boxes are a quick reference for the Java structures being introduced.

- **Self-Check and Programming Exercises** at the end of each section provide immediate feedback and practice for readers as they work through the chapter.
- **Quick-Check, Review Exercises, and Programming Projects** at the end of each chapter review chapter concepts and give readers a variety of skill-building activities, including longer projects that integrate chapter concepts as they exercise the use of data structures.

Theoretical Rigor

In Chapter 2, we discuss algorithm efficiency and big-O notation as a measure of algorithm efficiency. We have tried to strike a balance between pure "hand waving" and extreme rigor when determining the efficiency of algorithms. Rather than provide several paragraphs of formulas, we have provided simplified derivations of algorithm efficiency using big-O notation. We feel this will give readers an appreciation of the performance of various algorithms and methods and the process one follows to determine algorithm efficiency without bogging them down in unnecessary detail.

Overview of the book

Chapter 1 introduces Object Oriented Programming, inheritance, and class hierarchies including interfaces and abstract classes. We also introduce UML class diagrams and Javadoc-style documentation. The `Exception` class hierarchy is studied as an example of a Java class hierarchy.

Chapter 2 introduces the Java Collections Framework as the foundation for the traditional data structures. These are covered in separate chapters: lists (Chapter 2), stacks, queues and deques (Chapter 4), Trees (Chapters 6 and 9), Sets and Maps (Chapter 7), and Graphs (Chapter 10). Each new data structure is introduced as an abstract data type (ADT). We provide a specification of each ADT in the form of a Java interface. Next, we implement the data structure as a class that implements the interface. Finally, we study applications of the data structure by solving sample problems and case studies.

Chapter 3 covers different aspects of testing (e.g. top-down, bottom-up, white-box, black-box). It includes a section on developing a JUnit test harness and also a section on Test-Driven Development. It also discuses using a debugger to help find and correct errors.

Chapter 4 discusses stacks, queues, and deques. Several applications of these data structures are provided.

Chapter 5 covers recursion so that readers are prepared for the study of trees, a recursive data structure. This chapter could be studied earlier. There is an optional section on list processing applications of recursion that may be skipped if the chapter is covered earlier.

Chapter 6 discusses binary trees, including binary search trees, heaps, priority queues, and Huffman trees. It also shows how Java 8 lambda expressions and functional interfaces support functional programming.

Chapter 7 covers the Set and Map interfaces. It also discusses hashing and hash tables and shows how a hash table can be used in an implementation of these interfaces. Building an index for a file and Huffman Tree encoding and decoding are two case studies covered in this chapter.

Chapter 8 covers various sorting algorithms including mergesort, heapsort, quicksort and Timsort.

Chapter 9 covers self-balancing search trees, focusing on algorithms for manipulating them. Included are AVL and Red-Black trees, 2-3 trees, 2-3-4 trees, B-trees, and skip-lists.

Chapter 10 covers graphs. We provide several well-known algorithms for graphs, including Dijkstra's shortest path algorithm and Prim's minimal spanning tree algorithm. In most programs, the last few chapters would be covered in a second course in algorithms and data structures.

Supplements and Companion Web Sites

The following supplementary materials are available on the Instructor's Companion Web Site for this textbook at www.wiley.com/college/koffman. Items marked for students are accessible on the Student Companion Web Site at the same address.

- Additional homework problems with solutions
- Additional case studies, including one that illustrates a solution to the same problem using several different data structures
- Source code for all classes in the book (for students and instructors)
- PowerPoint slides
- Electronic test bank for instructors
- Solutions to end-of-section odd-numbered self-check and programming exercises (for students)
- Solutions to all exercises for instructors
- Solutions to chapter-review exercises for instructors
- Sample programming project solutions for instructors
- Additional homework and laboratory projects, including cases studies and solutions

Acknowledgments

Many individuals helped us with the preparation of this book and improved it greatly. We are grateful to all of them. These include students at Temple University who have used notes that led to the preparation of this book in their coursework, and who class-tested early drafts of the book. We would like to thank Rolf Lakaemper and James Korsh, colleagues at Temple University, who used earlier editions in their classes. We would also like to thank a former Temple student, Michael Mayle, who provided preliminary solutions to many of the exercises.

We would also like to acknowledge support from the National Science Foundation (grant number DUE-1225742) and Principal Investigator Peter J. Clarke, Florida International University (FIU), to attend the Fifth Workshop on Integrating Software Testing into Programming Courses (WISTPC 2014) at FIU. Some of the testing methodologies discussed at the workshop were integrated into the chapter on Testing and Debugging.

We are especially grateful to our reviewers who provided invaluable comments that helped us correct errors in each version and helped us set our revision goals for the next version. The individuals who reviewed this book are listed below.

Reviewers

Sheikh Iqbal Ahamed, *Marquette University*
Justin Beck, *Oklahoma State University*
John Bowles, *University of South Carolina*
Mary Elaine Califf, *Illinois State University*
Tom Cortina, *SUNY Stony Brook*
Adrienne Decker, *SUNY Buffalo*
Chris Dovolis, *University of Minnesota*
Vladimir Drobot, *San Jose State University*
Kenny Fong, *Southern Illinois University, Carbondale*
Ralph Grayson, *Oklahoma State University*
Allan M. Hart, *Minnesota State University, Mankato*
James K. Huggins, *Kettering University*
Chris Ingram, *University of Waterloo*
Gregory Kesden, *Carnegie Mellon University*
Sarah Matzko, *Clemson University*
Lester McCann, *University of Arizona*

Ron Metoyer, *Oregon State University*
Rich Pattis, *Carnegie Mellon University*
Thaddeus F. Pawlicki, *University of Rochester*
Sally Peterson, *University of Wisconsin—Madison*
Salam N. Salloum, *California State Polytechnic University, Pomona*
Mike Scott, *University of Texas—Austin*
Victor Shtern, *Boston University*
Mark Stehlik, *Carnegie Mellon University*
Ralph Tomlinson, *Iowa State University*
Frank Tompa, *University of Waterloo*
Renee Turban, *Arizona State University*
Paul Tymann, *Rochester Institute of Technology*
Karen Ward, *University of Texas—El Paso*
Jim Weir, *Marist College*
Lee Wittenberg, *Kean University*
Martin Zhao, *Mercer University*

Although all the reviewers provided invaluable suggestions, we do want to give special thanks to Chris Ingram who reviewed every version of the first edition of the manuscript, including the preliminary pages for the book. His care, attention to detail, and dedication helped us improve this book in many ways, and we are very grateful for his efforts.

Besides the principal reviewers, there were a number of faculty members who reviewed sample pages of the first edition online and made valuable comments and criticisms of its content. We would like to thank those individuals, listed below.

Content Connections Online Review

Razvan Andonie, *Central Washington University*
Antonia Boadi, *California State University Dominguez Hills*
Mikhail Brikman, *Salem State College*
Robert Burton, *Brigham Young University*
Chakib Chraibi, *Barry University*
Teresa Cole, *Boise State University*
Jose Cordova, *University of Louisiana Monroe*
Joyce Crowell, *Belmont University*
Robert Franks, *Central College*
Barabra Gannod, *Arizona State University East*
Wayne Goddard, *Clemson University*
Simon Gray, *College of Wooster*
Wei Hu, *Houghton College*
Edward Kovach, *Franciscan University of Steubenville*
Saeed Monemi, *California Polytechnic and State University*
Robert Noonan, *College of William and Mary*

Kathleen O'Brien, *Foothill College*
Rathika Rajaravivarma, *Central Connecticut State University*
Sam Rhoads, *Honolulu Community College*
Vijayakumar Shanmugasundaram, *Concordia College Moorhead*
Gene Sheppard, *Perimeter College*
Linda Sherrell, *University of Memphis*
Meena Srinivasan, *Mary Washington College*
David Weaver, *Sheperd University*
Stephen Weiss, *University of North Carolina—Chapel Hill*
Glenn Wiggins, *Mississippi College*
Bruce William, *California State University Pomona*

Finally, we want to acknowledge the participants in focus groups for the second programming course organized by John Wiley & Sons at the Annual Meeting of the SIGCSE Symposium in March 2004. They reviewed the preface, table of contents, and sample chapters and also provided valuable input on the book and future directions of the course.

Focus Group

Claude Anderson, *Rose-Hulman Institute of Technology*
Jay M. Anderson, *Franklin & Marshall University*
John Avitabile, *College of Saint Rose*
Cathy Bishop-Clark, *Miami University—Middletown*
Debra Burhans, *Canisius College*
Michael Clancy, *University of California—Berkeley*
Nina Cooper, *University of Nevada Las Vegas*
Kossi Edoh, *Montclair State University*
Robert Franks, *Central College*
Evan Golub, *University of Maryland*
Graciela Gonzalez, *Sam Houston State University*
Scott Grissom, *Grand Valley State University*
Jim Huggins, *Kettering University*
Lester McCann, *University of Wisconsin—Parkside*
Briana Morrison, *Southern Polytechnic State University*
Judy Mullins, *University of Missouri—Kansas City*
Roy Pargas, *Clemson University*
J.P. Pretti, *University of Waterloo*
Reza Sanati, *Utah Valley State College*
Barbara Smith, *University of Dayton*
Suzanne Smith, *East Tennessee State University*
Michael Stiber, *University of Washington, Bothell*
Jorge Vasconcelos, *University of Mexico (UNAM)*
Lee Wittenberg, *Kean University*

We would also like to acknowledge and thank the team at John Wiley & Sons who were responsible for the management of this edition and ably assisted us with all phases of the book development and production. They were Gladys Soto, Project Manager, Nichole Urban, Project Specialist, and Rajeshkumar Nallusamy, Production Editor.

We would like to acknowledge the help and support of our colleague Frank Friedman who also read an early draft of this textbook and offered suggestions for improvement. Frank and Elliot began writing textbooks together many years ago and Frank has had substantial influence on the format and content of these books. Frank also influenced Paul to begin his teaching career as an adjunct faculty member and then hired him as a full-time faculty member when he retired from industry. Paul is grateful for his continued support.

Finally, we would like to thank our wives who provided us with comfort and support through this arduous process. We very much appreciate their understanding and their sacrifices that enabled us to focus on this book, often during time we would normally be spending with them. In particular, Elliot Koffman would like to thank

Caryn Koffman

and Paul Wolfgang would like to thank

Sharon Wolfgang

Contents

Chapter 1

Object-Oriented Programming and Class Hierarchies

Chapter Objectives

- To learn about interfaces and their role in Java
- To understand inheritance and how it facilitates code reuse
- To understand how Java determines which method to execute when there are multiple methods with the same name in a class hierarchy
- To become familiar with the Exception hierarchy and the difference between checked and unchecked exceptions
- To learn how to define and use abstract classes as base classes in a hierarchy
- To learn the role of abstract data types and how to specify them using interfaces
- To study class **`Object`** and its methods and to learn how to override them
- To become familiar with a class hierarchy for shapes
- To understand how to create packages and to learn more about visibility

This chapter describes important features of Java that support Object-Oriented Programming (OOP). Object-oriented languages allow you to build and exploit hierarchies of classes in order to write code that may be more easily reused in new applications. You will learn how to extend an existing Java class to define a new class that inherits all the attributes of the original, as well as having additional attributes of its own. Because there may be many versions of the same method in a class hierarchy, we show how polymorphism enables Java to determine which version to execute at any given time.

We introduce interfaces and abstract classes and describe their relationship with each other and with actual classes. We introduce the abstract class `Number`. We also discuss class `Object`, which all classes extend, and we describe several of its methods that may be used in classes you create.

As an example of a class hierarchy and OOP, we describe the `Exception` class hierarchy and explain that the Java Virtual Machine (JVM) creates an `Exception` object whenever an error occurs during program execution. Finally, you will learn how to create packages in Java and about the different kinds of visibility for instance variables (data fields) and methods.

Inheritance and Class Hierarchies

1.1 ADTs, Interfaces, and the Java API

In earlier programming courses, you learned how to write individual classes consisting of attributes and methods (operations). You also learned how to use existing classes (e.g., `String` and `Scanner`) to facilitate your programming. These classes are part of the Java Application Programming Interface (API).

One of our goals is to write code that can be reused in many different applications. One way to make code reusable is to encapsulate the data elements together with the methods that operate on that data. A new program can then use the methods to manipulate an object's data without being concerned about details of the data representation or the method implementations. The encapsulated data together with its methods is called an abstract data type (ADT).

FIGURE 1.1
Diagram of an ADT

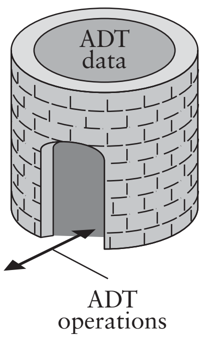


Figure 1.1 shows a diagram of an ADT. The data values stored in the ADT are hidden inside the circular wall. The bricks around this wall are used to indicate that these data values cannot be accessed except by going through the ADT's methods.

A class provides one way to implement an ADT in Java. If the data fields are private, they can be accessed only through public methods. Therefore, the methods control access to the data and determine the manner in which the data is manipulated.

Another goal of this text is to show you how to write and use ADTs in programming. As you progress through this book, you will create a large collection of ADT implementations (classes) in your own program library. You will also learn about ADTs that are available for you to use through the Java API.

Our principal focus will be on ADTs that are used for structuring data to enable you to more easily and efficiently store, organize, and process information. These ADTs are often called *data structures*. We introduce the Java Collections Framework (part of the Java API), which provides implementation of these common data structures, in Chapter 2 and study it throughout the text. Using the classes that are in the Java Collections Framework will make it much easier for you to design and implement new application programs.

Interfaces

A Java interface is a way to specify or describe an ADT to an applications programmer. An interface is like a contract that tells the applications programmer precisely what methods are available and describes the operations they perform. It also tells the applications programmer what arguments, if any, must be passed to each method and what result the method will return. Of course, in order to make use of these methods, someone else must have written a class that *implements the interface* by providing the code for these methods.

The interface tells the coder precisely what methods must be written, but it does not provide a detailed algorithm or prescription for how to write them. The coder must "program to the interface," which means he or she must develop the methods described in the interface without variation. If each coder does this job well, that ensures that other programmers can use the completed class exactly as it is written, without needing to know the details of how it was coded.

There may be more than one way to implement the methods; hence, several classes may implement the interface, but each must satisfy the contract. One class may be more efficient than the others at performing certain kinds of operations (e.g., retrieving information from a database), so that class will be used if retrieval operations are more likely in a particular application. The important point is that the particular implementation that is used will not affect other classes that interact with it because every implementation satisfies the contract.

Besides providing the complete definition (implementation) of all methods declared in the interface, each implementer of an interface may declare data fields and define other methods not in the interface, including constructors. An interface cannot contain constructors because it cannot be instantiated—that is, one cannot create objects, or instances, of it. However, it can be represented by instances of classes that implement it.

EXAMPLE 1.1 An automated teller machine (ATM) enables a user to perform certain banking operations from a remote location. It must support the following operations.

1. Verify a user's Personal Identification Number (PIN).
2. Allow the user to choose a particular account.
3. Withdraw a specified amount of money.
4. Display the result of an operation.
5. Display an account balance.

A class that implements an ATM must provide a method for each operation. We can write this requirement as the interface `ATM` and save it in file `ATM.java`, shown in Listing 1.1. The keyword `interface` on the header line indicates that an interface is being declared. If you are unfamiliar with the documentation style shown in this listing, read about Java documentation at the end of Section A.7 in Appendix A.

LISTING 1.1
Interface `ATM.java`

```
/** The interface for an ATM. */
public interface ATM {

    /** Verifies a user's PIN.
        @param pin The user's PIN
        @return Whether or not the User's PIN is verified
     */
    boolean verifyPIN(String pin);

    /** Allows the user to select an account.
        @return a String representing the account selected
     */
```

```
    String selectAccount();

    /** Withdraws a specified amount of money
        @param account The account from which the money comes
        @param amount The amount of money withdrawn
        @return Whether or not the operation is successful
     */
    boolean withdraw(String account, double amount);

    /** Displays the result of an operation
        @param account The account for the operation
        @param amount The amount of money
        @param success Whether or not the operation was successful
     */
    void display(String account, double amount, boolean success);

    /** Displays the result of a PIN verification
        @param pin The user's pin
        @param success Whether or not the PIN was valid
     */
    void display(String pin, boolean success);

    /** Displays an account balance
        @param account The account selected
     */
    void showBalance(String account);
}
```

The interface definition shows the heading only for several methods. Because only the headings are shown, they are considered *abstract methods*. Each actual method with its body must be defined in a class that implements the interface. Therefore, a class that implements this interface must provide a `void` method called `verifyPIN` with an argument of type `String`. There are also two display methods with different signatures. The first is used to display the result of a withdrawal, and the second is used to display the result of a PIN verification. The keywords `public abstract` are optional (and usually omitted) in an interface because all interface methods are public abstract by default.

SYNTAX Interface Definition

FORM:

```
public interface interfaceName {
    abstract method declarations
    constant declarations
}
```

EXAMPLE:

```
public interface Payable {
    public abstract double calcSalary();
    public abstract boolean salaried();
    public static final double DEDUCTIONS = 25.5;
}
```

MEANING:

Interface *interfaceName* is defined. The interface body provides headings for abstract methods and constant declarations. Each abstract method must be defined in a class that implements the interface. Constants defined in the interface (e.g., `DEDUCTIONS`) are accessible in classes that implement the interface or in the same way as static fields and methods in classes (see Section A.4).

NOTES:

The keywords `public` and `abstract` are implicit in each abstract method declaration, and the keywords `public static final` are implicit in each constant declaration. We show them in the example here, but we will omit them from now on.

Java 8 also allows for static and default methods in interfaces. They are used to add features to existing classes and interfaces while minimizing the impact on existing programs. We will discuss default and static methods when describing where they are used in the API.

The `implements` Clause

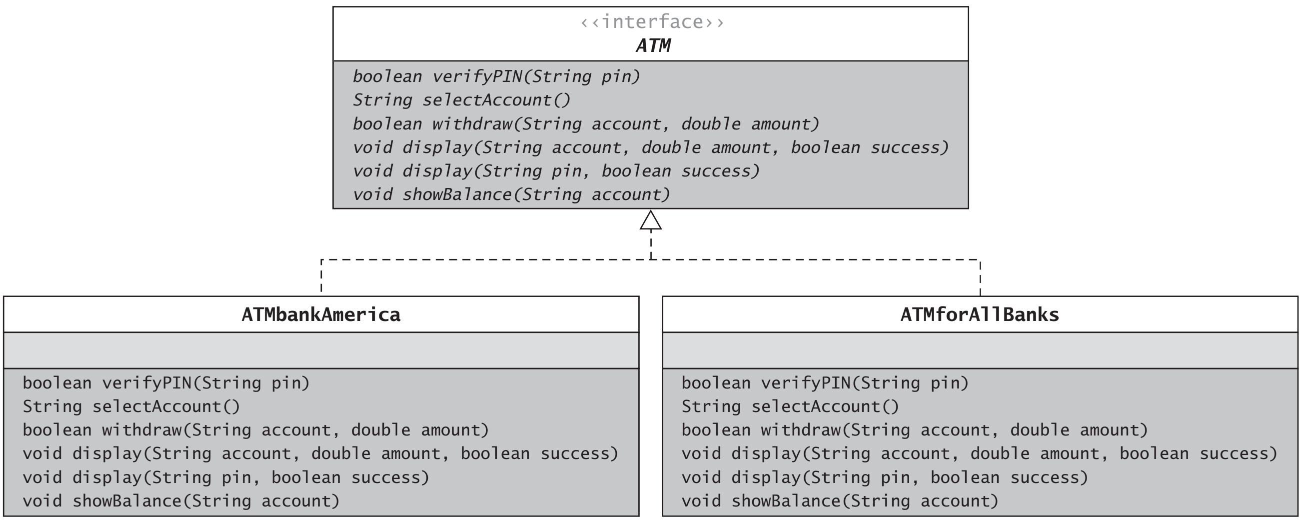
The class headings for two classes that implement interface ATM are

```
public class ATMbankAmerica implements ATM
public class ATMforAllBanks implements ATM
```

Each class heading ends with the clause `implements ATM`. When compiling these classes, the Java compiler will verify that they define the required methods in the way specified by the interface. If a class implements more than one interface, list them all after `implements`, with commas as separators.

Figure 1.2 is a UML (Unified Modeling Language) diagram that shows the ATM interface and these two implementing classes. Note that a dashed line from the class to the interface is used to indicate that the class implements the interface. We will use UML diagrams throughout this text to show relationships between classes and interfaces. Appendix B provides detailed coverage of UML diagrams.

FIGURE 1.2
UML Diagram Showing the ATM Interface and Its Implementing Classes



PITFALL

Not Properly Defining a Method to Be Implemented

If you neglect to define method `verifyPIN` in class `ATMforAllBanks` or if you use a different method signature, you will get the following syntax error:

```
class ATMforAllBanks should be declared abstract; it does not define method
verifyPIN(String) in interface ATM.
```

The above error indicates that the method `verifyPin` was not properly defined. Because it contains an abstract method that is not defined, Java incorrectly believes that `ATM` should be declared to be an abstract class. If you use a result type other than `boolean`, you will also get a syntax error.

PITFALL

Instantiating an Interface

An interface is not a class, so you cannot instantiate an interface. The statement

```
ATM anATM = new ATM();  // invalid statement
```

will cause the following syntax error:

```
interface ATM is abstract; cannot be instantiated.
```

Declaring a Variable of an Interface Type

In the previous programming pitfall, we mentioned that you cannot instantiate an interface. However, you may want to declare a variable that has an interface type and use it to reference an actual object. This is permitted if the variable references an object of a class type that implements the interface. After the following statements execute, variable `ATM1` references an `ATMbankAmerica` object, and variable `ATM2` references an `ATMforAllBanks` object, but both `ATM1` and `ATM2` are type `ATM`.

```
ATM ATM1 = new ATMbankAmerica();  // valid statement
ATM ATM2 = new ATMforAllBanks();  // valid statement
```

EXERCISES FOR SECTION 1.1

SELF-CHECK

1. What are the two parts of an ADT? Which part is accessible to a user and which is not? Explain the relationships between an ADT and a class, between an ADT and an interface, and between an interface and classes that implement the interface.
2. Correct each of the following statements that is incorrect, assuming that class `PDGUI` and class `PDConsoleUI` implement interface `PDUserInterface`.
   a. `PDGUI p1 = new PDConsoleUI();`
   b. `PDGUI p2 = new PDUserInterface();`

c. `PDUserInterface p3 = new PDUserInterface();`

d. `PDUserInterface p4 = new PDConsoleUI();`

e. `PDGUI p5 = new PDUserInterface();`
   `PDUserInterface p6 = p5;`

f. `PDUserInterface p7;`
   `p7 = new PDConsoleUI();`

3. Explain how an interface is like a contract.
4. What are two different uses of the term *interface* in programming?

PROGRAMMING

1. Define an interface named `Resizable` with just one abstract method, `resize`, that is a `void` method with no parameter.
2. Write a Javadoc comment for the following method of a class `Person`. Assume that class `Person` has two `String` data fields `familyName` and `givenName` with the obvious meanings. Provide preconditions and postconditions if needed.

```
public int compareTo(Person per) {
    if (familyName.compareTo(per.familyName) == 0)
        return givenName.compareTo(per.givenName);
    else
        return familyName.compareTo(per.familyName);
}
```

3. Write a Javadoc comment for the following method of class `Person`. Provide preconditions and postconditions if needed.

```
public void changeFamilyName(boolean justMarried, String newFamily) {
    if (justMarried)
        familyName = newFamily;
}
```

4. Write method `verifyPIN` for class `ATMbankAmerica` assuming this class has a data field `pin` (type `String`).

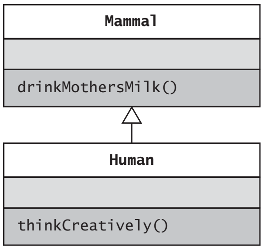
1.2 Introduction to Object-Oriented Programming (OOP)

In this course, you will learn to use features of Java that facilitate the practice of OOP. A major reason for the popularity of OOP is that it enables programmers to reuse previously written code saved as classes, reducing the time required to code new applications. Because previously written code has already been tested and debugged, the new applications should also be more reliable and therefore easier to test and debug.

However, OOP provides additional capabilities beyond the reuse of existing classes. If an application needs a new class that is similar to an existing class but not exactly the same, the programmer can create it by extending, or inheriting from, the existing class. The new class (called the subclass) can have additional data fields and methods for increased functionality. Its objects also inherit the data fields and methods of the original class (called the superclass).

Inheritance in OOP is analogous to inheritance in humans. We all inherit genetic traits from our parents. If we are fortunate, we may even have some earlier ancestors who have left us

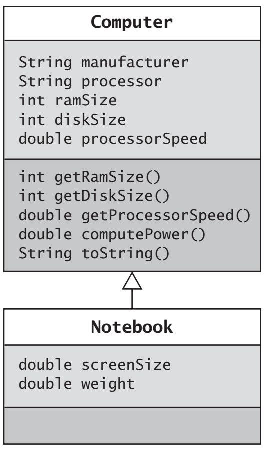
FIGURE 1.3
Classes `Mammal` and `Human`



an inheritance of monetary value. As we grow up, we benefit from our ancestors' resources, knowledge, and experiences, but our experiences will not affect how our parents or ancestors developed. Although we have two parents to inherit from, Java classes can have only one parent.

Inheritance and hierarchical organization allow you to capture the idea that one thing may be a refinement or an extension of another. For example, an object that is a `Human` is a `Mammal` (the superclass of `Human`). This means that an object of type `Human` has all the data fields and methods defined by class `Mammal` (e.g., method `drinkMothersMilk`), but it may also have more data fields and methods that are not contained in class `Mammal` (e.g., method `thinkCreatively`). Figure 1.3 shows this simple hierarchy. The solid line in the UML class diagram shows that `Human` is a subclass of `Mammal`, and, therefore, `Human` objects can use methods `drinkMothersMilk` and `thinkCreatively`. Objects farther down the hierarchy are more complex and less general than those farther up. For this reason an object that is a `Human` is a `Mammal`, but the converse is not true because every `Mammal` object does not necessarily have the additional properties of a `Human`. Although this seems counterintuitive, the subclass `Human` is actually more powerful than the superclass `Mammal` because it may have additional attributes that are not present in the superclass.

FIGURE 1.4
Classes `NoteBook` and `Computer`



A Superclass and Subclass Example

To illustrate the concepts of inheritance and class hierarchies, let's consider a simple case of two classes: `Computer` and `Notebook`. A `Computer` object has a `manufacturer`, `processor`, `RAM`, and `disk`. A notebook computer is a kind of computer, so it has all the properties of a computer plus some additional features (screen size and weight). There may be other subclasses, such as tablet computer or game computer, but we will ignore them for now. We can define class `Notebook` as a subclass of class `Computer`. Figure 1.4 shows the class hierarchy.

Class `Computer`

Listing 1.2 shows class `Computer.Java`. It is defined like any other class. It contains a constructor, several accessors, a `toString` method, and a method `computePower`, which returns the product of its RAM size and processor speed as a simple measure of its power.

LISTING 1.2
Class Computer.java

```
/** Class that represents a computer. */
public class Computer {
    // Data Fields
    private String manufacturer;
    private String processor;
    private double ramSize;
    private int diskSize;
    private double processorSpeed;

    // Methods
    /** Initializes a Computer object with all properties specified.
        @param man The computer manufacturer
        @param processor The processor type
        @param ram The RAM size
        @param disk The disk size
        @param procSpeed The processor speed
     */
    public Computer(String man, String processor, double ram,
                    int disk, double procSpeed) {
```

```
        manufacturer = man;
        this.processor = processor;
        ramSize = ram;
        diskSize = disk;
        processorSpeed = procSpeed;
    }

    public double computePower() { return ramSize * processorSpeed; }
    public double getRamSize() { return ramSize; }
    public double getProcessorSpeed() { return processorSpeed; }
    public int getDiskSize() { return diskSize; }
    // Insert other accessor and modifier methods here.

    public String toString() {
        String result = "Manufacturer: " + manufacturer +
                        "\nCPU: " + processor +
                        "\nRAM: " + ramSize + " gigabytes" +
                        "\nDisk: " + diskSize + " gigabytes" +
                        "\nProcessor speed: " + processorSpeed + " gigahertz";
        return result;
    }
}
```

Use of `this.`

In the constructor for the `Computer` class, the statement

```
this.processor = processor;
```

sets data field `processor` in the object under construction to reference the same string as parameter processor. The prefix `this.` makes data field `processor` visible in the constructor. This is necessary because the declaration of `processor` as a parameter hides the data field declaration.

PITFALL

Not Using `this.` to Access a Hidden Data Field

If you write the preceding statement as

```
processor = processor; // Copy parameter processor to itself.
```

you will not get an error, but the data field `processor` in the `Computer` object under construction will not be initialized and will retain its default value (`null`). If you later attempt to use data field `processor`, you may get an error or just an unexpected result. Some IDEs will provide a warning if `this.` is omitted.

Class `Notebook`

In the `Notebook` class diagram in Figure 1.4, we show just the data fields declared in class `Notebook`; however, `Notebook` objects also have the data fields that are inherited from class `Computer` (`processor`, `ramSize`, and so forth). The first line in class `Notebook` (Listing 1.3),

```
public class Notebook extends Computer {
```