DATA STRUCTURES

ABSTRACTION AND DESIGN USING JAVA

ELLIOT B. KOFFMAN | PAUL A. T. WOLFGANG

FOURTH EDITION

WILEY

DATA STRUCTURES

Abstraction and Design Using Java

FOURTH EDITION

ELLIOT B. KOFFMAN
Temple University

PAUL A. T. WOLFGANG
Temple University

WILEY

SENIOR VP	Smita Bakshi
EDITOR	Joanna Dingle
SENIOR MANAGING EDITOR	Judy Howarth
DIRECTOR OF CONTENT OPERATIONS	Martin Tribe
SENIOR MANAGER OF CONTENT OPERATIONS	Mary Corder
SENIOR PRODUCTION EDITOR	Loganathan Kandan
COVER PHOTO CREDIT	© Savgraf / Shutterstock

This book was set in 10/12 pt SabonLTStd-Roman by SPi Global.

ISBN: 978-1-119-70361-7 (PBK)
ISBN: 978-1-119-71249-7 (EVALC)

Library of Congress Cataloging-in-Publication Data

Names: Koffman, Elliot B., author. | Wolfgang, Paul A. T., author.
Title: Data structures : abstraction and design using Java / Elliot B. Koffman, Temple University, Paul A. T. Wolfgang, Temple University.
Other titles: Objects, abstraction, data structures and design using Java
Description: Fourth edition. | Hoboken, NJ : John Wiley & Sons, Inc., 2021. | Original edition published under title: Objects, abstraction, data structures and design using Java.
Identifiers: LCCN 2020033276 (print) | LCCN 2020033277 (ebook) | ISBN 9781119703617 (paperback) | ISBN 9781119712473 (adobe pdf) | ISBN 9781119703594 (epub)
Subjects: LCSH: Data structures (Computer science) | Java (Computer program language) | Object-oriented programming (Computer science) | Application program interfaces (Computer software)
Classification: LCC QA76.9.D35 K58 2021 (print) | LCC QA76.9.D35 (ebook) | DDC 005.7/3—dc23
LC record available at https://lccn.loc.gov/2020033276
LC ebook record available at https://lccn.loc.gov/2020033277

The inside back cover will contain printing identification and country of origin if omitted from this page. In addition, if the ISBN on the back cover differs from the ISBN on this page, the one on the back cover is correct.

Preface

Our goal in writing this book was to combine a strong emphasis on problem-solving, program analysis, design, and testing 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, managing a list of cell-phone contacts, finding the shortest path through a maze, finding the shortest route to a destination, 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 pseudocode and UML class diagrams) before the implementation. We also provide a performance analysis when appropriate and provide examples of how to test the code developed in the text. 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 and/or third programming course in a computing curriculum involving the study of data structures emphasizing Object-Oriented Design (OOD). The book 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, and graphs. Although we expect that most readers will have completed a first programming course in Java, there is an extensive chapter (Appendix A) for those who may have taken a first programming course in a different language, or for those who want a comprehensive review of 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 (Applications Programmers Interface). Readers are encouraged throughout the text to use the Java Collections Framework as a resource for their programming. We want our readers to understand how to use the classes in the Java API in their own programming and to understand how these classes are implemented. Each class implementation in the text 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. Most of the tools for OOD are provided in the first three chapters and carried out throughout the book.

For example, Chapter 1 focuses on OOD and Class Hierarchies. It introduces UML, 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 begins with a discussion of algorithm analysis and illustrates how to use big-O notation to compare the performance of different algorithms. It also introduces the Java Collections Framework and focuses on the `List` interface. It discusses the `ArrayList` and `LinkedList` implementations of this interface and shows how they can be used to implement single- and double-linked lists and ordered lists.

In Chapter 3, we cover different testing strategies and debugging strategies. We show how to trace program execution by inserting extra output statements and also by using a debugger program in two popular IDEs (Integrated Development Environments). We also discuss the use of the JUnit platform to write test classes. Finally, we demonstrate how to use test-driven design as a technique for program development.

Features of the Fourth Edition

There were two major goals for the fourth edition. The first was to create an electronic book (ebook) that would enable a student to test their understanding of each new section by adding Show/Hide buttons to all odd-numbered self-check exercises at the end of the section. If a student selects the Show button after answering the question, the solution will pop-up. In addition to odd-numbered self-check exercises, this feature would be added to select programming exercises at the end of each section, and also to all the quick-check exercises and review questions at the end of each chapter. Answers to all exercises and questions would be available on the Wiley website. We were also able to add color to the diagrams in the ebook and traditional textbook, which makes it easier to follow them.

The second major goal was to update the coverage of Java up to Java 11 by emphasizing new features that facilitate Java programming. For example, in Appendix A and Chapter 7, we show how to use `jshell` to execute individual Java statements. Also in Appendix A, we introduce the `var` keyword, which together with the diamond operator (`<>`) reduce redundancy in variable declarations. We use these features where appropriate throughout the text. We also introduced the `try` with resources statements in Appendix A and use this statement to simplify programs that use files for input/output. We also provide more emphasis on the use of the enhanced for statement and iterators beginning in Chapter 2. In Chapter 3, we show how to use JUnit5 to write test classes and increased our usage of JUnit5 to test new classes. Also in the JUnit5 section, we briefly introduced lambda expressions and we rewrote Section 6.4 to emphasize the use of standard functional interfaces with lambda expressions. This change facilitates the use of functional programming and helps to simplify code.

Another goal of both the third and fourth editions is 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. We also provided Appendix A to make the transition easier.

Another change in the third edition was to move the discussion of algorithm analysis to the beginning of the Chapter 2 so that big-O notation could be used to compare the efficiency of different `List` implementations introduced later in the chapter. In the fourth edition, we have added a brief discussion showing the improvement in performance of a binary search algorithm over a linear search.

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 Wiley website.

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 new Java feature 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.

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 (Chapter 6); Sets and Maps (Chapter 7); Self-Balancing Trees (Chapter 9); 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. We illustrate 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 testing classes using JUnit5 and also has a section on Test-Driven Development. It also discusses using JUnit5 and a debugger in IntelliJ and

Eclipse, two popular Java IDEs (Integrated Development Environments), 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 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. The skip-list data structure is also discussed in this chapter.

Chapter 8 covers various sorting algorithms including merge sort, 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, and B-trees.

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. There is a new section on the A* algorithm, which is an improvement of Dijkstra's algorithm that uses a heuristic to prune the search tree. In most programs, the last few chapters would be covered in a second course in algorithms and data structures.

Supplements and Companion Websites

The following supplementary materials are available on the Instructor's Companion Website for this textbook at www.wiley.com/go/koffman/datastructuresjava4e. Items marked for students are accessible on the Student Companion Website at the same address.

- 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 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 Paul LaFollette, Rolf Lakaemper and James Korsh,

colleagues at Temple University, who used earlier editions in their classes, and Elliott Moss, University of Massachusetts, who prepared powerpoint slides.

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 Course (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 of the Fourth Edition Revision Plan

David W. Binkley
Loyola University, Maryland

Eduardo Bonelli
Stevens Institute of Technology

Robert Boothe
University of Southern Maine

Wei Chang
Saint Joseph's University

Jose L. Cordova
University of Louisiana at Monroe

Terrence Fries
Indiana University of Pennsylvania

Carleton Moore
University of Hawaii at Manoa

Dave Musicant
Carleton College

Sneha Narayan
Carleton College

Chris Taylor
Milwaukee School of Engineering

David Wolff
Pacific Lutheran University

Reviewers of Previous Editions

Sheikh Iqbal Ahamed, *Marquette University*
Razvan Andonie, *Central Washington University*
Justin Beck, *Oklahoma State University*
Antonia Boadi, *California State University Dominguez Hills*
John Bowles, *University of South Carolina*

Mikhail Brikman, *Salem State College*
Robert Burton, *Brigham Young University*
Mary Elaine Califf, *Illinois State University*
Chakib Chraibi, *Barry University*
Teresa Cole, *Boise State University*
Jose Cordova, *University of Louisiana Monroe*
Tom Cortina, *SUNY Stony Brook*
Joyce Crowell, *Belmont University*
Adrienne Decker, *SUNY Buffalo*
Chris Dovolis, *University of Minnesota*
Vladimir Drobot, *San Jose State University*
Kenny Fong, *Southern Illinois University, Carbondale*
Robert Franks, *Central College*
Barabra Gannod, *Arizona State University East*
Wayne Goddard, *Clemson University*
Simon Gray, *College of Wooster*
Ralph Grayson, *Oklahoma State University*
Allan M. Hart, *Minnesota State University, Mankato*
Wei Hu, *Houghton College*
James K. Huggins, *Kettering University*
Chris Ingram, *University of Waterloo*
Gregory Kesden, *Carnegie Mellon University*
Edward Kovach, *Franciscan University of Steubenville*
Saeed Monemi, *California Polytechnic and State University*
Robert Noonan, *College of William and Mary*
Sarah Matzko, *Clemson University*
Lester McCann, *University of Arizona*
Ron Metoyer, *Oregon State University*
Kathleen O'Brien, *Foothill College*
Rich Pattis, *Carnegie Mellon University*
Thaddeus F. Pawlicki, *University of Rochester*
Sally Peterson, *University of Wisconsin—Madison*
Rathika Rajaravivarma, *Central Connecticut State University*
Sam Rhoads, *Honolulu Community College*
Salam N. Salloum, *California State Polytechnic University, Pomona*
Mike Scott, *University of Texas—Austin*
Vijayakumar Shanmugasundaram, *Concordia College Moorhead*
Gene Sheppard, *Perimeter College*
Linda Sherrell, *University of Memphis*
Victor Shtern, *Boston University*
Meena Srinivasan, *Mary Washington College*
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*
David Weaver, *Shepherd University*
Jim Weir, *Marist College*
Stephen Weiss, *University of North Carolina—Chapel Hill*
Glenn Wiggins, *Mississippi College*
Bruce William, *California State University Pomona*
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.

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 are our editors, Jennifer Yee and Jennifer Manias, Judy Howarth, Senior Managing Editor, and Loganathan Kandan and Padmapriya Soundararajan, Senior Production Editors.

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 copublished their first Computer Science textbook as coauthors in 1977 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.

Elliot Koffman would like to thank

Caryn Koffman

and Paul Wolfgang would like to thank

Sharon Wolfgang

Contents

Appendix A Introduction to Java A-1

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` class 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 Abstract Data Types (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

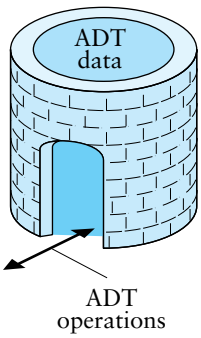


Figure 1.1 shows a diagram of an ADT. The data values stored in the ADT are hidden inside the circular wall. The bricks around this wall are used to indicate that these data values cannot be accessed except by going through the ADT's methods.

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 way 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.

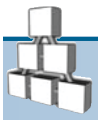
A class that implements an ATM must provide a method for each operation. We can write this requirement as the interface `ATM` and save it in file `ATM.java`, shown in Listing 1.1. The keyword `interface` on the header line indicates that an interface is being declared. If you are unfamiliar with the 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 and 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

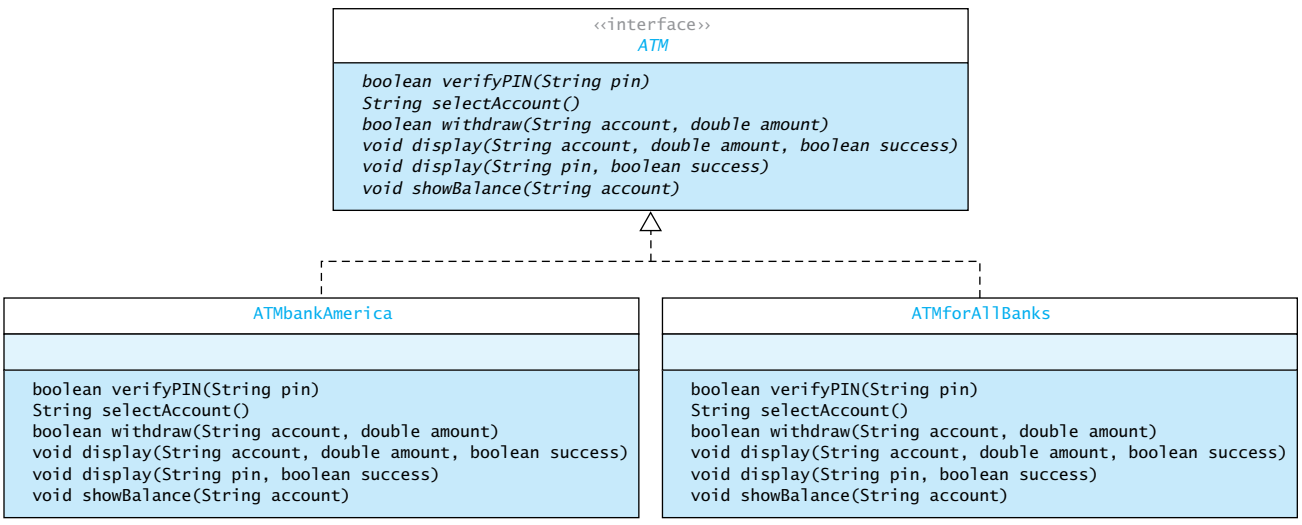
The class headings for two classes that implement interface ATM are

```
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) class 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 more details about UML class 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. Explain how an interface is like a contract.
3. Correct each of the following statements that is incorrect, assuming that class `PDGUI` and class `PDConsoleUI` implement interface `PDUserInterface` and neither is a subclass of the other.

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

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`).

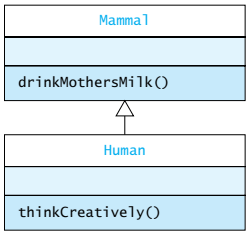
1.2 Introduction to OOP

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`



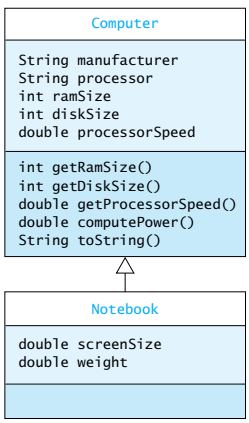
an inheritance of monetary value. As we grow up, we benefit from our ancestors' resources, knowledge, and experiences, but our experiences will not affect how our parents or ancestors developed. Although we have two parents to inherit from, Java classes can have only one parent.

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.

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`, Random Access Memory (`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.

FIGURE 1.4
Classes `Notebook` and `Computer`



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 int 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, int ram,
                    int disk, double procSpeed) {
        manufacturer = man;
        this.processor = processor;
        ramSize = ram;
        diskSize = disk;
        processorSpeed = procSpeed;
    }

    public double computePower() { return ramSize * processorSpeed; }
    public int 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 Integrated Development Environments (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 {
```

indicates that class Notebook extends class Computer and inherits its data and methods. Next, we define any additional data fields

```
// Data Fields
private double screenSize;
private double weight;
```

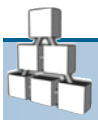
Initializing Data Fields in a Subclass

The constructor for class Notebook must begin by initializing the four data fields inherited from class Computer. Because those data fields are private to the superclass, Java requires that they be initialized by a superclass constructor. Therefore, a superclass constructor must be invoked as the first statement in the constructor body using a statement such as

```
super(man, proc, ram, disk, procSpeed);
```

This statement invokes the superclass constructor with the signature Computer(String, String, double, int, double), passing the four arguments listed to the constructor. (A method signature consists of the method's name followed by its parameter types.) The following constructor for Notebook also initializes the data fields that are not inherited. Listing 1.3 shows class Notebook.

```
public Notebook(String man, String proc, double ram, int disk,
                double procSpeed, double screen, double wei) {
    super(man, proc, ram, disk, procSpeed);
    screenSize = screen;
    weight = wei;

}
```



SYNTAX **super(. . .);**

FORM:

```
super();
super(argumentList);
```

EXAMPLE:

```
super(man, proc, ram, disk, procSpeed);
```

MEANING:

The `super()` call in a class constructor invokes the superclass's constructor that has the corresponding *argumentList*. The superclass constructor initializes the inherited data fields as specified by its *argumentList*. The `super()` call must be the first statement in a constructor.

LISTING 1.3

Class Notebook

```
/** Class that represents a notebook computer. */
public class Notebook extends Computer {
    // Data Fields
    private double screenSize;
    private double weight;

    // Methods
    /** Initializes a Notebook object with all properties specified.
```

```
        @param man The computer manufacturer
        @param proc The processor type
        @param ram The RAM size
        @param disk The disk size
        @param procSpeed The processor speed
        @param screen The screen size
        @param wei The weight
     */
    public Notebook(String man, String proc, int ram, int disk,
                    double procSpeed, double screen, double wei) {
        super(man, proc, ram, disk, procSpeed);
        screenSize = screen;
        weight = wei;
    }
}
```

The No-Parameter Constructor

If the execution of any constructor in a subclass does not invoke a superclass constructor, Java automatically invokes the no-parameter constructor for the superclass. Java does this to initialize that part of the object inherited from the superclass before the subclass starts to initialize its part of the object. Otherwise, the part of the object that is inherited would remain uninitialized.

PITFALL

Not Defining the No-Parameter Constructor

If no constructors are defined for a class, the no-parameter constructor for that class will be provided by default. However, if any constructors are defined, the no-parameter constructor must also be defined explicitly if it needs to be invoked. Java does not provide it automatically because it may not make sense to create a new object of that type without providing initial data field values. (It was not defined in class `Notebook` or `Computer` because we want the client to specify some information about a `Computer` object when that object is created.) If the no-parameter constructor is defined in a subclass but is not defined in the superclass, you will get a syntax error `constructor not defined`. You can also get this error if a subclass constructor does not explicitly call a superclass constructor. There will be an implicit call to the no-parameter superclass constructor, so it must be defined.

Protected Visibility for Superclass Data Fields

The data fields inherited from class `Computer` have private visibility. Therefore, they can be accessed only within class `Computer`. Because it is fairly common for a subclass method to reference data fields declared in its superclass, Java provides a less restrictive form of visibility called *protected visibility*. A data field (or method) with protected visibility can be accessed in the class defining it, in any subclass of that class, or in any class in the same package. Therefore, if we had used the declaration

```
protected String manufacturer;
```

in class `Computer`, the following assignment statement would be valid in class `Notebook`:

```
manufacturer = man;
```

We will use protected visibility on occasion when we are writing a class that we intend to extend. However, in general, it is better to use private visibility because subclasses may be written by different programmers, and it is always a good practice to restrict and control access to the superclass data fields. We discuss visibility further in Section 1.7.

Is-a versus *Has-a* Relationships

One misuse of inheritance is confusing: the *has-a* relationship with the *is-a* relationship. The *is-a* relationship between classes means that one class is a subclass of the other class. For example, a game computer is a computer with specific attributes that make it suitable for gaming applications (enhanced graphics, fast processor) and is a subclass of the `Computer` class. The *is-a* relationship is achieved by extending a class.

The *has-a* relationship between classes means that one class has the second class as an attribute. For example, a game box is not really a computer (it is a kind of entertainment device), but it has a computer as a component. The *has-a* relationship is achieved by declaring a `Computer` data field in the game box class.

Another issue that sometimes arises is determining whether to define a new class in a hierarchy or whether a new object is a member of an existing class. For example, netbook computers are smaller portable computers that can be used for general purpose computing but are also used extensively for Web browsing. Should we define a separate class `NetBook`, or is a netbook computer a `Notebook` object with a smaller screen and lower weight?

EXERCISES FOR SECTION 1.2

SELF-CHECK

1. Explain the effect of each valid statement in the following fragment. Indicate any invalid statements.

```
Computer c1 = new Computer();
Computer c2 = new Computer("Ace", "AMD", 16, 1536, 4.1);
Notebook c3 = new Notebook("Ace", "AMD", 32, 3584, 3.8);
Notebook c4 = new Notebook("Bravo", "Intel", 8, 256, 3.9, 14.0, 2.4);
System.out.println(c2.manufacturer + ", " + c4.processor);
System.out.println(c2.getDiskSize() + ", " + c4.getRamSize());
System.out.println(c2.toString() + "\n" + c4.toString());
```

2. Indicate where in the hierarchy you might want to add data fields for the following and the kind of data field you would add.

   Cost
   The battery identification
   Time before battery discharges
   Number of expansion slots
   Wireless Internet available

3. Can you add the following constructor to class `Notebook`? If so, what would you need to do to class `Computer`?

```
public Notebook() {}
```

PROGRAMMING

1. Write accessor and modifier methods for class `Computer`.
2. Write accessor and modifier methods for class `Notebook`.

1.3 Method Overriding, Method Overloading, and Polymorphism

In the preceding section, we discussed inherited data fields. We found that we could not access an inherited data field in a subclass object if its visibility was private. Next, we consider inherited methods. Methods generally have public visibility, so we should be able to access a method that is inherited. However, what if there are multiple methods with the same name in a class hierarchy? How does Java determine which one to invoke? We answer this question next.

Method Overriding

Let's use the following main method to test our class hierarchy.

```
/** Tests classes Computer and Notebook. Creates an object of each and
    displays them.
    @param args[] No control parameters
 */
public static void main(String[] args) {
    Computer myComputer =
        new Computer("Acme", "Intel", 32, 2560, 4.7);
    Notebook yourComputer =
        new Notebook("DellGate", "AMD", 8, 256,
                     3.4, 13.3, 2.94);
    System.out.println("My computer is:\n" + myComputer.toString());
    System.out.println("\nYour computer is:\n" +
                       yourComputer.toString());
}
```

In the second call to `println`, the method call

```
yourComputer.toString()
```

applies method `toString` to object `yourComputer` (type `Notebook`). Because class `Notebook` doesn't define its own `toString` method, class `Notebook` inherits the `toString` method defined in class `Computer`. Executing this method displays the following output lines:

```
My computer is:
Manufacturer: Acme
CPU: Intel
RAM: 32 gigabytes
Disk: 2560 gigabytes
Speed: 4.7 gigahertz

Your computer is:
Manufacturer: DellGate
CPU: AMD
RAM: 8 gigabytes
Disk: 256 gigabytes
Speed: 3.4 gigahertz
```

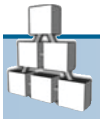
Unfortunately, this output doesn't show the complete state of object `yourComputer`. To show the complete state of a notebook computer, we need to define a `toString` method for class `Notebook`. If class `Notebook` has its own `toString` method, it will override the inherited method and will be invoked by the method call `yourComputer.toString()`. We define method `toString` for class `Notebook` next.

```
public String toString() {
    String result = super.toString() +
                    "\nScreen size: " + screenSize + " inches" +
                    "\nWeight: " + weight + " pounds";
    return result;
}
```

This method `Notebook.toString` returns a string representation of the state of a `Notebook` object. The first line

```
String result = super.toString()
```

uses method call **`super`**`.toString()` to invoke the `toString` method of the superclass (method `Computer.toString`) to get the string representation of the four data fields that are inherited from the superclass. The next two lines append the data fields defined in class `Notebook` to this string.



SYNTAX super.

FORM:

`super.`*`methodName`*`()`
`super.`*`methodName`*`(`*`argumentList`*`)`

EXAMPLE:

```
super.toString()
```

MEANING:

Using the prefix **`super.`** in a call to method *`methodName`* calls the method with that name defined in the superclass of the current class.

PROGRAM STYLE

Calling Method `toString()` Is Optional

In the `println` statement shown earlier,

```
System.out.println("My computer is:\n" + myComputer.toString());
```

the explicit call to method `toString` is not required. The statement could be written as

```
System.out.println("My computer is:\n" + myComputer);
```

Java automatically applies the `toString` method to an object referenced in a `String` expression. Normally, we will not explicitly call `toString`.

PITFALL

Overridden Methods Must Have Compatible Return Types

If you write a method in a subclass that has the same signature as one in the superclass but a different return type, you may get the following error message: `in` *`subclass-name`* `cannot override` *`method-name`* `in` *`superclass-name`*`; attempting to use incompatible return type`. The subclass method return type must be the same as or a subclass of the superclass method's return type.

Method Overloading

Let's assume we have decided to standardize and purchase our notebook computers from only one manufacturer. We could then introduce a new constructor with one less parameter for class `Notebook`.

```
public Notebook(String proc, int ram, int disk, double procSpeed,
                double screen, double wei) {
    this(DEFAULT_NB_MAN, proc, ram, disk, procSpeed, screen, wei);
}
```

The method call

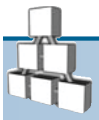
```
this(DEFAULT_NB_MAN, proc, ram, disk, procSpeed, screen, wei);
```

invokes the six-parameter constructor (see Listing 1.3), passing on the five arguments it receives and the constant string `DEFAULT_NB_MAN` (defined in class `Notebook`). The six-parameter constructor begins by calling the superclass constructor, satisfying the requirement that it be called first. We now have two constructors with different signatures in class `Notebook`. Having multiple methods with the same name but different signatures in a class is called *method overloading*.

Now we have two ways to create new `Notebook` objects. Both of the following statements are valid:

```
Notebook lTP1 = new Notebook("Intel", 16, 512, 4.6, 13.3, 3.18);
Notebook lTP2 = new Notebook("MicroSys", "AMD", 8, 256, 3.9, 17, 5.4);
```

The manufacturer of `lTP1` is `DEFAULT_NB_MAN`.



SYNTAX this (. . .);

FORM:
`this(argumentList);`

EXAMPLE:
`this(DEFAULT_NB_MAN, proc, ram, disk, procSpeed);`

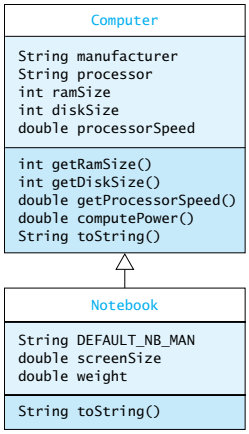
MEANING:
The call to **`this()`** invokes the constructor for the current class whose parameter list matches the argument list. The constructor initializes the new object as specified by its arguments. The invocation of another constructor (through either **`this()`** or **`super()`**) must be the first statement in a constructor.

Listing 1.4 shows the complete class `Notebook`. Figure 1.5 shows the UML diagram, revised to show that `Notebook` has a `toString` method and a constant data field. The next Pitfall discusses the reason for the `@Override` annotation preceding method `toString`.

LISTING 1.4
Complete Class `Notebook` with Method `toString`

```
/** Class that represents a notebook computer. */
public class Notebook extends Computer {
    // Data Field
    private static final String DEFAULT_NB_MAN = "MyBrand";
    private double screenSize;
    private double weight;
```

FIGURE 1.5
Revised UML Diagram for `Computer` Class Hierarchy




```
    /** Initializes a Notebook object with all properties specified.
        @param man The computer manufacturer
        @param proc The processor type
        @param ram The RAM size
        @param disk The disk size
        @param screen The screen size
        @param wei The weight
     */
    public Notebook(String man, String proc, int ram, int disk,
                    double procSpeed, double screen, double wei) {
        super(man, proc, ram, disk, procSpeed);
        screenSize = screen;
        weight = wei;
    }

    /** Initializes a Notebook object with 6 properties specified. */
    public Notebook(String proc, int ram, int disk,
                    double procSpeed, double screen, double wei) {
        this(DEFAULT_NB_MAN, proc, ram, disk, procSpeed, screen, wei);
    }

    @Override
    public String toString() {
        String result = super.toString() +
                        "\nScreen size: " + screenSize + " inches" +
                        "\nWeight: " + weight + " pounds";
        return result;
    }
}
```

PITFALL

Overloading a Method When Intending to Override It

To override a method, you must use the same name and the same number and types of the parameters as the superclass method that is being overridden. If the name is the same but the number or types of the parameters are different, then the method is overloaded instead. Normally, the compiler will not detect this as an error. However, it is a sufficiently common error that a feature was added to the Java compiler so that programmers can indicate that they intend to override a method. If you precede the declaration of the method with the annotation `@Override`, the compiler will issue an error message if the method is overloaded instead of overridden.

PROGRAM STYLE

Precede an Overridden Method with the Annotation `@Override`

Whenever a method is overridden, we recommend preceding it with the annotation `@Override`. Some Java IDEs will either issue a warning or add this annotation automatically.

Polymorphism

An important advantage of OOP is that it supports a feature called *polymorphism*, which means many forms or many shapes. Polymorphism enables the JVM to determine at run time which of the classes in a hierarchy is referenced by a superclass variable or parameter. Next, we will see how this simplifies the programming process.

Suppose you are not sure whether a computer referenced in a program will be a notebook or a regular computer. If you declare the reference variable

```
Computer theComputer;
```

you can use it to reference an object of either type because a type `Notebook` object can be referenced by a type `Computer` variable. In Java, a variable of a superclass type (general) can reference an object of a subclass type (specific). `Notebook` objects are `Computer` objects with more features. When the following statements are executed,

```
theComputer = new Computer("Acme", "Intel", 2, 160, 2.6);
System.out.println(theComputer.toString());
```

you would see four output lines, representing the state of the object referenced by `theComputer`.

Now suppose you have purchased a notebook computer instead. What happens when the following statements are executed?

```
theComputer = new Notebook("Bravo", "Intel", 4, 240, 2.4. 15.0, 7.5);
System.out.println(theComputer.toString());
```

Recall that `theComputer` is type `Computer`. Will the `theComputer.toString()` method call return a string with all seven data fields or just the five data fields defined for a `Computer` object? The answer is a string with all seven data fields. The reason is that the type of the object receiving the `toString` message determines which `toString` method is called. Even though variable `theComputer` is type `Computer`, it references a type `Notebook` object, and the `Notebook` object receives the `toString` message. Therefore, the method `toString` for class `Notebook` is the one called.

This is an example of polymorphism. Variable `theComputer` references a `Computer` object at one time and a `Notebook` object another time. At compile time, the Java compiler can't determine what type of object `theComputer` will reference, but at run time, the JVM knows the type of the object that receives the `toString` message and can call the appropriate `toString` method.

EXAMPLE 1.2 If we declare the array `labComputers` as follows:

```
Computer[] labComputers = new Computer[10];
```

each subscripted variable `labComputers[i]` can reference either a `Computer` object or a `Notebook` object because `Notebook` is a subclass of `Computer`. For the method call `labComputers[i].toString()`, polymorphism ensures that the correct `toString` method is called. For each value of subscript `i`, the actual type of the object referenced by `labComputers[i]` determines which `toString` method will execute (`Computer.toString or Notebook.toString`).

Methods with Class Parameters

Polymorphism also simplifies programming when we write methods that have class parameters. For example, if we want to compare the power of two computers without polymorphism, we will need to write overloaded `comparePower` methods in class `Computer`, one for each subclass parameter and one with a class `Computer` parameter. However, polymorphism enables us to write just one method with a `Computer` parameter.

EXAMPLE 1.3 Method `Computer.comparePowers` compares the power of the `Computer` object it is applied to with the `Computer` object passed as its argument. It returns –1, 0, or +1 depending on which computer has more power. It does not matter whether `this` or `aComputer` references a `Computer` or a `Notebook` object.

```
/** Compares power of this computer and its argument computer
    @param aComputer The computer being compared to this computer
    @return -1 if this computer has less power,
            0 if the same, and
            +1 if this computer has more power.
 */
public int comparePower(Computer aComputer) {
    if (this.computePower() < aComputer.computePower())
        return -1;
    else if (this.computePower() == aComputer.computePower())
        return 0;
    else return 1;
}
```

EXERCISES FOR SECTION 1.3

SELF-CHECK

1. Explain the effect of each of the following statements. Which one(s) would you find in class `Computer`? Which one(s) would you find in class `Notebook`?

```
super(man, proc, ram, disk, procSpeed);
this(man, proc, ram, disk, procSpeed);
```

2. Indicate whether methods with each of the following signatures and return types (if any) would be allowed and in what classes they would be allowed. Explain your answers.

```
Computer()
Notebook()
int toString()
double getRamSize()
String getRamSize()
String getRamSize(String)
String getProcessor()
double getScreenSize()
```

3. For the loop body in the following fragment, indicate which method is invoked for each value of `i`. What is printed?

```
Computer comp[] = new Computer[3];
comp[0] = new Computer("Ace", "AMD", 16, 1024, 3.5);
comp[1] = new Notebook("Dell", "Intel", 8, 512, 2.2, 15.5, 4.5);
comp[2] = comp[1];
for (int i = 0; i < comp.length; i++) {
    System.out.println(comp[i].getRamSize() + "\n" +
                       comp[i].toString());
}
```

4. When does Java determine which `toString` method to execute for each value of `i` in the `for` statement in the preceding question: at compile time or at run time? Explain your answer.

PROGRAMMING

1. Write constructors for both classes that allow you to specify only the processor, RAM size, and disk size.

1.4 Abstract Classes

In this section, we introduce another kind of class called an *abstract class*. An abstract class is denoted using the word *abstract* in its heading:

visibility `abstract class` *className*

An abstract class differs from an actual class (sometimes called a concrete class) in two respects:

- An abstract class cannot be instantiated.
- An abstract class may declare abstract methods.

Just as in an interface, an abstract method is declared through a method heading in the abstract class definition. This heading indicates the result type, method name, and parameters, thereby specifying the form that any actual method declaration must take:

visibility `abstract` *resultType methodName(parameterList);*

However, the complete method definition, including the method body (implementation), does not appear in the abstract class definition.

In order to compile without error, an actual class that is a subclass of an abstract class must provide an implementation for each abstract method of its abstract superclass. The heading for each actual method must match the heading for the corresponding abstract method.

We introduce an abstract class in a class hierarchy when we need a base class for two or more actual classes that share some attributes. We may want to declare some of the attributes and define some of the methods that are common to these base classes. If, in addition, we want to require that the actual subclasses implement certain methods, we can accomplish this by making the base class an abstract class and declaring these methods abstract.

EXAMPLE 1.4 The Food Guide Pyramid provides a recommendation of what to eat each day based on established dietary guidelines. There are six categories of foods in the pyramid: fats, oils, and sweets; meats, poultry, fish, and nuts; milk, yogurt, and cheese; vegetables; fruits; and bread, cereal, and pasta. If we wanted to model the Food Guide Pyramid, we might have each of these as actual subclasses of an abstract class called `Food`:

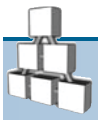
```
/** Abstract class that models a kind of food. */
public abstract class Food {
    // Data Field
    private double calories;

    // Abstract Methods
    /** Calculates the percent of protein in a Food object. */
    public abstract double percentProtein();
```

```
    /** Calculates the percent of fat in a Food object. */
    public abstract double percentFat();
    /** Calculates the percent of carbohydrates in a Food object. */
    public abstract double percentCarbohydrates();

    // Actual Methods
    public double getCalories() { return calories; }
    public void setCalories(double cal) {
        calories = cal;
    }
}
```

The three abstract method declarations

```
public abstract double percentProtein();
public abstract double percentFat();
public abstract double percentCarbohydrates();
```

impose the requirement that all actual subclasses implement these three methods. We would expect a different method definition for each kind of food. The keyword abstract must appear in all abstract method declarations in an abstract class. Recall that this is not required for abstract method declarations in interfaces.



SYNTAX Abstract Class Definition

FORM:

```
public abstract class className {
    data field declarations
    abstract method declarations
    actual method definitions
}
```

EXAMPLE:

```
public abstract class Food {
    // Data Field
    private double calories;

    // Abstract Methods
    public abstract double percentProtein();
    public abstract double percentFat();
    public abstract double percentCarbohydrates();

    // Actual Methods
    public double getCalories() { return calories; }
    public void setCalories(double cal) {
        calories = cal;
    }
}
```

INTERPRETATION:

Abstract class *className* is defined. Normally, the class body will have declarations for abstract methods, but data field declarations and concrete method definitions are also allowed but are optional. Each abstract method declaration consists of a method heading containing the keyword **abstract**.

PITFALL

Omitting the Definition of an Abstract Method in a Subclass

If you write class `Vegetable` and forget to define method `percentProtein`, you will get the syntax error class `Vegetable should be declared abstract`, it does not define `method percentProtein` in class `Food`. Although this error message is misleading (you did not intend `Vegetable` to be abstract), any class with undefined methods is abstract by definition. The compiler's rationale is that the undefined method is intentional, so `Vegetable` must be an abstract class, with a subclass that defines `percentProtein`.

Referencing Actual Objects

Because class `Food` is abstract, you can't create type `Food` objects. However, you can use a type `Food` variable to reference an actual object that belongs to a subclass of type `Food`. For example, an object of type `Vegetable` can be referenced by a `Vegetable` or `Food` variable because `Vegetable` is a subclass of `Food` (i.e., a `Vegetable` object is also a `Food` object).

EXAMPLE 1.5 The following statement creates a `Vegetable` object that is referenced by variable `mySnack` (type `Food`).

```
Food mySnack = new Vegetable("carrot sticks");
```

Initializing Data Fields in an Abstract Class

An abstract class can't be instantiated. However, an abstract class can have constructors that initialize its data fields when a new subclass object is created. The subclass constructor will use `super(...)` to call such a constructor.

Abstract Class Number and the Java Wrapper Classes

The abstract class `Number` is predefined in the Java class hierarchy. It has as its subclasses all the wrapper classes for primitive numeric types (e.g., `Byte`, `Double`, `Integer`, `Long`, and `Short`). A *wrapper class* is used to store a primitive-type value in an object type.

Each wrapper class contains a `static valueOf` *factory method* that is used to create an object of that class. The `valueOf` methods take a primitive value or a numeric `String` and return an object that stores the corresponding primitive-type value. For example, `Integer.valueOf(35)` or `Integer.valueOf("35")` returns a type `Integer` object that stores the `int` `35`. A wrapper class also has methods for converting the value stored in an object to a different numeric type.

Figure 1.6 shows a portion of the class hierarchy with base class `Number`. Italicizing the class name `Number` in its class box indicates that `Number` is an abstract class and, therefore, cannot be instantiated. Listing 1.5 shows part of the definition for class `Number`. Two abstract methods are declared (`intValue` and `doubleValue`), and one actual method (`byteValue`) is defined.

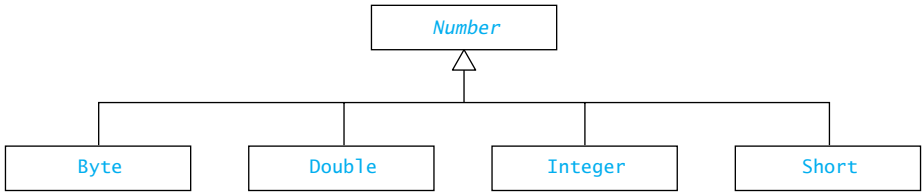


FIGURE 1.6 The Abstract Class `Number` and Selected Subclasses

In the actual implementation of `Number`, the body of `byteValue` would be provided, but we just indicate its presence in Listing 1.5.

LISTING 1.5

Part of Abstract Class `java.lang.Number`

```
public abstract class Number {
    // Abstract Methods
    /** Returns the value of the specified number as an int.
        @return The numeric value represented by this object after
                conversion to type int
     */
    public abstract int intValue();

    /** Returns the value of the specified number as a double.
        @return The numeric value represented by this object
                after conversion to type double
     */
    public abstract double doubleValue();

    ...

    // Actual Methods
    /** Returns the value of the specified number as a byte.
        @return The numeric value represented by this object
                after conversion to type byte
     */
    public byte byteValue() {
        // Implementation not shown.
        ...
    }
}
```

Summary of Features of Actual Classes, Abstract Classes, and Interfaces

It is easy to confuse abstract classes, interfaces, and actual classes (concrete classes). Table 1.1 summarizes some important points about these constructs.

TABLE 1.1

Comparison of Actual Classes, Abstract Classes, and Interfaces

Property	Actual Class	Abstract Class	Interface
Instances (objects) of this can be created	Yes	No	No
This can define instance variables	Yes	Yes	No
This can define methods	Yes	Yes	Yes
This can define constants	Yes	Yes	Yes
The number of these a class can extend	0 or 1	0 or 1	0
The number of these a class can implement	0	0	Any number
This can extend another class	Yes	Yes	No
This can declare abstract methods	No	Yes	Yes
Variables of this type can be declared	Yes	Yes	Yes

A class (abstract or actual) can extend only one other class; however, there is no restriction on the number of interfaces a class can implement. An interface cannot extend a class.

An abstract class may implement an interface just as an actual class does, but unlike an actual class, it doesn't have to define all of the methods declared in the interface. It can leave the implementation of some of the abstract methods to its subclasses.

Both abstract classes and interfaces declare abstract methods. However, unlike an interface, an abstract class can also have data fields and methods that are not abstract. You can think of an abstract class as combining the properties of an actual class, by providing inherited data fields and methods to its subclasses, and of an interface, by specifying requirements on its subclasses through its abstract method declarations.

Implementing Multiple Interfaces

A class can extend only one other class, but it may implement more than one interface. For example, assume interface `StudentInt` specifies methods required for student-like classes and interface `EmployeeInt` specifies methods required for employee-like classes. The following header for class `StudentWorker`

```
public class StudentWorker implements StudentInt, EmployeeInt
```

means that class `StudentWorker` must define (provide code for) all of the implement methods declared in both interfaces. Therefore, class `StudentWorker` supports operations required for both interfaces.

Extending an Interface

Interfaces can also extend other interfaces. In Chapter 2, we will introduce the Java Collection Framework. This class hierarchy contains several interfaces and classes that manage the collection of objects. At the top of this hierarchy is the interface `Iterable`, which declares the method `iterator`. At the next lower level is interface `Collection`, which extends `Iterable`. This means that all classes that implement `Collection` must also implement `Iterable` and therefore must define the method `iterator`.

An interface can extend more than one other interface. In this case, the resulting interface includes the union of the methods defined in the superinterfaces. For example, we can define the interface `ComparableCollection`, which extends both `Comparable` and `Collection`, as follows:

```
public interface ComparableCollection extends Comparable, Collection { }
```

Note that this interface does not define any methods itself but does require any implementing class to implement all of the methods required by `Comparable` and by `Collection`.

PROGRAM STYLE

Using Factory Methods versus Constructors

The built-in `static valueOf` method in each wrapper class returns an object of that class. We have seen that constructors do the same thing. However, factory methods have an advantage in that they can have meaningful names that describe what they do rather than just the same name as the class that defines them. Also, they can return objects of different types. We will see an example of this in Section 1.8.

EXERCISES FOR SECTION 1.4

SELF-CHECK

1. What are two important differences between an abstract class and an actual class? What are the similarities?
2. What do abstract methods and interfaces have in common? How do they differ?
3. Explain the effect of each statement in the following fragment and trace the loop execution for each value of `i`, indicating which `doubleValue` method executes, if any. What is the final value of `x`?

```
Number[] nums = new Number[5];
nums[0] = Integer.valueOf(35);
nums[1] = Double.valueOf(3.45);
nums[4] = Double.valueOf("2.45e6");
double x = 0;
for (int i = 0; i < nums.length; i++) {
    if (nums[i] != null)
        x += nums[i].doubleValue();
}
```

4. What is the purpose of the `if` statement in the loop in Question 3? What would happen if it were omitted?

PROGRAMMING

1. Write class `Vegetable`. Assume that a vegetable has three `double` constants: `VEG_FAT_CAL`, `VEG_PROTEIN_CAL`, and `VEG_CARBO_CAL`. Compute the fat percentage as `VEG_FAT_CAL` divided by the sum of all the constants.
2. Earlier we discussed a `Computer` class with a `Notebook` class as its only subclass. However, there are many different kinds of computers. An organization may have servers, mainframes, desktop PCs, and notebooks. There are also personal data assistants and game computers. So it may be more appropriate to declare class `Computer` as an abstract class that has an actual subclass for each category of computer. Write an abstract class `Computer` that defines all the methods shown earlier and declares an abstract method with the signature `costBenefit(double)` that returns the cost–benefit (type `double`) for each category of computer.

1.5 Class `Object` and Casting

The class `Object` is a special class in Java because it is the root of the class hierarchy, and every class has `Object` as a superclass. All classes inherit the methods defined in class `Object`; however, these methods may be overridden in the current class or in a superclass (if any). Table 1.2 shows a few of the methods of class `Object`. We discuss method `toString` next and the other `Object` methods shortly thereafter.

The Method `toString`

You should always override the `toString` method if you want to represent an object's state (information stored). If you don't override it, the `toString` method for class `Object` will execute and return a string, but not what you are expecting.

TABLE 1.2
The Class `Object`

Method	Behavior
`boolean equals(Object obj)`	Compares this object to its argument
`int hashCode()`	Returns an integer hash code value for this object
`String toString()`	Returns a string that textually represents the object
`Class<?> getClass()`	Returns a unique object that identifies the class of this object

EXAMPLE 1.6 If we didn't have a `toString` method in class `Computer` or `Notebook`, the method call `aComputer.toString()` would call the `toString` method inherited from class `Object`. This method would return a string such as `Computer@ef08879`, which shows the object's class name and a special integer value that is its "hash code"—not its state. Method `hashCode` is discussed in Chapter 7.

Operations Determined by Type of Reference Variable

You have seen that a variable can reference an object whose type is a subclass of the variable type. Because `Object` is a superclass of class `Integer`, the statement

```
Object aThing = Integer.valueOf(25);
```

will compile without error, creating the object reference shown in Figure 1.7. However, even though `aThing` references a type `Integer` object, we can't process this object like other `Integer` objects. For example, the method call `aThing.intValue()` would cause the syntax error method `intValue()` not found in class `java.lang.Object`. The reason for this is that the type of the reference, not the type of the object referenced, determines what operations can be performed, and class `Object` doesn't have an `intValue` method. During compilation, Java can't determine what kind of object will be referenced by a type `Object` variable, so the only operations permitted are those defined for class `Object`. The type `Integer` instance methods not defined in class `Object` (e.g., `intValue` and `doubleValue`) can't be invoked.

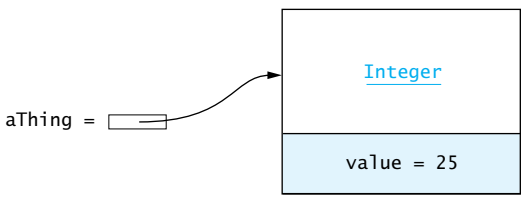
The method call `aThing.equals(Integer.valueOf("25"))` will compile because class `Object` has an `equals` method, and a subclass object has everything that is defined in its superclass. During execution, the `equals` method for class `Integer` is invoked, not class `Object`. (Why?)

Another surprising result is that the assignment statement

```
Integer aNum = aThing; // incompatible types
```

won't compile even though `aThing` references a type `Integer` object. The syntax error `incompatible types: found: Java.lang.Object, required: Java.lang.Integer` indicates that the expression type is incorrect (type `Object`, not type `Integer`). The reason Java won't compile this assignment is that Java is a strongly typed language, so the Java compiler always verifies that the type of the expression (`aThing` is type `Object`) being assigned is compatible with the variable type (`aNum` is type `Integer`). We show how to use casting to accomplish this in the next section.

FIGURE 1.7
Type `Integer` Object Referenced by `aThing` (Type `Object`)



Strong typing is also the reason that `aThing.intValue()` won't compile; the method invoked must be an instance method for class `Object` because `aThing` is type `Object`.

DESIGN CONCEPT

The Importance of Strong Typing

Suppose Java did not check the expression type and simply performed the assignment

```
Integer aNum = aThing; // incompatible types
```

Farther down the line, we might attempt to apply an `Integer` method to the object referenced by `aNum`. Because `aNum` is type `Integer`, the compiler would permit this. If `aNum` were referencing a type `Integer` object, then performing this operation would do no harm. But if `aNum` was referencing an object that was not type `Integer`, performing this operation would cause either a run-time error or an undetected logic error. It is much better to have the compiler tell us that the assignment is invalid.

Casting in a Class Hierarchy

Java provides a mechanism, *casting*, that enables us to process the object referenced by `aThing` through a reference variable of its actual type, instead of through a type `Object` reference. The expression

```
(Integer) aThing
```

casts the type of the object referenced by `aThing` (type `Object`) to type `Integer`. The casting operation will succeed only if the object referenced by `aThing` is, in fact, type `Integer`; if not, a `ClassCastException` will be thrown.

What is the advantage of performing the cast? Casting gives us a type `Integer` reference to the object in Figure 1.7 that can be processed just like any other type `Integer` reference.

The expression

```
((Integer) aThing).intValue()
```

will compile because now `intValue` is applied to a type `Integer` reference. Note that all parentheses are required so that method `intValue` will be invoked after the cast. Similarly, the assignment statement

```
Integer aNum = (Integer) aThing;
```

is valid because a type `Integer` reference is being assigned to `aNum` (type `Integer`).

Keep in mind that the casting operation does not change the object referenced by `aThing`; instead, it tells the compiler to treat the reference as if it was a type `Integer` reference. We can then invoke any instance method of class `Integer` and process the object just like any other type `Integer` object.

The cast

```
(Integer) aThing
```

is called a downcast because we are casting from a higher type (`Object`) to a lower type (`Integer`). It is analogous to a narrowing cast when dealing with primitive types:

```
double x = ... ;
int count = (int) x; // Narrowing cast, double is wider type than int
```

You can downcast from a more general type (a superclass type) to a more specific type (a subclass type) in a class hierarchy, provided that the more specific type is the same type as the

object being cast (e.g., (Integer) aThing). You can also downcast from a more general type to a more specific type that is a superclass of the object being cast (e.g., (Number) aThing). *Upcasts* (casting from a more specific type to a more general type) are always valid; however, they are unnecessary and are rarely done.

PITFALL

Performing an Invalid Cast

Assume that aThing (type Object) references a type Integer object as before, and you want to get its string representation. The downcast

```
(String) aThing // Invalid cast
```

is invalid and would cause a ClassCastException (a subclass of RuntimeException) because aThing references a type Integer object, and a type Integer object cannot be downcast to type String (String is not a superclass of Integer). However, the method call aThing.toString() is valid (and returns a string) because type Object has a toString method. (Which toString method would be called: Object.toString or Integer.toString?)

Using instanceof to Guard a Casting Operation

In the preceding Pitfall, we mentioned that a ClassCastException occurs if we attempt an invalid casting operation. Java provides the instanceof operator, which you can use to guard against this kind of error.

EXAMPLE 1.7 The following array stuff can store 10 objects of any data type because every object type is a subclass of Object.

```
Object[] stuff = new Object[10];
```

Assume that the array stuff has been loaded with data, and we want to find the sum of all numbers that are wrapped in objects. We can use the following loop to do so:

```
double sum = 0;
for (int i = 0; i < stuff.length; i++) {
    if (stuff[i] instanceof Number) {
        Number next = (Number) stuff[i];
        sum += next.doubleValue();
    }
}
```

The if condition (stuff[i] instanceof Number) is true if the object referenced by stuff[i] is a subclass of Number. It would be false if stuff[i] referenced a String or other nonnumeric object. The statement

```
Number next = (Number) stuff[i];
```

casts the object referenced by stuff[i] (type Object) to type Number and then references it through variable next (type Number). The variable next contains a reference to the same object as does stuff[i], but the type of the reference is different (type Number instead of type Object). Then the statement

```
sum += next.doubleValue();
```

invokes the appropriate `doubleValue` method to extract the numeric value and add it to `sum`. Rather than declare variable `next`, you could write the if statement as

```
if (stuff[i] instanceof Number)
    sum += ((Number) stuff[i]).doubleValue();
```

PROGRAM STYLE

Polymorphism Eliminates Nested `if` Statements

If Java didn't support polymorphism, the `if` statement in Example 1.7 would be much more complicated. You would need to write something like the following:

```
// Inefficient code that does not take advantage of polymorphism
if (stuff[i] instanceof Integer)
    sum += ((Integer) stuff[i]).doubleValue();
else if (stuff[i] instanceof Double)
    sum += ((Double) stuff[i]).doubleValue();
else if (stuff[i] instanceof Float)
    sum += ((Float) stuff[i]).doubleValue();
...
```

Each condition here uses the `instanceof` operator to determine the data type of the actual object referenced by `stuff[i]`. Once the type is known, we cast to that type and call its `doubleValue` method. Obviously, this code is very cumbersome and is more likely to be flawed than the original **`if`** statement. More importantly, if a new wrapper class is defined for numbers, we would need to modify the **`if`** statement to process objects of this new class type. So be wary of selection statements like the one shown here; their presence often indicates that you are not taking advantage of polymorphism.

EXAMPLE 1.8 Suppose we have a class `Employee` with the following data fields:

```
public class Employee {
    // Data Fields
    private String name;
    private double hours;
    private double rate;
    private Address address;
...
```

To determine whether two `Employee` objects are equal, we could compare all four data fields. However, it makes more sense to determine whether two objects are the same employee by comparing their name and address data fields. Below, we show a method `equals` that overrides the `equals` method defined in class `Object`. By overriding this method, we ensure that the `equals` method for class `Employee` will always be called when method `equals` is applied to an `Employee` object. If we had declared the parameter type for `Employee.equals` as type `Employee` instead of `Object`, then the `Object.equals` method would be called if the argument was any data type except `Employee`.

```
/** Determines whether the current object matches its argument.
    @param obj The object to be compared to the current object
    @return true if the objects have the same name and address;
            otherwise, return false
 */
@Override
public boolean equals(Object obj) {
```

```
        if (obj == this) return true;
        if (obj == null) return false;
        if (this.getClass() == obj.getClass()) {
            Employee other = (Employee) obj;
            return name.equals(other.name) &&
                   address.equals(other.address);
        } else {
            return false;
        }
    }
```

If the object referenced by `obj` is not type `Employee`, we return `false`. If it is type `Employee`, we downcast that object to type `Employee`. After the downcast, the return statement calls method `String.equals` to compare the name field of the current object to the name field of object `other`, and method `Address.equals` to compare the two `address` data fields. Therefore, method `equals` must also be defined in class `Address`. The method result is `true` if both the `name` and `address` fields match, and it is `false` if one or both fields do not match. The method result is also `false` if the downcast can't be performed because the argument is an incorrect type or **`null`.**

The Class Class

Every class has a `Class` object that is automatically created when the class is loaded into an application. The `Class` class provides methods that are mostly beyond the scope of this text. The important point is that each `Class` object is unique for the class, and the `getClass` method (a member of `Object`) will return a reference to this unique object. Thus, if `this.getClass() == obj.getClass()` in Example 1.8 is true, then we know that `obj` and `this` are both of class `Employee`.

EXERCISES FOR SECTION 1.5

SELF-CHECK

1. Indicate whether each statement below is valid and explain what the valid ones do.

```
Object o = new String("Hello");
String s = o;
Object p = 25;
int k = p;
Number n = k;
```

2. If Java did not support autoboxing and unboxing, what statements in Question 1 would become invalid?

3. Rewrite the invalid statements found in Question 2 to remove compile-time errors.

PROGRAMMING

1. Write an `equals` method for class `Computer` (Listing 1.2).

2. Write an `equals` method for class `Notebook` (Listing 1.4).

3. Write an `equals` method for the following class. What other `equals` methods should be defined?

```
public class Airplane {
    // Data Fields
    Engine eng;
    Rudder rud;
    Wing[] wings = new Wing[2];
    ...
}
```

1.6 A Java Inheritance Example—The `Exception` Class Hierarchy

Next, we show how Java uses inheritance to build a class hierarchy that is fundamental to detecting and correcting errors during program execution (run-time errors). A run-time error occurs during program execution when the JVM detects an operation that it knows to be incorrect. A run-time error will cause the JVM to *throw an exception*—that is, to create an object of an exception type that identifies the kind of incorrect operation and interrupts normal processing. Table 1.3 shows some examples of exceptions that are run-time errors. All are subclasses of class `RuntimeException`. Following are some examples of the exceptions listed in the table.

Division by Zero

If `count` represents the number of items being processed and it is possible for `count` to be zero, then the assignment statement

```
average = sum / count;
```

can cause a division-by-zero error. If `sum` and `count` are **`int`** variables, this error is indicated by the JVM throwing an `ArithmeticException`. You can easily guard against such a division with an `if` statement so that the division operation will not be performed when `count` is zero:

```
if (count == 0)
    average = 0;
else
    average = sum / count;
```

Normally, you would compute an average as a `double` value, so you could cast an `int` value in `sum` to type `double` before doing the division. In this case, an exception is not thrown if `count` is zero. Instead, `average` will have one of the special values `Double.POSITIVE_INFINITY`, `Double.NEGATIVE_INFINITY`, or `Double.NaN` depending on whether `sum` was positive, negative, or zero.

Array Index Out of Bounds

An `ArrayIndexOutOfBoundsException` is thrown by the JVM when an index value (subscript) used to access an element in an array is less than zero or greater than or equal to the array's length. For example, suppose we define the array `scores` as follows:

```
int[] scores = new int[500];
```

The subscripted variable `scores[i]` uses `i` (type **`int`**) as the array index. An `ArrayIndexOutOfBoundsException` will be thrown if `i` is less than zero or greater than 499.

TABLE 1.3
Subclasses of `java.lang.RuntimeException`

Class	Cause/Consequence
`ArithmeticException`	An attempt to perform an integer division by zero
`ArraylndexOutOfBoundsException`	An attempt to access an array element using an index (subscript) less than zero or greater than or equal to the array's length
`NumberFormatException`	An attempt to convert a string that is not numeric to a number
`NullPointerException`	An attempt to use a `nul1` reference value to access an object
`NoSuchElementException`	An attempt to get a next element after all elements were accessed
`InputMismatchException`	The token returned by a data entry method does not match the pattern for the expected data type

Array index out of bounds errors can be prevented by carefully checking the boundary values for an index that is also a loop control variable. A common error is using the array size as the upper limit rather than the array size minus 1.

EXAMPLE 1.9 The following loop would cause an `ArrayIndexOutOfBoundsException` on the last pass, when `i` is equal to `x.length`.

```
for (int i = 0; i <= x.length; i++)
    x[i] = i * i;
```

The loop repetition test should be `i < x.length` .

NumberFormatException and InputMismatchException

The `NumberFormatException` is thrown when a program attempts to convert a nonnumeric string (usually a data value) to a numeric value. For example, if the user types in the string `"2.6e"`, method `parseDouble` in the following code

```
String speedStr = JOptionPane.showInputDialog("Enter speed");
double speed = Double.parseDouble(speedStr);
```

would throw a `NumberFormatException` because `"2.6e"` is not a valid numeric string (it has no exponent after the `e`). There is no general way to avoid this exception because it is impossible to guard against all possible data entry errors the user can make.

A similar error can occur if you are using a `Scanner` object for data entry. If `scan` is a `Scanner`, the statement

```
double speed = scan.nextDouble();
```

will throw an `InputMismatchException` if the next token scanned is `"2.6e"`.

Null Pointer

The `NullPointerException` is thrown when there is an attempt to access an object that does not exist; that is, the reference variable being accessed contains a special value, known as `null`. You can guard against this by testing for `null` before invoking a method.

The Exception Class Hierarchy

The exceptions in Table 1.3 are all subclasses of `RuntimeException`. All `Exception` classes are defined within a class hierarchy that has the class `Throwable` as its superclass (see the UML diagram in Figure 1.8). The UML diagram shows that classes `Error` and `Exception` are subclasses of `Throwable`. Each of these classes has subclasses that are shown in the figure. We will focus on class `Exception` and its subclasses in this chapter. Because `RuntimeException` is a subclass of `Exception`, it is also a subclass of `Throwable` (the subclass relationship is transitive).

The Class Throwable

The class `Throwable` is the superclass of all exceptions. The methods that you will use from class `Throwable` are summarized in Table 1.4. Because all `Exception` classes are subclasses of class `Throwable`, they can call any of its methods including `getMessage`, `printStackTrace`, and `toString`. If `ex` is an `Exception` object, the call

```
ex.printStackTrace();
```

displays a stack trace, discussed in Appendix A (Section A.11). The statement

```
System.err.println(ex.getMessage());
```

FIGURE 1.8
Summary of `Exception` Class Hierarchy

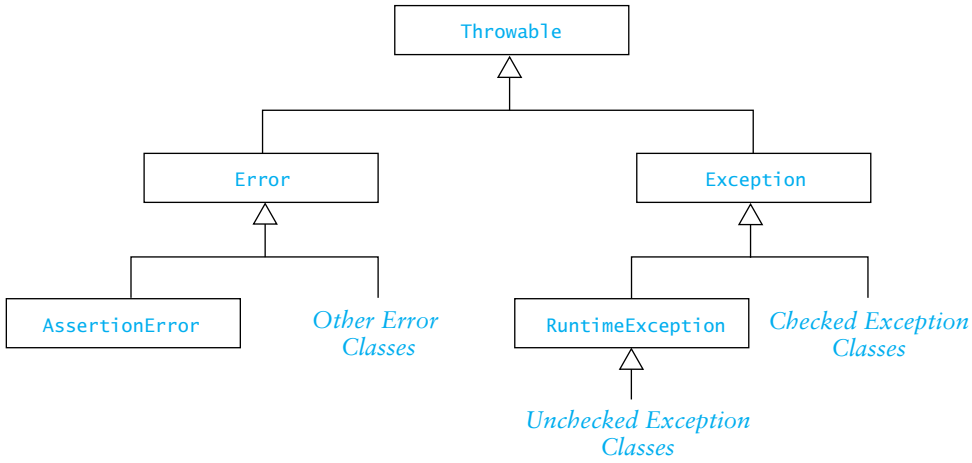


TABLE 1.4
Summary of Commonly Used Methods from the `java.lang.Throwable` Class

Method	Behavior
`String getMessage()`	Returns the detail message
`void printStackTrace()`	Prints the stack trace to `System.err`
`String toString()`	Returns the name of the exception followed by the detail message

displays a detail message (or error message) describing the exception. The statement

```
System.err.println(ex.toString);
```

displays the name of the exception followed by the detail message.

Checked and Unchecked Exceptions

There are two categories of exceptions: *checked* and *unchecked*. A checked exception is an error that is normally not due to programmer error and is beyond the control of the programmer. All exceptions caused by input/output errors are considered checked exceptions. For example, if the programmer attempts to access a data file that is not available because of a user or system error, a `FileNotFoundException` is thrown. The class `IOException` and its subclasses (see Table 1.5) are checked exceptions. Even though checked exceptions are beyond the control of the programmer, the programmer must be aware of them and must handle them in some way (discussed later). All checked exceptions are subclasses of `Exception`, but they are not subclasses of `RuntimeException`. Figure 1.9 is a more complete diagram of the Exception hierarchy.

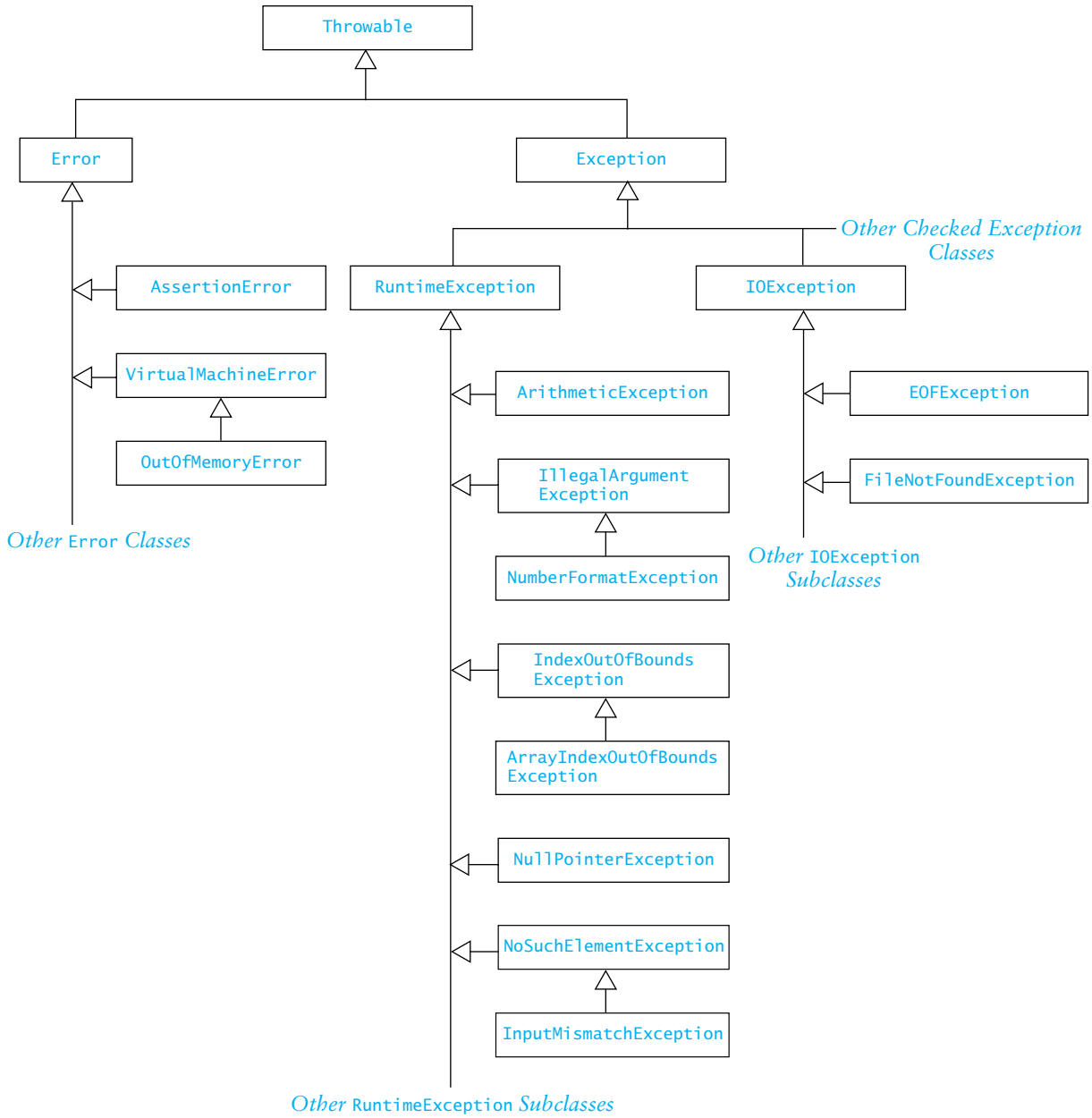
The *unchecked* exceptions represent error conditions that may occur as a result of programmer error or of serious external conditions that are considered unrecoverable. For example, exceptions such as `NullPointerException` or `ArrayIndexOutOfBoundsException` are unchecked exceptions that are generally due to programmer error. These exceptions are all subclasses of `RuntimeException`. While you can sometimes prevent these exceptions via defensive programming, it is impractical to try to prevent them all or to provide exception handling for all of them. Therefore, you can handle these exceptions, but Java does not require you to do so.

The class `Error` and its subclasses represent errors that are due to serious external conditions. An example of such an error is `OutOfMemoryError`, which is thrown when there is no memory available. You can't foresee or guard against these kinds of errors. You can attempt to handle

TABLE 1.5
Class `java.io.IOException` and Some Subclasses

Exception Class	Cause
`IOException`	Some sort of input/output error
`EOFException`	Attempt to read beyond the end of data with a `DataInputStream`
`FileNotFoundException`	Inability to find a file

FIGURE 1.9
Exception Hierarchy Showing Selected Checked and Unchecked Exceptions



these exceptions, but you are strongly discouraged from trying to do so because you probably will be unsuccessful. For example, if an `OutOfMemoryError` is thrown, there is no memory available to process the exception-handling code, so the exception would be thrown again.

How do we know which exceptions are checked and which are unchecked? `Exception` classes that are subclasses of `RuntimeException` and `Error` are unchecked. All other `Exception` classes are checked exceptions.

We discuss Java exceptions further in Appendix A. Section A.11 describes how to use the **`try-catch`** statement to handle different kinds of exceptions; Section A.12 shows how to write statements that throw exceptions when your code detects an error during run time.

Handling Exceptions to Recover from Errors

Exceptions enable Java programmers to write code that can report errors and sometimes recover from them. The key to this process is the **`try-catch`** sequence. We will cover the essentials of catching and throwing exceptions in this section. A complete discussion is provided in Sections A.11 and A.12.

The `try-catch` Sequence

The **`try-catch`** sequence is used to catch and handle exceptions. It resembles an **`if-then-else`** statement. It consists of one **`try`** block followed by one or more **`catch`** clauses. The statements in the **`try`** block are executed first. If they execute without error, the **`catch`** clauses are skipped. If a statement in the **`try`** block throws an exception, the rest of the statements in the **`try`** block are skipped and execution continues with the statements in the **`catch`** clause for that particular type of exception. If there is no **`catch`** clause for that exception type, the exception is rethrown to the calling method. If the `main` method is reached and no appropriate **`catch`** clause is located, the program terminates with an unhandled exception error. A **`try`** block with two **`catch`** clauses follows.

```
try {
    // Execute the following statements until an exception is thrown
    ...
    // Skip the catch blocks if no exceptions were thrown
} catch (ExceptionTypeA ex) {
    // Execute this catch block if an exception of type ExceptionTypeA
    // was thrown in the try block
...
} catch (ExceptionTypeB ex) {
    // Execute this catch block if an exception of type ExceptionTypeB
    // was thrown in the try block
    ...
}
```

A **`catch`** clause header resembles a method header. The expression in parentheses in the **`catch`** clause header is like a method parameter declaration (the parameter is `ex`). The statements in curly braces, the **`catch`** block, execute if the exception that was thrown is the specified exception type or is a subclass of that exception type.

PITFALL

Unreachable `catch` Block

In the above, `ExceptionTypeA` cannot be a superclass of `ExceptionTypeB`. If it is, `ExceptionTypeB` is considered unreachable because its exceptions would be caught by the first `catch` clause.

Using try-catch to Recover from an Error

One common source of exceptions is user input. For example, the `Scanner` `nextInt` method is supposed to read a type **`int`** value. If an **`int`** is not the next item read, Java throws an `InputMismatchException`. Rather than have this problem terminate the program, you can read the data value in a **`try`** block and catch an `InputMismatchException` in the **`catch`** clause. If one is thrown, you can give the user another chance to enter an integer as shown in method `getIntValue`, as follows:

```
/** Reads an integer using a scanner.
    @return the first integer read.
*/
public static int getIntValue(Scanner scan) {
    int nextInt = 0;          // next int value
    boolean validInt = false; // flag for valid input
    while (!validInt) {
        try {
            System.out.println("Enter number of kids:");
            nextInt = scan.nextInt();
            validInt = true;
        }  catch (InputMismatchException ex) {
            scan.nextLine();   // clear buffer
            System.out.println ("Bad data -- enter an integer:");
        }
    }
    return nextInt;
}
```

The **`while`** loop repeats while `validInt` is **`false`** (its initial value). The **`try`** block attempts to read a type **`int`** value using `Scanner` scan. If the user enters an integer, `validInt` is set to **`true`** and the **`try-catch`** statement and **`while`** loop are exited. The integer data value will be returned as the method result.

If the user enters a data item that is not an integer, however, Java throws an `InputMismatchException`. This is caught by the **`catch`** clause

```
catch (InputMismatchException ex)
```

The first statement in the **`catch`** block clears the `Scanner` buffer, and the user is prompted to enter an integer. Because `validInt` is still `false`, the `while` loop repeats until the user successfully enters an integer.

Throwing an Exception When Recovery Is Not Obvious

In the last example, method `getIntValue` was able to recover from a bad data item by giving the user another chance to enter data. In some cases, you may be able to write code that detects certain kinds of errors, but there may not be an obvious way to recover from them. In these cases, the best approach is just to throw an exception reporting the error to the method that called it. The caller can then catch the exception and handle it.

Method `processPositiveInteger` requires a positive integer as its argument. If the argument is not positive, there is no reason to continue executing the method because the result may be meaningless, or the method execution may cause a different exception to be thrown, which could confuse the method caller. There is also no obvious way to correct this error because the method has no way of knowing what `n` should be, so a **`try-catch`** sequence would not fix the problem.

```
public static void processPositiveInteger(int n) {
    if (n < 0)
```

```
            throw new IllegalArgumentException(
                    "Invalid negative argument");
        else {
             // Process n as required
             //...
             System.out.println("Finished processing " + n);
        }
    }
```

If the argument `n` is not positive, the statement

```
throw new IllegalArgumentException("Invalid negative argument");
```

executes and throws an `IllegalArgumentException` object. The string in the last line is stored in the exception object's `message` data field, and the method is exited, returning control to the caller. The caller is then responsible for handling this exception. If possible, the caller may be able to recover from this error and would attempt to do so.

The `main` method, which follows, calls both `getIntValue` and `processPositiveInteger` in the **`try`** block. If an `IllegalArgumentException` is thrown, the message `invalid negative argument` is displayed, and the program terminates with an error indication. If no exception is thrown, the program exits normally.

```
public static void main(String[] args) {
    Scanner scan = new Scanner(System.in);
    try {
        int num = getIntValue(scan);
        processPositiveInteger(num);
    } catch (IllegalArgumentException ex) {
        System.err.println(ex.getMessage());
        System.exit(1); // error indication
    }
    System.exit(0); //normal exit
}
```

EXERCISES FOR SECTION 1.6

SELF-CHECK

1. Explain the key difference between checked and unchecked exceptions. Give an example of each kind of exception. What criterion does Java use to decide whether an exception is checked or unchecked?
2. What is the difference between the kind of unchecked exceptions in class `Error` and the kind in class `Exception`?
3. List four subclasses of `RuntimeException`.
4. List two subclasses of `IOException`.
5. What happens in the `main` method preceding the exercises if an exception of a different type occurs in method `processPositiveInteger`?
6. Trace the execution of method `getIntValue` if the following data items are entered by a careless user. What would be displayed?

```
ace
7.5
-5
```

7. Trace the execution of method `main` preceding the exercises if the data items in Question 6 were entered. What would be displayed?

1.7 Packages and Visibility

Packages

You have already seen packages. The Java API is organized into packages such as `java.lang`, `java.util`, `java.io`, and `javax.swing`. The package to which a class belongs is declared by the first statement in the file in which the class is defined using the keyword **package**, followed by the package name. For example, we could begin each class in the computer hierarchy (class `Notebook` and class `Computer`) with the line

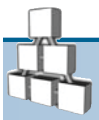
```
package computers;
```

All classes in the same package are stored in the same directory or folder. The directory must have the same name as the package. All the classes in the folder must declare themselves to be in the package.

Classes that are not part of a package may access only public members (data fields or methods) of classes in the package. If the application class is not in the package, it must reference the classes by their complete names. The complete name of a class is `packageName.className`. However, if the package is imported by the application class, then the prefix `packageName` is not required. For example, we can reference the constant `GREEN` in class `java.awt.Color` as `Color.GREEN` if we import package `java.awt`. Otherwise, we would need to use the complete name `java.awt.Color.GREEN`.

The No-Package-Declared Environment

So far we have not specified packages, yet objects of one class could communicate with objects of another class. How does this work? Just as there is a default visibility, there is a default package. Files that do not specify a package are considered part of the default package. Therefore, if you don't declare packages, all your classes belong to the same package (the default package).



SYNTAX Package Declaration

FORM:

`package` *packageName*`;`

EXAMPLE:

```
package computers;
```

INTERPRETATION:

This declaration appears as the first line of the file in which a class is defined. The class is now considered part of the package. This file must be contained in a folder with the same name as the package.

PROGRAM STYLE

When to Package Classes

The default package facility is intended for use during the early stages of implementing classes or for small prototype programs. If you are developing an application that has several classes that are part of a hierarchy of classes, you should declare them all to be in the same package. The package declaration will keep you from accidentally referring to classes by their short names in other classes that are outside the package. It will also restrict the visibility of protected members of a class to only its subclasses outside the package (and to other classes inside the package) as intended.

Package Visibility

So far, we have discussed three layers of visibility for classes and class members (data fields and methods): private, protected, and public. There is a fourth layer, called *package visibility*, that sits between private and protected. Classes, data fields, and methods with package visibility are accessible to all other methods of the same package but are not accessible to methods outside of the package. By contrast, classes, data fields, and methods that are declared protected are visible within subclasses that are declared outside the package, in addition to being visible to all members of the package.

We have used the visibility modifiers `private`, `public`, and `protected` to specify the visibility of a class member. If we do not use one of these visibility modifiers, then the class member has package visibility and it is visible in all classes of the same package, but not outside the package. Note that there is no visibility modifier `package`; package visibility is the default if no visibility modifier is specified.

Visibility Supports Encapsulation

The rules for visibility control how encapsulation occurs in a Java program. Table 1.6 summarizes the rules in order of decreasing protection. Note that private visibility is for members of a class that should not be accessible to anyone but the class, not even classes that extend it. Except for inner classes, it does not make sense for a class to be private. It would mean that no other class can use it.

Also, note that package visibility (the default if a visibility modifier is not given) allows the developer of a library to shield classes and class members from classes outside the package. Typically, such classes perform tasks required by the public classes within the package.

Use of protected visibility allows the package developer to give control to other programmers who want to extend classes in the package. Protected data fields are typically essential to an object. Similarly, protected methods are those that are essential to an extending class.

Table 1.6 shows that public classes and members are universally visible. Within a package, the public classes are those that are essential to communicating with objects outside the package.

TABLE 1.6
Summary of Kinds of Visibility

Visibility	Applied to Classes	Applied to Class Members
`private`	Applicable to inner classes. Accessible only to members of the class in which it is declared	Visible only within this class
Default or package	Visible to classes in this package	Visible to classes in this package
`protected`	Applicable to inner classes. Visible to classes in this package and to classes outside the package that extend the class in which it is declared	Visible to classes in this package and to classes outside the package that extend this class
`public`	Visible to all classes	Visible to all classes. The class defining the member must also be public

PITFALL

Protected Visibility Can Be Equivalent to Public Visibility

The intention of protected visibility is to enable a subclass to access a member (data field or method) of a superclass directly. However, protected members can also be accessed within any class that is in the same package. This is not a problem if the class with the protected members is declared to be in a package; however, if it is not, then it is in the default package. Protected members of a class in the default package are visible in all other classes you have defined that are not part of an actual package. This is generally not a desirable situation. You can avoid this dilemma by using protected visibility only with members of classes that are in explicitly declared packages. In all other classes, use either public or private visibility because protected visibility is virtually equivalent to public visibility.

EXERCISES FOR SECTION 1.7

SELF-CHECK

1. Consider the following declarations:

```
package pack1;
public class Class1 {
    private int v1;
    protected int v2;
    int v3;
    public int v4;
}

package pack1;
public class Class2 {...}

package pack2;
public class Class3 extends pack1.Class1 {...}
```

```
package pack2;
public class Class4 {...}
```

a. What visibility must variables declared in `pack1.Class1` have in order to be visible in `pack1.Class2`?

b. What visibility must variables declared in `pack1.Class1` have in order to be visible in `pack2.Class3`?

c. What visibility must variables declared in `pack1.Class1` have in order to be visible in `pack2.Class4`?

1.8 A Shape Class Hierarchy

In this section, we provide a case study that illustrates some of the principles in this chapter. For each case study, we will begin with a statement of the problem (Problem). Then we analyze the problem to determine exactly what is expected and to develop an initial strategy for solution (Analysis). Next, we design a solution to the problem, developing and refining an algorithm (Design). We write one or more Java classes that contain methods for the algorithm steps (Implementation). Finally, we provide a strategy for testing the completed classes and discuss special cases that should be investigated (Testing). We often provide a separate class that does the testing.

CASE STUDY Processing Geometric Figures

Problem We would like to process some standard geometric shapes. Each figure object will be one of three standard shapes (rectangle, circle, and right triangle). We would like to be able to do standard computations, such as finding the area and perimeter, for any of these shapes.

Analysis For each of the geometric shapes we can process, we need a class that represents the shape and knows how to perform the standard computations on it (i.e., find its area and perimeter). These classes will be `Rectangle`, `Circle`, and `RtTriangle`. To ensure that these shape classes all define the required computational methods (finding area and perimeter), we will make them abstract methods in the base class for the shape hierarchy. If a shape class does not have the required methods, we will get a syntax error when we attempt to compile it.

Figure 1.10 shows the class hierarchy. We used abstract class `Shape` as the base class of the hierarchy. We didn't consider using an actual class because there are no actual objects of the base class type. The single data field `shapeName` stores the kind of shape object as a `String`.

Design We will discuss the design of the `Rectangle` class here. The design of the other classes is similar and is left as an exercise. Table 1.7 shows class `Rectangle`. Class `Rectangle` has data fields `width` and `height`. It has methods to compute area and perimeter, a method to read in the attributes of a rectangular object (`readShapeData`), and a `toString` method.

FIGURE 1.10
Abstract Class Shape and Its Three Actual Subclasses

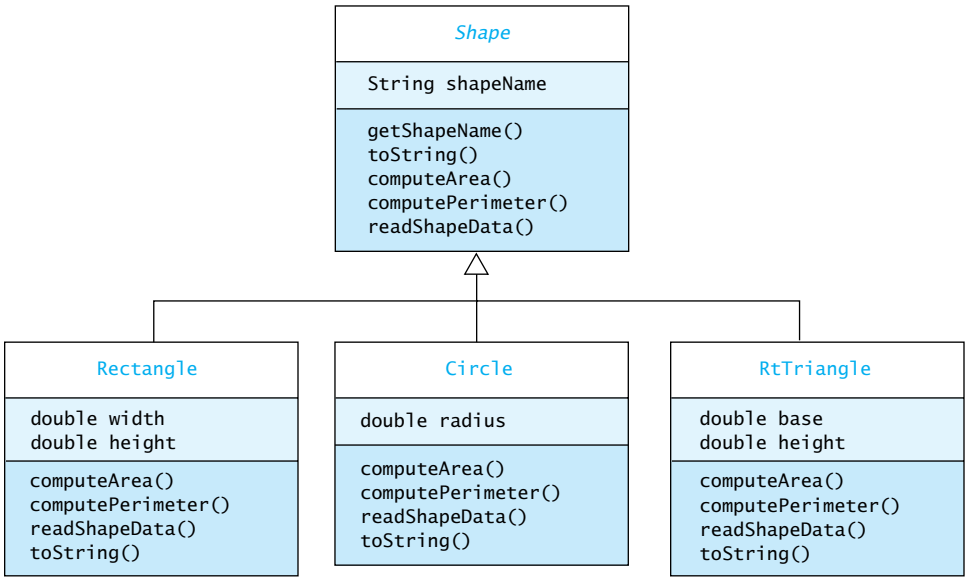


TABLE 1.7
Class Rectangle

Data Field	Attribute
`double width`	Width of a rectangle
`double height`	Height of a rectangle
Method	**Behavior**
`double computeArea()`	Computes the rectangle area (width × height)
`double computePerimeter()`	Computes the rectangle perimeter (2 × width + 2 × height)
`void readShapeData()`	Reads the width and height
`String toString()`	Returns a string representing the state

Implementation

Listing 1.6 shows abstract class `Shape`.

LISTING 1.6
Abstract Class Shape (Shape.java)

```
/** Abstract class for a geometric shape. */
public abstract class Shape {

    /** The name of the shape */
    private String shapeName = "";
    /** Initializes the shapeName.
        @param shapeName the kind of shape
     */
```

```
    public Shape(String shapeName) {
        this.shapeName = shapeName;
    }

    /** Get the kind of shape.
        @return the shapeName
     */
    public String getShapeName() { return shapeName; }

    @Override
    public String toString() { return "Shape is a " + shapeName; }

    // abstract methods
    public abstract double computeArea();
    public abstract double computePerimeter();
    public abstract void readShapeData();
}
```

Listing 1.7 shows class `Rectangle`.

LISTING 1.7
Class Rectangle (Rectangle.java)

```
import java.util.Scanner;

/** Represents a rectangle.
    Extends Shape.
 */

public class Rectangle extends Shape {

    // Data Fields
    /** The width of the rectangle. */
    private double width = 0;
    /** The height of the rectangle. */
    private double height = 0;

    // Constructors
    public Rectangle() {
        super("Rectangle");
    }

    /** Constructs a rectangle of the specified size.
        @param width the width
        @param height the height
     */
    public Rectangle(double width, double height) {
        super("Rectangle");
        this.width = width;
        this.height = height;
    }

    // Methods
    /** Get the width.
        @return The width
     */
```

```
    public double getWidth() {
        return width;
    }

    /** Get the height.
        @return The height
     */
    public double getHeight() {
        return height;
    }

    /** Compute the area.
        @return The area of the rectangle
     */
    @Override
    public double computeArea() {
        return height * width;
    }

    /** Compute the perimeter.
        @return The perimeter of the rectangle
     */
    @Override
    public double computePerimeter() {
        return 2 * (height + width);
    }

    /** Read the attributes of the rectangle. */
    @Override
    public void readShapeData() {
        Scanner in = new Scanner(System.in);
        System.out.println("Enter the width of the Rectangle");
        width = in.nextDouble();
        System.out.println("Enter the height of the Rectangle");
        height = in.nextDouble();
    }

    /** Create a string representation of the rectangle.
        @return A string representation of the rectangle
     */
    @Override
    public String toString() {
        return super.toString() + ": width is " + width + ", height is " +
               height;
    }
}
```

Testing

To test the shape hierarchy, we will write a program that will prompt for the kind of figure, read the parameters for that figure, and display the results. The code for `ComputeAreaAndPerimeter` is shown in Listing 1.8. The `main` method is very straightforward, and so is `displayResult`. The `main` method first calls `getShape`, which displays a list of available shapes and prompts the user for the choice. The reply is expected to be a single character. The nested `if` statement determines which shape instance to return. For example, if the user's choice is C (for Circle), the statement

```
return new Circle();
```

returns a reference to a new `Circle` object.

After the new shape instance is returned to `myShape` in `main`, the statement

```
myShape.readShapeData();
```

uses polymorphism to invoke the correct member function `readShapeData` to read the shape object's parameter(s). The methods `computeArea` and `computePerimeter` are then called to obtain the values of the area and perimeter. Finally, `displayResult` is called to display the result.

A sample of the output from `ComputeAreaAndPerimeter` follows.

```
Enter C for circle
Enter R for Rectangle
Enter T for Right Triangle
R
Enter the width of the Rectangle
120
Enter the height of the Rectangle
200
Shape is a Rectangle: width is 120.0, height is 200.0
The area is 24000.00
The perimeter is 640.00
```

LISTING 1.8

ComputeAreaAndPerimeter.java

```
import java.util.Scanner;

/**
   Computes the area and perimeter of selected figures.
   @author Koffman and Wolfgang
 */
public class ComputeAreaAndPerimeter {

    /** The main program performs the following steps.
        1. It asks the user for the type of figure.
        2. It asks the user for the characteristics of that figure.
        3. It computes the perimeter.
        4. It computes the area.
        5. It displays the result.
        @param args The command line arguments -- not used
     */
    public static void main(String args[]) {
        Shape myShape;
        double perimeter;
        double area;
        myShape = getShape();     // Ask for figure type
        myShape.readShapeData(); // Read the shape data
        perimeter = myShape.computePerimeter();  // Compute perimeter
        area = myShape.computeArea();            // Compute the area
        displayResult(myShape, area, perimeter); // Display the result
        System.exit(0);                          // Exit the program
  }

  /** Ask the user for the type of figure.
      @return An instance of the selected shape
   */
```

```
    public static Shape getShape() {
        var in = new Scanner(System.in);
        System.out.println("Enter C for circle");
        System.out.println("Enter R for Rectangle");
        System.out.println("Enter T for Right Triangle");
        String figType = in.next();
        if (figType.equalsIgnoreCase("c")) {
          return new Circle();
        }
        else if (figType.equalsIgnoreCase("r")) {
             return new Rectangle();
        }
        else if (figType.equalsIgnoreCase("t")) {
             return new RtTriangle();
        }
        else {
             return null;
        }
    }

   /** Display the result of the computation.
        @param area The area of the figure
        @param perim The perimeter of the figure
    */

    private static void displayResult(Shape myShape, double area, double perim) {
        System.out.println(myShape);
        System.out.printf("The area is %.2f%nThe perimeter is %.2f%n",
                          area, perim);
    }
}
```

PROGRAM STYLE

Using Factory Methods to Return Objects of Different Types

The method `getShape` is another example of a factory method because it creates a new object and returns a reference to it. The author of the `main` method does not need to know what kinds of shapes are available. Knowledge of the available shapes is confined to the `getShape` method. This function must present a list of available shapes to the user and decode the user's response to return an instance of the desired shape. If you add a new geometric shape class to the class hierarchy, you only need to modify the `if` statement in the factory method so that it can create and return an object of that type.

PROGRAM STYLE

Using Keyword var to Simplify Local Variable Declarations

In method `getShape`, we use the keyword `var` introduced in Java 10 (see Appendix A.3) in the declaration of local variable `Scanner in`. This is simpler and more readable than

```
Scanner in = new Scanner(System.in);
```

EXERCISES FOR SECTION 1.8

SELF-CHECK

1. Explain why `Shape` cannot be an actual class.
2. Explain why `Shape` cannot be an interface.

PROGRAMMING

1. Write class `Circle`.
2. Write class `RtTriangle`.

Chapter Review

- Inheritance and class hierarchies enable you to capture the idea that one thing may be a refinement or an extension of another. For example, a plant is a living thing. Such *is-a* relationships create the right balance between too much and too little structure. Think of inheritance as a means of creating a refinement of an abstraction. The entities farther down the hierarchy are more complex and less general than those higher up. The entities farther down the hierarchy may inherit data members (attributes) and methods from those farther up, but not vice versa. A class that inherits from another class **`extends`** that class.
- Encapsulation and inheritance impose structure on object abstractions. Polymorphism provides a degree of flexibility in defining methods. It loosens the structure a bit in order to make methods more accessible and useful. *Polymorphism* means "many forms." It captures the idea that methods may take on a variety of forms to suit different purposes.
- All exceptions in the `Exception` class hierarchy are derived from a common superclass called `Throwable`. This class provides methods for collecting and reporting the state of the program when an exception is thrown. The commonly used methods are `getMessage` and `toString`, which return a detail message describing what caused the exception to be thrown, and `printStackTrace`, which prints the exception and then shows the line where the exception occurred and the sequence of method calls leading to the exception.

- There are two categories of exceptions: checked and unchecked. Checked exceptions are generally due to an error condition external to the program. Unchecked exceptions are generally due to a programmer error or a dire event.
- The keyword **`interface`** defines an interface. A Java interface can be used to specify an ADT, and a Java class can be used to implement an ADT. A class that implements an interface must define the methods that the interface declares.
- The keyword **`abstract`** defines an abstract class or method. An abstract class is like an interface in that it leaves method implementations up to subclasses, but it can also have data fields and actual methods. You use an abstract class as the superclass for a group of classes in a hierarchy.
- Visibility is influenced by the package in which a class is declared. You assign classes to a package by including the statement **`package`** *packageName*; at the top of the file. You can refer to classes within a package by their direct names when the package is imported through an `import` declaration.

Java Constructs Introduced in This Chapter

`abstract`	`package`	`super.`
`extends`	`private`	`super(...)`
`instanceof`	`protected`	`this.`
`interface`	`public`	`this(...)`

Java API Classes Introduced in This Chapter

`java.lang.Byte`	`java.lang.Number`
`java.lang.Float`	`java.lang.Object`
`java.lang.Integer`	`java.lang.Short`

User-Defined Interfaces and Classes in This Chapter

`ComputeAreaAndPerimeter`	`Food`	`Student`
`Computer`	`Notebook`	`StudentInt`
`Employee`	`Rectangle`	`StudentWorker`
`EmployeeInt`	`Shape`	

Quick-Check Exercises

1. What does *polymorphism* mean, and how is it used in Java? What is method overriding? Method overloading?
2. What is a method signature? Describe how it is used in method overloading.
3. Describe the use of the keywords **`super`** and **`this`**.
4. Indicate whether each error or exception in the following list is checked or unchecked: `IOException`, `EOFException`, `VirtualMachineError`, `IndexOutOfBoundsException`, `OutOfMemoryError`.
5. When would you use an abstract class, and what should it contain?

6. An _____ specifies the requirements of an ADT as a contract between the _____ and _____; a _____ implements the ADT.
7. An interface can be implemented by multiple classes. (True/False)
8. Describe the difference between *is-a* and *has-a* relationships.
9. Which can have more data fields and methods: the superclass or the subclass?
10. You can reference an object of a _____ type through a variable of a _____ type.
11. You cast an object referenced by a _____ type to an object of a _____ type in order to apply methods of the _____ type to the object. This is called a _____.
12. The four kinds of visibility in order of decreasing visibility are _____, _____, _____, and _____.

Review Questions

1. Which method is invoked in a particular class when a method definition is overridden in several classes that are part of an inheritance hierarchy? Answer the question for the case in which the class has a definition for the method and also for the case where it doesn't.
2. Explain how assignments can be made within a class hierarchy and the role of casting in a class hierarchy. What is strong typing? Why is it an important language feature?
3. If Java encounters a method call of the following form:

   ```
   superclassVar.methodName()
   ```

   where `superclassVar` is a variable of a superclass that references an object whose type is a subclass, what is necessary for this statement to compile? During run time, will method `methodName` from the class that is the type of `superclassVar` always be invoked, or is it possible that a different method `methodName` will be invoked? Explain your answer.
4. Assume the situation in Review Question 3, but method `methodName` is not defined in the class that is the type of `superclassVar`, although it is defined in the subclass type. Rewrite the method call so that it will compile.
5. Explain the process of initializing an object that is a subclass type in the subclass constructor. What part of the object must be initialized first? How is this done?
6. What is default or package visibility?
7. Indicate what kind of exception each of the following errors would cause. Indicate whether each error is a checked or an unchecked exception.
   a. Attempting to create a `Scanner` for a file that does not exist
   b. Attempting to call a method on a variable that has not been initialized
   c. Using –1 as an array index
8. Discuss when abstract classes are used. How do they differ from actual classes and from interfaces?
9. What is the advantage of specifying an ADT as an interface instead of just going ahead and implementing it as a class?
10. Define an interface to specify an ADT `Money` that has methods for arithmetic operations (addition, subtraction, multiplication, and division) on real numbers having exactly two digits to the right of the decimal point, as well as methods for representing a `Money` object as a string and as a real number. Also, include methods `equals` and `compareTo` for this ADT.
11. Answer Review Question 10 for an ADT `Complex` that has methods for arithmetic operations on a complex number (a number with a real and an imaginary part). Assume that the same operations (+, –, *, /) are supported. Also, provide methods `toString` and `equals` for the ADT `Complex`.
12. Like a rectangle, a parallelogram has opposite sides that are parallel, but it has a corner angle, theta, that is less than 90 degrees. Discuss how you would add parallelograms to the class hierarchy for geometric shapes (see Figure 1.10). Write a definition for class `Parallelogram`.

Programming Projects

1. A veterinary office wants to store information regarding the kinds of animals it treats. Data includes diet, whether the animal is nocturnal, whether its bite is poisonous (as for some snakes), whether it flies, and so on. Use a superclass `Pet` with abstract methods and create appropriate subclasses to support about 10 animals of your choice.
2. A student is a person, and so is an employee. Create a class `Person` that has the data attributes common to both students and employees (name, social security number, age, gender, address, and telephone number) and appropriate method definitions. A student has a grade-point average (GPA), major, and year of graduation. An employee has a department, job title, and year of hire. In addition, there are hourly employees (hourly rate, hours worked, and union dues) and salaried employees (annual salary). Define a class hierarchy and write an application class that you can use to first store the data for an array of people and then display that information in a meaningful way.
3. Create a pricing system for a company that makes individualized computers, such as you might see on a website. There are two kinds of computers: notebooks and desktop computers. The customer can select the processor speed, the amount of memory, and the amount of storage. The customer can also choose a wireless network, a DVD drive or both. For notebooks, there is a choice of screen size and whether the screen is a touch screen. You should have an abstract class `Computer` and subclasses `DeskTop` and `Notebook`. Each subclass should have methods for calculating the price of a computer, given the base price plus the cost of the different options. You should have methods for calculating memory price, hard drive price, and so on. There should be a method to calculate shipping cost.
4. Write a banking program that simulates the operation of your local bank. You should declare the following collection of classes.

   Class `Account`

   Data fields: `customer` (type `Customer`), balance, `accountNumber`, `transactions` array (type `Transaction[]`). Allocate an initial `Transaction` array of a reasonable size (e.g., 100).

   Methods: `getBalance, getCustomer, toString, setCustomer`

   Class `SavingsAccount` extends `Account`

   Methods: `deposit, withdraw, addInterest`

   Class `CheckingAccount` extends `Account`

   Methods: `deposit, withdraw, addInterest`

   Class `Customer`

   Data fields: `name, address, age, telephoneNumber, customerNumber`

   Methods: `Accessors` and modifiers for data fields plus the additional abstract methods `getSavingsInterest`, `getCheckInterest`, and `getCheckCharge`.

   Classes `Senior`, `Adult`, `Student`, all these classes extend `Customer`

   Each has constant data fields `SAVINGS_INTEREST`, `CHECK_INTEREST`, `CHECK_CHARGE`, and `OVERDRAFT_PENALTY` that define these values for customers of that type, and each class implements the corresponding accessors.

   Class `Bank`

   Data field: `accounts array (type Account[])`. Allocate an array of a reasonable size (e.g., 100).

   Methods: `addAccount, makeDeposit, makeWithdrawal, getAccount`

Class `Transaction`

Data fields: `customerNumber`, `transactionType`, `amount`, `date`, and `fees` (a string describing unusual fees)

Methods: `processTran`

You need to write all these classes and an application class that interacts with the user. In the application, you should first open several accounts and then enter several transactions.

5. You have a sizable collection of music and videos and want to develop a database for storing and processing information about this collection. You need to develop a class hierarchy for your media collection that will be helpful in designing the database. Try the class hierarchy shown in Figure 1.11, where Audio and Video are media categories. Then CDs and cassette tapes would be subclasses of `Audio`, and DVDs and VHS tapes would be subclasses of `Video`.

   If you go to the video store to get a movie, you can rent or purchase only movies that are recorded on VHS tapes or DVDs. For this reason, class `Video` (and also classes `Media` and `Audio`) should be abstract classes because there are no actual objects of these types. However, they are useful classes to help define the hierarchy.

   Class `Media` should have data fields and methods common to all classes in the hierarchy. Every media object has a title, major artist, distributor, playing time, price, and so on. Class `Video` should have additional data fields for information describing movies recorded on DVDs and videotapes. This would include information about the supporting actors, the producer, the director, and the movie's rating. Class `DVD` would have specific information about DVD movies only, such as the format of the picture and special features on the disk. Figure 1.12 shows a possible class diagram for `Media`, `Video`, and subclasses of `Video`.

   Provide methods to load the media collection from a file and write it back out to a file. Also, provide a method to retrieve the information for a particular item identified by its title and a method to retrieve all your items for a particular artist.

6. Add shape classes `Square` and `EquilateralTriangle` to the figures hierarchy in Section 1.8. Modify class `ComputeAreaAndPerimeter` (Listing 1.8) to accept the new figures.

7. Complete the `Food` class hierarchy in Section 1.4. Read and store a list of your favorite foods. Show the total calories for these foods and the overall percentages of fat, protein, and carbohydrates for this list. To find the overall percentage, if an item has 200 calories and 10 percent is fat calories, then that item contributes 20 fat calories. You need to find the totals for fat calories, protein calories, and carbohydrate calories and then calculate the percentages.

8. A hospital has different kinds of patients who require different procedures for billing and approval of procedures. Some patients have insurance and some do not. Of the insured patients, some are on Medicare, some are in HMOs, and some have other health insurance plans. Develop a collection of classes to model these different kinds of patients.

9. A company has two different kinds of employees: professional and nonprofessional. Generally, professional employees have a monthly salary, whereas nonprofessional employees are paid an hourly rate. Similarly, professional employees have a certain number of days of vacation, whereas nonprofessional employees receive vacation hours based on the number of hours they have worked.

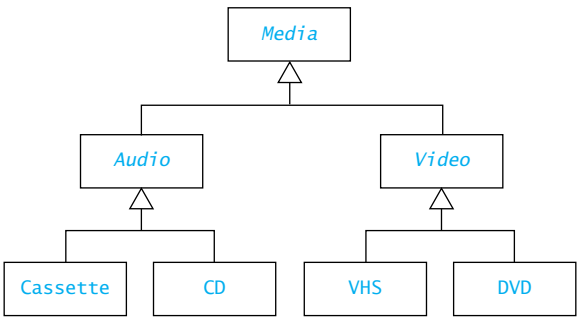


FIGURE 1.11
`Media` Class Hierarchy

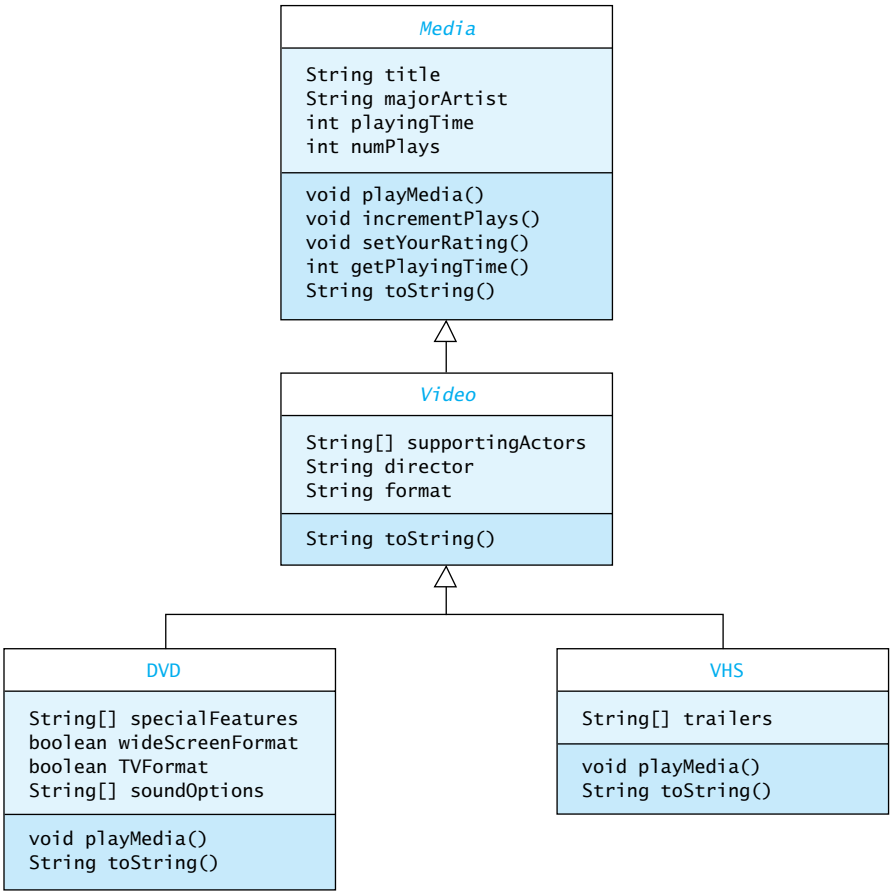


FIGURE 1.12
Class `Video` and Its Superclass and Subclasses

The amount contributed for health insurance is also different for each kind of employee. Use an abstract class `Employee` to store information common to all employees and to declare methods for calculating weekly salary and computing health care contributions and vacation days earned that week. Define subclasses `Professional` and `Nonprofessional`. Test your class hierarchy.

10. Implement class `ATMbankAmerica` in Section 1.1.
11. For the shape class hierarchy discussed in Section 1.8, consider adding classes `DrawableRectangle`, `DrawableCircle`, and so on that would have additional data fields and methods that would enable a shape to be drawn on a monitor. Provide an interface `DrawableInt` that specifies the methods required for drawing a shape. Class `DrawableRectangle`, for example, should extend `Rectangle` and implement this interface. Draw the new class hierarchy and write the new interface and classes. Use the Java Abstract Window Toolkit (AWT) to draw the objects.

Answers to Quick-Check Exercises

1. *Polymorphism* means "many forms." Method overriding means that the same method appears in a subclass and a superclass. Method overloading means that the same method appears with different signatures in the same class.
2. A signature is the form of a method determined by its name and arguments. For example, `doIt(int, double)` is the signature for a method `doIt` that has one type `int` parameter and one type `double` parameter. If several methods in a class have the same name (method overloading), Java applies the one with the same signature as the method call.

3. The keyword **`this`** followed by a dot and a name means use the named member (data field or method) of the object to which the current method is applied rather than the member with the same name declared locally in the method. The keyword **`super.`** means use the method (or data field) with this name defined in the superclass of the object, not the one belonging to the object. Using **`super(...)`** as a method call in a constructor tells Java to call a constructor for the superclass of the object being created. Similarly, using **`this(...)`** as a method call in a constructor tells Java to call another constructor for the same class but with a different parameter list. The **`super(...)`** or **`this(...)`** call must be the first statement in a subclass constructor.
4. `VirtualMachineError`, `OutOfMemoryError`, and `IndexOutOfBoundsException` are unchecked; the rest are checked.
5. An abstract class is used as a parent class for a collection of related subclasses. An abstract class cannot be instantiated. The abstract methods (identified by modifier `abstract`) defined in the abstract class act as placeholders for the actual methods. Also, you should define data fields that are common to all the subclasses in the abstract class. An abstract class can have actual methods as well as abstract methods.
6. An interface specifies the requirements of an ADT as a contract between the developer and the user; a class implements the ADT.
7. True.
8. An *is-a* relationship between classes means that one class is a subclass of a parent class. A *has-a* relationship means that one class has data members of the other class type.
9. Subclass.
10. You can reference an object of a subclass type through a variable of a superclass type.
11. You cast an object referenced by a superclass type to an object of a subclass type in order to apply methods of the subclass type to the object. This is called a *downcast*.
12. The four kinds of visibility in order of decreasing visibility are *public*, *protected*, *package*, and *private*.

Chapter 2

Lists and the Collections Framework

Chapter Objectives

- To understand the meaning of big-O notation and how it is used as a measure of an algorithm's efficiency
- To become familiar with the `List` interface and the Java Collections Framework
- To understand how to write an array-based implementation of the `List` interface
- To study the differences between single-, double-, and circular-linked list data structures
- To learn how to implement a single-linked list
- To learn how to implement the `List` interface using a double-linked list
- To understand the `Iterator` interface
- To learn how to implement the `Iterator` for a linked list
- To become familiar with the Java Collections Framework

So far, we have one data structure that you can use in your programming—the array. Giving a programmer an array and asking her to develop software systems is like giving a carpenter a hammer and asking him to build a house. In both cases, more tools are needed. The Java designers attempted to supply those tools by providing a rich set of data structures written as Java classes. The classes are all part of a hierarchy called the Java Collections Framework. We will discuss classes from this hierarchy in the rest of the book, starting in this chapter with the classes that are considered lists.

A list is an expandable collection of elements in which each element has a position or index. Some lists enable their elements to be accessed in arbitrary order (called *random access*) using a position value to select an element. Alternatively, you can start at the beginning and process the elements in sequence. We will also discuss iterators and their role in facilitating sequential access to lists.

In this chapter, we will discuss the `ArrayList` and linked lists (class `LinkedList`) and their similarities and differences. We will show that these classes are subclasses of the abstract class `AbstractList` and that they implement the `List` interface.

First, we will discuss algorithm efficiency and how to characterize the efficiency of an algorithm. You will learn about big-O notation, which you can use to compare the relative efficiency of different algorithms.

Lists and the Collections Framework

2.1 Algorithm Efficiency and Big-O

Whenever we write a new class, we will discuss the efficiency of its methods so that you know how they compare to similar methods in other classes. You can't easily measure the amount of time it takes to run a program with modern computers. When you issue the command

```
java MyProgram
```

(or click the Run button of your integrated development environment [IDE]), the operating system first loads the Java Virtual Machine (JVM). The JVM then loads the `.class` file for `MyProgram`, it then loads other `.class` files that `MyProgram` references, and finally your program executes. (If the `.class` files have not yet been created, the Java IDE will compile the source file before executing the program.) Most of the time it takes to run your program is occupied with the first two steps. If you run your program a second time immediately after the first, it may seem to take less time. This is because the operating system may have kept the files in a local memory area called a cache. However, if you have a large enough or complicated enough problem, then the actual running time of your program will dominate the time required to load the JVM and `.class` files.

Because it is very difficult to get a precise measure of the performance of an algorithm or program, we normally try to approximate the effect of a change in the number of data items, n, that an algorithm processes. In this way, we can see how an algorithm's execution time increases with respect to n, so we can compare two algorithms by examining their growth rates.

For many problems, there are algorithms that are relatively obvious but inefficient. Although every day computers are getting faster, with larger memories, there are algorithms whose growth rate is so large that no computer, no matter how fast or with how much memory, can solve the problem above a certain size. Furthermore, if a problem that has been too large to be solved can now be solved with the latest, biggest, and fastest supercomputer, adding a few more inputs may make the problem impractical, if not impossible, again. Therefore, it is important to have some idea of the relative efficiency of different algorithms. Next, we see how we might obtain such an idea by examining three methods in the following examples.

EXAMPLE 2.1 Consider the following method, which searches an array for a value:

```
public static int search(int[] x, int target) {
    for (int i = 0; i < x.length; i++) {
         if (x[i] == target)
             return i;
    }
    // target not found
    return -1;
}
```

If the target is not present in the array, the for loop body will be executed `x.length` times. If the target is present, it could be anywhere. If we consider the average overall cases where the target is present, then the loop body will execute `x.length/2` times. Therefore, the total execution time is directly proportional to `x.length`. If we doubled the size of the array, we would expect the time to double (not counting the overhead discussed earlier).

EXAMPLE 2.2 Now let us consider another problem. We want to find out whether two arrays have no common elements. We can use our `search` method to search one array for values that are in the other.

```
/** Determine whether two arrays have no common elements.
    @param x One array
    @param y The other array
    @return true if there are no common elements
*/
public static boolean areDifferent(int[] x, int[] y) {
    for (int i = 0; i < x.length; i++) {
        if (search(y, x[i]) != -1)
            return false;
    }
    return true;
}
```

The loop body will execute at most `x.length` times. During each iteration, it will call method `search` to search for current element, `x[i]`, in array `y`. The loop body in `search` will execute at most `y.length` times. Therefore, the total execution time would be proportional to the product of `x.length` and `y.length`.

EXAMPLE 2.3 Let us consider the problem of determining whether each item in an array is unique. We could write the following method:

```
/** Determine whether the contents of an array are all unique.
    @param x The array
    @return true if all elements of x are unique
*/
public static boolean areUnique(int[] x) {
    for (int i = 0; i < x.length; i++) {
        for (int j = 0; j < x.length; j++) {
            if (i != j && x[i] == x[j])
                return false;
        }
    }
    return true;
}
```

If all values are unique, the outer loop will execute `x.length` times. For each iteration of the outer loop, the inner loop will also execute `x.length` times. Thus, the total number of times the body of the inner loop will execute is $(x.length)^2$.

EXAMPLE 2.4 The method we showed in Example 2.3 is very inefficient because we do approximately twice as many tests as necessary. We can rewrite the inner loop as follows:

```
/** Determine whether the contents of an array are all unique.
    @param x The array
    @return true if all elements of x are unique
*/
public static boolean areUnique(int[] x) {
    for (int i = 0; i < x.length; i++) {
        for (int j = i + 1; j < x.length; j++) {
            if (x[i] == x[j])
                return false;
        }
    }
    return true;
}
```

We can start the inner loop index at `i+1` because we have already determined that elements preceding this one are unique. The first time, the inner loop will execute `x.length-1` times. The second time, it will execute `x.length-2` times, and so on. The last time, it will execute just once. The total number of times it will execute is

```
x.length-1 + x.length-2 +. . .+ 2 + 1
```

The series $1+2+3+\ldots+(n-1)$ is a well-known series that has a value of

$$\frac{n\times(n-1)}{2} \text{ or } \frac{n^2-n}{2}$$

Therefore, this sum is

$$\frac{x.length^2 - x.length}{2}$$

Big-O Notation

Today, the type of analysis just illustrated is more important to the development of efficient software than measuring the milliseconds in which a program runs on a particular computer. Understanding how the execution time (and memory requirements) of an algorithm grows as a function of increasing input size gives programmers a tool for comparing various algorithms and how they will perform. Computer scientists have developed a useful terminology and notation for investigating and describing the relationship between input size and execution time. For example, if the time is approximately doubled when the number of inputs, n, is doubled, then the algorithm grows at a linear rate. Thus, we say that the growth rate has an order of n. If, however, the time is approximately quadrupled when the number of inputs is doubled, then the algorithm grows at a quadratic rate. In this case, we say that the growth rate has an order of n^2.

In the previous section, we looked at four methods: one whose execution time was related to `x.length`, another whose execution time was related to `x.length` times `y.length`, one whose execution time was related to `(x.length)`^{2}, and one whose execution time was related to `(x.length)`^{2} and `x.length`. Computer scientists use the notation O(n) to represent the first case, O($n \times m$) to represent the second, and O(n^2) to represent the third and fourth, where n is `x.length` and m is `y.length`. The symbol O (which you will see in a variety of typefaces and styles in computer science literature) can be thought of as an abbreviation for "order of magnitude." This notation is called *big-O notation*.

Often, a simple way to determine the big-O of an algorithm or program is to look at the loops and to see whether the loops are nested. Assuming that the loop body consists only of simple statements, a single loop is O(n), a pair of nested loops is O(n^2), a nested loop pair inside another is O(n^3), and so on. However, you also must examine the number of times the loop executes.

Consider the following:

```
for (i = 1; i < x.length; i *= 2) {
    // Do something with x[i]
}
```

The loop body will execute $k-1$ times, with `i` having the following values: 1, 2, 4, 8, 16, 32, . . ., 2^k until 2^k is greater than `x.length`. Since $2^{k-1} \leq$ `x.length` $< 2^k$ and $\log_2 2^k$ is k, we know that $k - 1 \leq \log_2$`(x.length)` $< k$. Thus, we say that this loop is O($\log n$). The logarithm function grows slowly. The log to the base 2 of 1,000,000 is approximately 20. Typically, in analyzing the running time of algorithms, we use logarithms to the base 2.

Formal Definition of Big-O

Consider a program that is structured as follows:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      Simple Statement
    }
}
for (int k = 0; i < n; k++) {
     Simple Statement 1
     Simple Statement 2
     Simple Statement 3
     Simple Statement 4
     Simple Statement 5
}
Simple Statement 6
Simple Statement 7
...
Simple Statement 30
```

Let us assume that each *Simple Statement* takes one unit of time and that the **for** statements are free. The nested loop executes a *Simple Statement* n^2 times. Then five *Simple Statements* are executed n times in the loop with control variable k. Finally, 25 *Simple Statements* are executed after this loop. We would then conclude that the expression

$$T(n) = n^2 + 5n + 25$$

shows the relationship between processing time and n (the number of data items processed in the loop), where T(n) represents the processing time as a function of n. It should be clear that the n^2 term dominates as n becomes large.

In terms of T(n), formally, the big-O notation

$$T(n) = \mathbf{O}(f(n))$$

means that there exist two constants, n_0 and c, greater than zero, and a function, f(n), such that for all $n > n_0$, $cf(n) \geq T(n)$. In other words, as n gets sufficiently large (larger than n_0), there is some constant c for which the processing time will always be less than or equal to $cf(n)$, so $cf(n)$ is an upper bound on the performance. The performance will never be worse than $cf(n)$ and may be better.

If we can determine how the value of f(n) increases with n, we know how the processing time will increase with n. The growth rate of f(n) will be determined by the growth rate of the fastest-growing term (the one with the largest exponent), which, in this case, is the n^2 term. This means that the algorithm in this example is an O(n^2) algorithm rather than an O(n^2 + $5n$ + 25) algorithm. In general, it is safe to ignore all constants and drop the lower-order terms when determining the order of magnitude for an algorithm.

EXAMPLE 2.5 Given $T(n) = n^2 + 5n + 25$, we want to show that this is indeed $O(n^2)$. Thus, we want to show that there are constants n_0 and c such that for all $n > n_0$, $cn^2 > n^2 + 5n + 25$.

One way to do this is to find a point where

$$cn^2 = n^2 + 5n + 25$$

If we let n be n_0 and solve for c, we get

$$c = 1 + 5 / n_0 + 25 / n_0^2$$

We can evaluate the expression on the right easily when n_0 is 5(1 + 5/5 + 25/25). This gives us a c of 3. So $3n^2 > n^2 + 5n + 25$ for all n greater than 5, as shown in Figure 2.1.

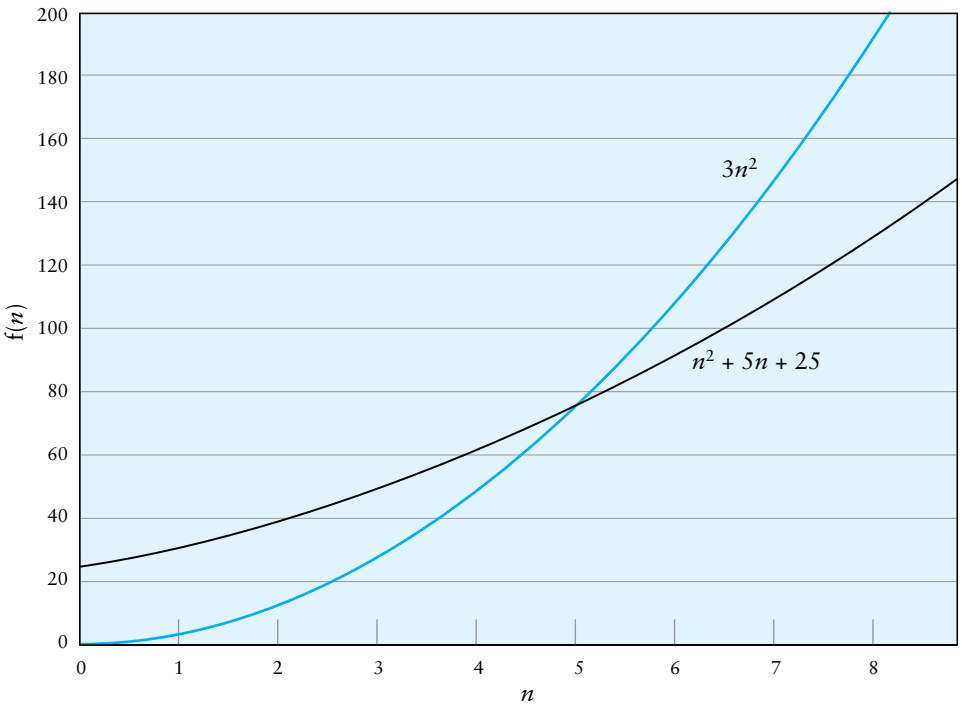


FIGURE 2.1
$3n^2$ versus $n^2 + 5n + 25$

EXAMPLE 2.6 Consider the following program loop:

```
for (int i = 0; i < n - 1; i++) {
     for (int j = i + 1; j < n; j++) {
          3 simple statements
     }
}
```

The first time through the outer loop, the inner loop is executed $n - 1$ times; the next time, $n - 2$; and the last time, once. The outer loop is executed n times. So we get the following expression for T(n):

$$3(n-1)+3(n-2)+\cdots+3(2)+3(1)$$

We can factor out the 3 to get

$$3\big((n-1)+(n-2)+\cdots+2+1\big)$$

The sum $1 + 2 + \cdots + (n - 2) + (n - 1)$ (in parentheses above) is equal to

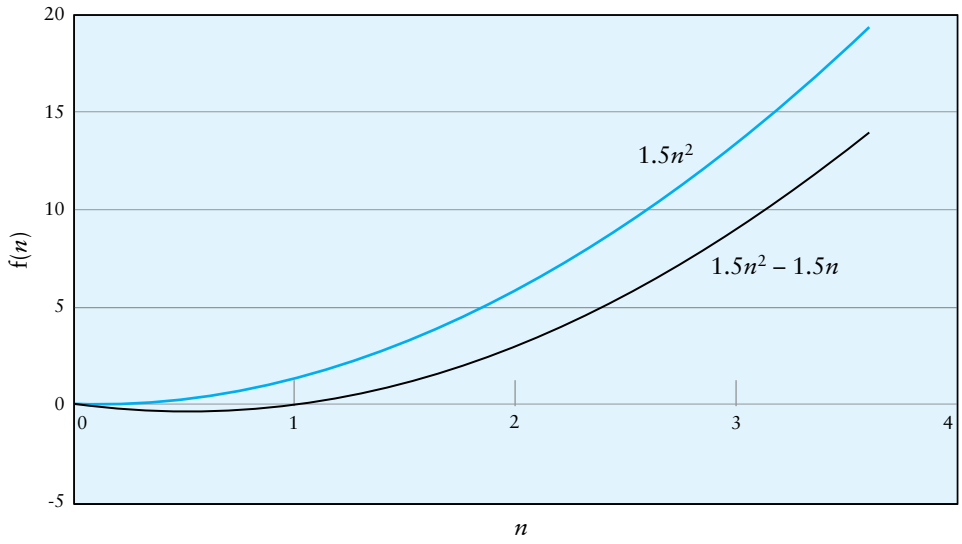
$$\frac{n^2-n}{2}$$

Thus, our final T(n) is

$$T(n)=1.5n^2-1.5n$$

This polynomial is zero when n is 1. For values greater than 1, $1.5n^2$ is always greater than $1.5n^2 - 1.5n$. Therefore, we can use 1 for n_0 and 1.5 for c to conclude that our T(n) is O(n^2) (see Figure 2.2).

FIGURE 2.2
$1.5n^2$ versus $1.5n^2 - 1.5n$



If T(n) is the form of a polynomial of degree d (the highest exponent), then it is O(n^d). A mathematically rigorous proof of this is beyond the scope of this text. An intuitive proof is demonstrated in the previous two examples. If the remaining terms have positive coefficients, find a value of n where the first term is equal to the remaining terms. As n gets bigger than this value, the n^d term will always be bigger than the remaining terms.

We use the expression O(1) to represent a constant growth rate. This is a value that doesn't change with the number of inputs. The simple steps all represent O(1). Any finite number of O(1) steps is still considered O(1).

Summary of Notation

In this section, we have used the symbols T(n), f(n), and O(f(n)). Their meaning is summarized in Table 2.1.

TABLE 2.1
Symbols Used in Quantifying Software Performance

Symbol	Meaning
T(n)	The time that a method or program takes as a function of the number of inputs, n. We may not be able to measure or determine this exactly
f(n)	Any function of n. Generally, f(n) will represent a simpler function than T(n), for example, n^2 rather than $1.5n^2 - 1.5n$
O(f(n))	Order of magnitude. O(f(n)) is the set of functions that grow no faster than f(n). We say that T(n) = O(f(n)) to indicate that the growth of T(n) is bounded by the growth of f(n)

Comparing Performance

Throughout this text, as we discuss various algorithms, we will discuss how their execution time or storage requirements grow as a function of the problem size using this big-O notation. Several common growth rates will be encountered and are summarized in Table 2.2.

TABLE 2.2
Common Growth Rates

Big-O	Name
O(1)	Constant
O(log n)	Logarithmic
O(n)	Linear
O(n log n)	Log-linear
O(n^2)	Quadratic
O(n^3)	Cubic
O(2^n)	Exponential
O(n!)	Factorial

FIGURE 2.3
Different Growth Rates

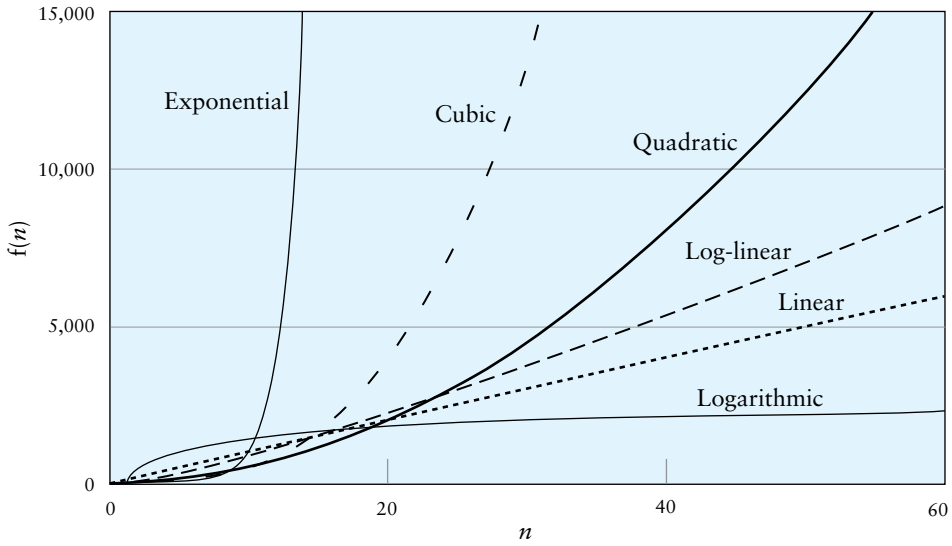


Figure 2.3 shows the growth rate of a logarithmic, a linear, a log-linear, a quadratic, a cubic, and an exponential function by plotting f(n) for a function of each type. Note that for small values of n, the exponential function is smaller than all of the others. As shown, it is not until n reaches 20 that the linear function is smaller than the quadratic. This illustrates two points. For small values of n, the less efficient algorithm may be actually more efficient. If you know that you are going to process only a limited amount of data, the $O(n^2)$ algorithm may be much more appropriate than the $O(n \log n)$ algorithm that has a large constant factor. However, algorithms with exponential growth rates can start out small but very quickly grow to be quite large.

The raw numbers in Figure 2.3 can be deceiving. Part of the reason is that big-O notation ignores all constants. An algorithm with a logarithmic growth rate $O(\log n)$ may be more complicated to program, so it may actually take more time per data item than an algorithm with a linear growth rate $O(n)$. For example, at $n = 25$, Figure 2.3 shows that the processing time is approximately 1800 units for an algorithm with a logarithmic growth rate and 2500 units for an algorithm with a linear growth rate. Comparisons of this sort are pretty meaningless. The logarithmic algorithm may actually take more time to execute than the linear algorithm for this relatively small data set. Again, what is important is the growth rate of these two kinds of algorithms, which tells you how the performance of each kind of algorithm changes with n.

EXAMPLE 2.7 Let's look at how growth rates change as we double the value of n (say, from $n = 50$ to $n = 100$). The results are shown in Table 2.3. The last column gives the ratio of processing times for the two different data sizes. For example, it shows that it will take twice as long to process 100 numbers as it would to process 50 numbers with an $O(n)$ algorithm, whereas for an $O(\log n)$ algorithm, increasing the data size from 50 to 100 would increase the processing time by a factor of only 1.18.

TABLE 2.3
Effects of Different Growth Rates

O(f(*n*))	f(50)	f(100)	f(100)/f(50)
O(1)	1	1	1
O(log *n*)	5.64	6.64	1.18
O(*n*)	50	100	2
O(*n* log *n*)	282	664	2.35
O(n^2)	2500	10,000	4
O(n^3)	12,500	100,000	8
O(2^n)	1.126×10^{15}	1.27×10^{30}	1.126×10^{15}
O(*n*!)	3.0×10^{64}	9.3×10^{57}	3.1×10^{93}

The Power of O(log *n*) Algorithms

As discussed in Example 2.7, there is a potential for great improvement by using an O(log *n*) algorithm to process a large data set versus an O(*n*) algorithm. For example, if we wanted to determine whether a particular item was present in a data set and its position in that data set using an O(*n*) algorithm, we could test every element in sequence until we found the item we were looking for. So, if the item was not present, we would have to test all *n* items before we could determine that. For a data set of size 1024, we would have to test all 1024 elements if the item we were looking for was not present.

For O(log *n*) algorithms, the size of the data set left to process decreases by a factor of 2 after each test. For example, if the data size has 1024 elements, there will only be 512 elements left to process after the first test, 256 elements after the third test, and so on. The sequence 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1 shows the size of the data set left to process after each test. Therefore, it will require 10 tests to determine that a particular element was not present in a data set of size 1024 using an O(log *n*) algorithm. We will see several examples of O(log *n*) algorithms in this book.

Algorithms with Exponential and Factorial Growth Rates

Algorithms with exponential and factorial (even faster) growth rates have an effective practical upper limit on the size of problem they can be used for, even with faster and faster computers. For example, if we have an O(2^n) algorithm that takes an hour for 100 inputs, adding the 101st input will take a second hour, adding 5 more inputs will take 32 hours (more than a day!), and adding 14 inputs will take 16,384 hours, which is almost 2 years!

This relation is the basis for cryptographic algorithms—algorithms that *encrypt* text using a special key to make it unreadable by anyone who intercepts it and does not know the key. Encryption is used to provide security for sensitive data sent over the Internet. Some cryptographic algorithms can be broken in O(2^n) time, where *n* is the number of bits in the key. A key length of 40 bits is considered breakable by a modern personal computer, but a key length of 100 (60 bits longer) is not because the key with a length of 100 bits will take approximately a billion–billion (10^{18}) times as long as the 40-bit key to crack. Modern supercomputers can crack much longer keys in reasonable time, so currently key lengths of 2048 or longer are recommended.

EXERCISES FOR SECTION 2.1

SELF-CHECK

1. Determine how many times the output statement is executed in each of the following fragments. Indicate whether the algorithm is O(n) or O(n^2).

   a.
   ```
   for (int i = 0; i < n; i++)
       for (int j = 0; j < n; j++)
           System.out.println(i + " " + j);
   ```

   b.
   ```
   for (int i = 0; i < n; i++)
       for (int j = 0; j < 2; j++)
           System.out.println(i + " " + j);
   ```

   c.
   ```
   for (int i = 0; i < n; i++)
       for (int j = n - 1; j >= i; j--)
           System.out.println(i + " " + j);
   ```

   d.
   ```
   for (int i = 1; i < n; i++)
       for (int j = 0; j < i; j++)
           if (j % i == 0)
              System.out.println(i + " " + j);
   ```

2. For the following T(n), find values of n_0 and c such that cn^3 is larger than T(n) for all n larger than n_0.

   $$T(n) = n^3 - 5n^2 + 20n - 10$$

3. How does the performance change as n goes from 1000 to 4000 for the following? Answer the same question as n goes from 1000 to 8000. Provide tables similar to Table 2.3.

   a. O(log n)
   b. O(n)
   c. O(n log n)
   d. O(n^2)
   e. O(n^3)

4. According to the plots in Figure 2.3, what are the processing times at $n = 20$ and at $n = 50$ for each of the growth rates shown?

PROGRAMMING

1. Write a program that compares the values of `y1` and `y2` in the following expressions for values of n up to 100 in increments of 10. Does the result surprise you?

   ```
   y1 = 100 * n + 10
   y2 = 5 * n * n + 2
   ```

2.2 The `List` Interface and `ArrayList` Class

An *array* is an indexed data structure, which means you can select its elements in arbitrary order as determined by the subscript value. You can also access the elements in sequence using a loop that increments the subscript. However, you can't do the following with an array object:

- Increase or decrease its length, which is fixed.
- Add an element at a specified position without shifting the other elements to make room.

- Remove an element at a specified position without shifting the other elements to fill in the resulting gap.

The classes that implement the Java `List` interface (part of Java API `java.util`) all provide methods to do these operations and more. Table 2.4 shows some of the methods in the Java `List` interface.

These methods perform the following operations:

- Return a reference to an element at a specified location (method `get`).
- Find a specified target value (method `get`).
- Add an element at the end of the list (method `add`).
- Insert an element anywhere in the list (method `add`).
- Remove an element (method `remove`).
- Replace an element in the list with another (method `set`).
- Return the size of the list (method `size`).
- Sequentially access all the list elements without having to manipulate a subscript.

The symbol `E` in Table 2.4 is a *type parameter*. Type parameters are analogous to method parameters. In the declaration of an interface or class, the type parameter represents the data type of all objects stored in a collection.

Although all of the classes we study in this chapter support the operations in Table 2.4, they do not do them with the same degree of efficiency. The kinds of operations you intend to perform in a particular application should influence your decision as to which `List` class to use.

One feature that the array data structure provides that these classes don't is the ability to store primitive-type values. The `List` classes all store references to `Objects`, so all primitive-type values must be wrapped in objects. Autoboxing facilitates this.

Figure 2.4 shows an overview of the `List` interface and the four actual classes that implement it. We study the `ArrayList` and `LinkedList` classes in this chapter; we will study the `Stack` class in Chapter 4.

Class `Vector` has been *deprecated*, which means it is included for historical reasons. Deprecated classes have been replaced by better classes and should not be used in new applications.

TABLE 2.4
Selected Methods of Interface `java.util.List<E>`

Method	Behavior
`E get(int index)`	Returns the data in the element at position `index`
`E set(int index, E anEntry)`	Stores a reference to `anEntry` in the element at position `index`. Returns the data formerly at position `index`
`int size()`	Gets the current size of the `List`
`boolean add(E anEntry)`	Adds a reference to `anEntry` at the end of the `List`. Always returns `true`
`void add(int index, E anEntry)`	Adds a reference to `anEntry`, inserting it before the item at position `index`
`int indexOf(E target)`	Searches for `target` and returns the position of the first occurrence, or –1 if it is not in the `List`
`E remove (int index)`	Removes the entry formerly at position `index` and returns it
`static of(E... elements)`	Creates an unmodifiable list of elements of type `E` (useful for testing)

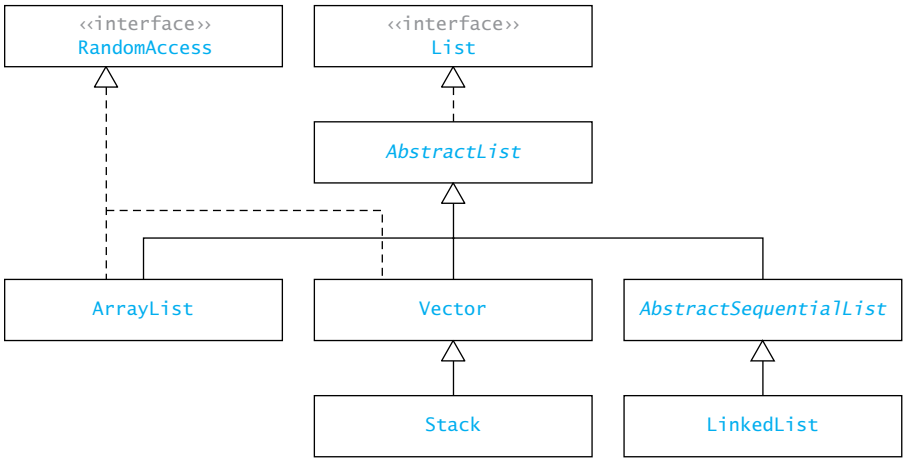


FIGURE 2.4
The `java.util.List` Interface and Its Implementers

We briefly discuss the `RandomAccess` interface and the two abstract classes `AbstractList` and `AbstractSequentialList` in Section 2.10.

The `ArrayList` Class

The simplest class that implements the `List` interface is the `ArrayList` class. An `ArrayList` object is an improvement over an array object in that it supports all of the operations just listed. `ArrayList` objects are used most often when a programmer wants to be able to grow a list by adding new elements to the end but still needs the capability to access the elements stored in the list in arbitrary order. These are the features we would need for an e-mail address book application: New entries should be added at the end, and we would also need to find e-mail addresses for entries already in the address book. The size of an `ArrayList` automatically increases as new elements are added to it, and the size decreases as elements are removed. An `ArrayList` object has an instance method size that returns its current size.

Each `ArrayList` object has a *capacity*, which is the number of elements it can store. If you add a new element to an `ArrayList` whose current size is equal to its capacity, the capacity is automatically increased.

FOR PYTHON PROGRAMMERS

The Python `list class` is similar to the Java `ArrayList` class. Both can store a collection of objects and both automatically expand when extra space is needed. Both have methods to add elements, insert elements, and get the list length. However, you cannot use array index notation (`e.g., scores[3]`) with an `ArrayList` but you can with a `Python list`.

EXAMPLE 2.8 The statements

```
List<String> yourList;
yourList = new ArrayList<>();

List<String> myList = new ArrayList<>();
```

declare `List` variables `myList` and `yourList` whose elements will reference `String` objects. The actual lists referenced by `myList` and `yourList` are `ArrayList<String>` objects. Variable `yourList` is declared as type `List` in the first statement and then it is created as an `ArrayList` in the second statement. The third statement both declares the type of `myList` (`List`) and creates it as an `ArrayList`. Initially, `myList` is empty; however, it has an initial capacity of 10 elements (the default capacity).

The statements

```
myList.add("Bashful");
myList.add("Awful");
myList.add("Jumpy");
myList.add("Happy");
```

add references to four strings as shown in the top diagram of Figure 2.5. The value of `myList.size()` is now 4.

The middle diagram of Figure 2.5 shows `ArrayList` object `myList` after the insertion of the reference to `"Doc"` at the element with subscript 2:

```
myList.add(2, "Doc");
```

The new size is 5. The strings formerly referenced by the elements with subscripts 2 and 3 are now referenced by the elements with subscripts 3 and 4. This is the same as what happens when someone cuts into a line of people waiting to buy tickets; everyone following the person who cuts in moves back one position in the line.

The bottom diagram in Figure 2.5 shows the effect of the statement

```
myList.add("Dopey");
```

which adds a reference to `"Dopey"` at the end of the `ArrayList`. The size of `myList` is now 6.

Similarly, if you remove an element from an `ArrayList` object, the size automatically decreases, and the elements following the one removed shift over to fill the vacated space. This is the same as when someone leaves a ticket line; the people in back all move forward. Here is object `myList` after using the statement

```
myList.remove(1);
```

to remove the element with subscript 1. Note that the strings formerly referenced by subscripts 2 through 5 are now referenced by subscripts 1 through 4 (in the darker color), and the size has decreased by 1.

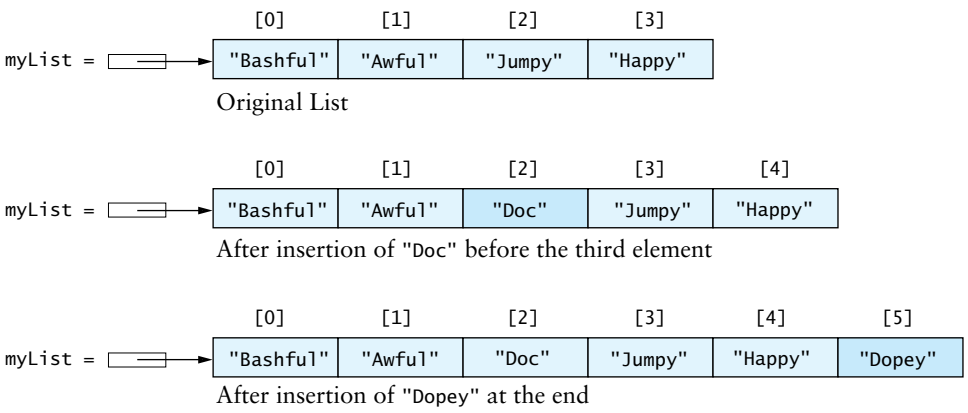
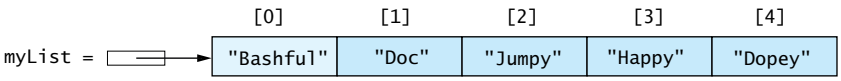


FIGURE 2.5
Insertion into an `ArrayList`



After removal of "Awful"

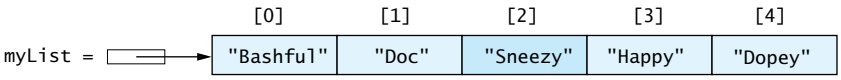
Even though an ArrayList is an indexed collection, you can't access its elements using a subscript. Instead, you use the get method to access its elements. For example, the statement

```
String dwarf = myList.get(2)
```

stores a reference to the string object "Jumpy" in variable dwarf, without changing myList. You use the set method to store a value in an ArrayList. The method call

```
myList.set(2, "Sneezy")
```

stores a reference to string "Sneezy" at index 2, replacing the reference to string "Jumpy". However, variable dwarf would still reference the string "Jumpy".



After replacing "Jumpy" with"Sneezy"

You can also search an ArrayList. The method call

```
myList.indexOf("Jumpy")
```

would return –1 after the reference to "Jumpy" was replaced, indicating an unsuccessful search. The method call

```
myList.indexOf("Sneezy")
```

would return 2.

PITFALL

Using Subscripts with an Arraylist

If you use a subscript with an ArrayList (e.g., myList[i]), you will get the syntax error array type required for [] but ArrayList found. This means that subscripts can be used only with arrays, not with indexed collections.

Generic Collections

The statement

```
List<String> myList = new ArrayList<>();
```

uses a language feature introduced in Java 5.0 called *generic collections* (or *generics*). Generics allow you to define a collection that contains references to objects of a specific type. The declaration for myList specifies that it is a List of String where String is a *type parameter*. Furthermore, myList references an ArrayList<String> object. Therefore, only references to objects of type String can be stored in myList, and all items retrieved would be of type String.

SYNTAX Creating a Generic Collection

FORM:

```
CollectionClassName<E> variable = new CollectionClassName<>();
CollectionClassName<E> variable = new CollectionClassName<E>();
var variable = new CollectionClassName<E>();
```

EXAMPLE:

```
List<Person> people = new ArrayList<>();
ArrayList<Integer> numList1 = new ArrayList<>();
var numList2 = new ArrayList<Integer>();
```

MEANING:

An initially empty *CollectionClassName*<E> object is created that can be used to store references to objects of type E (the type parameter). The actual object type stored in an object of type *CollectionClassName*<E> is specified when the object is created. If the *CollectionClassName* on the left is an interface, the *CollectionClassName* on the right must be a class that implements it. Otherwise, it must be the same class or a subclass of the one on the left.

The examples above show different ways to create an `ArrayList`. In this text, we often specify the interface name on the left of the = operator and the implementing class name on the right as shown in the first example. Since the type parameter `E` must be the same on both sides of the assignment operator, Java 7 introduced the diamond operator <>, which eliminates the need to specify the type parameter twice. We will follow this convention. In some cases, we will declare the variable type in one statement and create it in a later statement.

Java 10 introduced the keyword `var`, which allows us to simplify declaration statements as shown in the declaration of `numList2` above. The keyword `var` can be used for local variable declarations when the data type of the variable can be inferred by the compiler as discussed in the syntax display at the end of Appendix Section A.1.

PROGRAM STYLE

The examples above show different ways to create an `ArrayList`. In this text, we normally specify the interface name on the left of the = operator and the implementing class name on the right as shown in the first example in the Syntax box Creating a Generic Collection. Since the type parameter `E` must be the same on both sides of the assignment operator (=), we will also use <> rather than specify the type parameter twice. If we use `var` to declare an `ArrayList`, then the type parameter appears on the right side of =.

PITFALL

Adding Incompatible Type Objects to a Generic ArrayList

The advantage of generics is that the compiler can ensure that all operations involving objects referenced by a generic `ArrayList` are "safe" and will not cause exceptions during run time. Any type of incompatibility will be detected during compilation. If `myList` is type `ArrayList<String>`, the statement

```
myList.add(35);
```

will not compile because 35 (type `int`) is not compatible with type `String`.

Besides more strict type checking, another advantage of generics that they enable us to write classes whose methods can be used to do the same operations on objects of different types. For example, if `UnBox<E>` is a generic class with type parameter `E`, then we can use it to process a collection of `Integer` objects by passing it the type parameter `Integer` or process a collection of `String` objects by passing it the type parameter `String`.

EXERCISES FOR SECTION 2.2

SELF-CHECK

1. Describe the effect of each of the following operations on object `myList` as shown at the bottom of Figure 2.5. What is the value of `myList.size()` after each operation?

```
myList.add("Pokey");
myList.add("Campy");
int i = myList.indexOf("Happy");
myList.set(i, "Bouncy");
myList.remove(myList.size() -2);
String temp = myList.get(1);
myList.set(1, temp.toUpperCase());
```

2. For each statement(s) below, indicate whether it is valid in Java. If not, indicate how you might change it so that it will compile without error.
   a. `ArrayList<String> me = new ArrayList<String>();`
   b. `List<String> you = new ArrayList<>();`
   c. `ArrayList<Integer> them = new ArrayList<String>();`
   d. `var we = new ArrayList<String>;`
   e. `List<String> you2; you2 = new ArrayList<String>();`

3. Show three declaration statements that create a reference `myClowns` to an `ArrayList` of `Clown` objects. Use a `List` interface in one, `var` in one, and `<>` where applicable. Assume the class `Clown` has been previously declared.

4. Draw a sketch of `myClowns` like the ones in Figure 2.5 after the statement below executes.

```
myClowns.add(new Clown("Krusty", 55));
```

PROGRAMMING

1. Write the following static method:

```
/** Replaces each occurrence of oldItem in aList with newItem. */
public static void replace(ArrayList<String> aList, String oldItem,
                           String newItem)
```

2. Write the following static method:

```
/** Deletes the first occurrence of target in aList. */
public static void delete(ArrayList<String> aList, String target)
```

2.3 Applications of ArrayList

We illustrate two applications of `ArrayList` objects next. We also introduce a different version of the `for` statement.

EXAMPLE 2.9 The following statements create an `ArrayList<Integer>` object and load it with the values stored in a type `int[] array`.

```
var some = new ArrayList<Integer>();
int[] nums = {5, 7, 2, 15};
for (int numsNext : nums) {
    some.add(numsNext);
    System.out.println(some);
}
```

The statement

```
some.add(numsNext);
```

retrieves a value from array `nums (type int[])`, automatically wraps it in an `Integer` object, and stores a reference to that object in `some` (type `ArrayList<Integer>`).

The `println` statement shows how the list grows as each number is inserted.

Loop exit occurs after the last `Integer` object is stored in `some`. The output displayed by this fragment follows:

```
[5]
[5, 7]
[5, 7, 2]
[5, 7, 2, 15]
```

The `for` statement in Example 2.9 uses the enhanced `for` statement (for each). The loop header

```
for (int numsNext : nums)
```

provides sequential access to each element in array `nums`, starting with `nums[0]`. The enhanced **`for`** statement can also be used to access elements of a `List`. We talk more about this in Section 2.7.

The following fragment computes and displays the sum (29) of the `Integer` object values in `ArrayList some`.

```
int sum = 0;
for (Integer someNext : some) {
    sum += someNext;
}
System.out.println("sum is " + sum);
```

Although it may seem wasteful to carry out these operations when you already have an array of `int`s, the purpose of this example is to illustrate the steps needed to process a collection of `Integer` objects referenced by an `ArrayList<Integer>`.

SYNTAX The Enhanced `for` Loop (`for each`)

FORM:

```
for (formalParameter : expression) {. . .}
```

EXAMPLE:

```
for (String nextStr : myList) {. . .}
for (int nextInt : aList) {. . .}
```

MEANING:

During each repetition of the loop, the variable specified by *`formalParameter`* accesses the next element of *`expression`*, starting with the first element and ending with the last. The *`expression`* must evaluate to an object of a class that implements the `Iterable` interface (see Section 2.7).

A Phone Directory Application

If we want to write a program to store a list of names and phone numbers, we can use class `DirectoryEntry` to represent each item in our phone directory.

```
public class DirectoryEntry {
    String name;
    String number;
}
```

We can declare an `ArrayList<DirectoryEntry>` object to store a phone directory (`theDirectory`) with our friends' names and phone numbers:

```
private List<DirectoryEntry> theDirectory =
        new ArrayList<>();
```

We can use the statement

```
theDirectory.add(new DirectoryEntry("Jane Smith", "555-549-1234"));
```

to add an entry to `theDirectory`. If we want to retrieve the entry for a particular name (`String aName`), we can use the statements

```
int index = theDirectory.indexOf(new DirectoryEntry(aName, ""));
```

to locate the entry for the person referenced by `aName`. Method `indexOf` searches `theDirectory` by applying the `equals` method for class `DirectoryEntry` to each element of `theDirectory`. We are assuming that method `DirectoryEntry.equals` compares the `name` field of each element to the `name` field of the argument of `indexOf` (an anonymous object with the desired name). The statement

```
if (index != -1)
   dE = theDirectory.get(index);
else
   dE = null;
```

uses `ArrayList.get` to retrieve the desired entry (name and phone number) if found and stores a reference to it in `dE` (type `DirectoryEntry`). Otherwise, `null` is stored in `dE`.

EXERCISES FOR SECTION 2.3

SELF-CHECK

1. What does the following code fragment do?

```
List<Double> myList = new ArrayList<>();
myList.add(3.4);
myList.add(5.0);
myList.add(myList.get(0) + myList.get(1));
double result = myList.get(2);
System.out.println("Result is " + result);
```

2. Assume a collection of real numbers has been stored in `myList`. Write an indexed `for` loop to find the sum of all elements with an even subscript.

3. Explain why the use of an indexed `for` loop in Question 2 is more efficient than using an enhanced `for` loop to find the desired sum. How could you use an enhanced `for` loop to find the desired sum?

4. Write the code for your solution to Question 2 using an enhanced `for` loop.

PROGRAMMING

1. Write a method `addOrChangeEntry` for a class that has a data field `theDirectory` (type `ArrayList<DirectoryEntry>`) where class `DirectoryEntry is described just before the exercises`. Assume class `DirectoryEntry` has an accessor method `getNumber` and a setter method `setNumber`.

```
/** Add an entry to theDirectory or change an existing entry.
    @param aName The name of the person being added or changed
    @param newNumber The new number to be assigned
    @return The old number, or if a new entry, null
 */
public String addOrChangeEntry(String aName, String newNumber) {
```

2. Write a `removeEntry` method for the class in programming exercise 1. Use `ArrayList` methods `indexOf` and `remove`.

```
/** Remove an entry.
    @param aName The name of the person being removed
    @return The entry removed, or null if there is no entry for aName
 */
public Entry removeEntry(String aName) {
```

2.4 Implementation of an ArrayList Class

We will implement our own version of the `ArrayList` class called `KWArrayList`. Just as Java does for an `ArrayList`, we use a Java array internally to contain the data of a `KWArrayList`, as shown in Figure 2.6. The physical size of the array is indicated by the data field `capacity`. The number of data items is indicated by the data field `size`. The elements between `size` and `capacity` are available for the storage of new items.

We are assuming the following data fields in the discussion of our `KWArrayList` class. This is not exactly how it is done in Java, but it will give you a feel for how to write the class

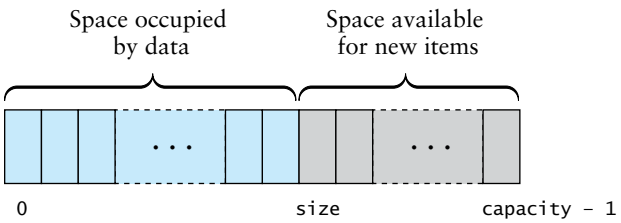


FIGURE 2.6
Internal Structure of `KWArrayList`

methods. The constructor shown in the following code allocates storage for the underlying array and initializes its capacity to 10. We will not provide a complete implementation because we expect you to use the standard `ArrayList` class provided by the Java API (package `java.util`).

We show the definition of a generic class `KWArrayList<E>` where `E` is the parameter type. The actual parameter type is specified when a generic `KWArrayList` object is declared. The data type of the references stored in the underlying array `theData` (type `E[]`) is also determined when the `KWArrayList` object is declared. If no parameter type is specified, the implicit parameter type is `Object`, and the underlying data array is type `Object[]`.

```
import java.util.*;
/** This class implements some of the methods of the Java ArrayList class. It
    does not implement the List interface.
*/
public class KWArrayList<E> {
    // Data Fields
    /** The default initial capacity */
    private static final int INITIAL_CAPACITY = 10;

    /** The underlying data array */
    private E[] theData;

    /** The current size */
    private int size = 0;

    /** The current capacity */
    private int capacity = 0;
...
```

The Constructor for Class `KWArrayList<E>`

The constructor declaration follows. Because the constructor is for a generic class, the type parameter `<E>` is implied but it must not appear in the constructor heading.

```
public KWArrayList() {
    capacity = INITIAL_CAPACITY;
    theData = (E[]) new Object[capacity];
}
```

The statement

```
theData = (E[]) new Object[capacity];
```

allocates storage for an array with type `Object` references and then casts this array object to type `E[]` so that it is type compatible with variable `theData`. Because the actual type corresponding to `E` is not known, the compiler issues the warning message: `KWArrayList.java uses unchecked or unsafe operations`. Don't be concerned about this warning—everything is fine.

PROGRAM STYLE

Java provides an annotation that enables you to compile the constructor without an error message. If you place the statement

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

before the constructor, the compiler warning will not appear.

PITFALL

Declaring a Generic Array

Rather than use the approach shown in the above constructor, you might try to create a generic array directly using the statement

```
theData = new E[capacity]; // Invalid generic array type.
```

However, this statement will not compile because Java does not allow you to create an array with an unspecified type. Remember, `E` is a type parameter that is not specified until a generic `ArrayList` object is created. Therefore, the constructor must create an array of type `Object[]` since `Object` is the superclass of all types and then downcast this array object to type `E[]`.

The add(E anEntry) Method

We implement two `add` methods with different signatures. The first appends an item to the end of a `KWArrayList`; the second inserts an item at a specified position. If `size` is less than capacity, then to append a new item:

a. Insert the new item at the position indicated by the value of `size`.

b. Increment the value of `size`.

c. Return `true` to indicate successful insertion.

This sequence of operations is illustrated in Figure 2.7. The `add` method is specified in the `Collection` interface, which is discussed in Section 2.10. The `Collection` interface is a super-interface to the `List` interface. The `add` method must return a `boolean` to indicate whether or not the insertion is successful. For an `ArrayList`, this is always true. The old value of `size` is in gray; its new value is in color.

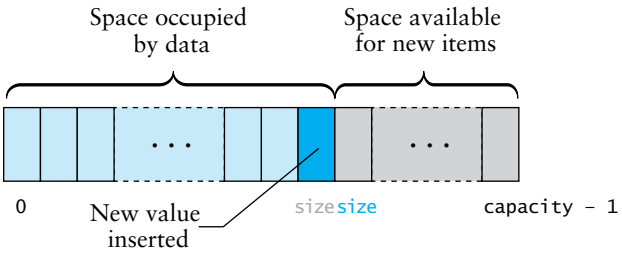


FIGURE 2.7 Adding an Element to the End of a `KWArrayList`

If the `size` is already equal to the `capacity`, we must first allocate a new array to hold the data and then copy the data to this new array. The method `reallocate` (explained shortly) does this. The code for the `add` method follows.

```
public boolean add(E anEntry) {
    if (size == capacity) {
        reallocate();
    }
    theData[size] = anEntry;
    size++;
    return true;
}
```

PROGRAM STYLE

Using the Postfix (or Prefix) Operator with a Subscript

Some programmers prefer to combine the two statements before `return` in the `add` method and write them as

```
theData[size++] = theValue;
```

This is perfectly valid. Java uses the current value of `size` as the array subscript and then increments it. The only difficulty is the fact that two operations are written in one statement and are carried out in a predetermined order (first access array and then increment subscript). If you write the prefix operator (`++size`) by mistake, the subscript will increase before array access.

The `add(int index, E anEntry)` Method

To insert an item into the middle of the array (anywhere but the end), the values that are at the insertion point and beyond must be shifted over to make room. In Figure 2.8, the arrow with label 1 shows the first element moved, the arrow with label 2 shows the next element moved, and so on. This data move is done using the following loop:

```
for (int i = size; i > index; i--) {
     theData[i] = theData[i - 1];
}
```

Note that the array subscript starts at `size` and moves toward the beginning of the array (down to `index + 1`). If we had started the subscript at `index + 1` instead, we would duplicate the value at `index` in each element of the array following it. Before we execute this loop, we need to be sure that `size` is not equal to `capacity`. If it is, we must call `reallocate`.

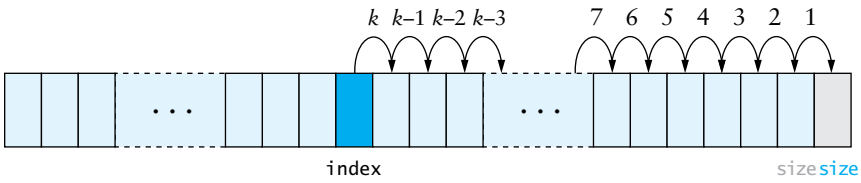


FIGURE 2.8
Making Room to Insert an Item into an Array

After increasing the capacity (if necessary) and moving the other elements, we can then add the new item. The complete code, including a test for an out-of-bounds value of `index`, follows:

```
public void add(int index, E anEntry) {
    if (index < 0 || index > size) {
        throw new ArrayIndexOutOfBoundsException(index);
    }
    if (size == capacity) {
        reallocate();
    }
    // Shift data in elements from index to size - 1
    for (int i = size; i > index; i--) {
        theData[i] = theData[i - 1];
    }
    // Insert the new item.
    theData[index] = anEntry;
    size++;
}
```

The set and get Methods

Methods `set` and `get` throw an exception if the array index is out of bounds; otherwise, method `get` returns the item at the specified index. Method `set` inserts the new item (parameter `newValue`) at the specified index and returns the value (`oldValue`) that was previously stored at that index.

```
public E get(int index) {
    if (index < 0 || index >= size) {
        throw new ArrayIndexOutOfBoundsException(index);
    }
    return theData[index];
}

public E set(int index, E newValue) {
    if (index < 0 || index >= size) {
        throw new ArrayIndexOutOfBoundsException(index);
    }
    E oldValue = theData[index];
    theData[index] = newValue;
    return oldValue;
}
```

The remove Method

To remove an item, the items that follow it must be moved forward to close up the space. In Figure 2.9, the arrow with label 1 shows the first element moved, the arrow with label 2 shows the next element moved, and so on. This data move is done using the `for` loop in method `remove` shown next. The item removed is returned as the method result.

```
public E remove(int index) {
    if (index < 0 || index >= size) {
        throw new ArrayIndexOutOfBoundsException(index);
    }
    E returnValue = theData[index];
    for (int i = index + 1; i < size; i++) {
        theData[i - 1] = theData[i];
    }
    size--;
    return returnValue;
}
```

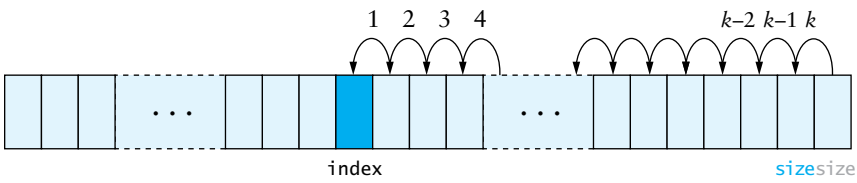


FIGURE 2.9
Removing an Item from an Array

The reallocate Method

The `reallocate` method creates a new array that is twice the size of the current array and then copies the contents of the current array into the new one. The `Arrays.copyOf` method makes a copy of the given array truncating if the new array is shorter or padding with nulls if the new array is larger, so that the copy has the specified length. The reference variable `theData` is then set to reference this new array. The code is as follows:

```
private void reallocate() {
    capacity = 2 * capacity;
    theData = Arrays.copyOf(theData, capacity);
}
```

The reason for doubling is to spread out the cost of copying. We discuss this next.

Performance of the KWArrayList Algorithms

The `set` and `get` methods are each a few lines of code and contain no loops. Thus, we say that these methods execute in constant time, or O(1).

If we insert into (or remove from) the middle of a `KWArrayList`, then at most n items have to be shifted. Therefore, the cost of inserting or removing an element is O(n). What if we must allocate a larger array before we can insert? Recall that when we reallocate the array, we double its size. Doubling an array of size n allows us to add n more items before we need to do another array copy. Therefore, we can add n new items after we have copied over n existing items. This averages out to 1 copy per add. Because each new array is twice the size of the previous one, it will take only about 20 `reallocate` operations to create an array that can store over a million references (2^{20} is greater than 1,000,000). Therefore, reallocation is effectively an O(1) operation, so the insertion is still O(n).

EXERCISES FOR SECTION 2.4

SELF-CHECK

1. Trace the execution of the following:

```
int[] anArray = {0, 1, 2, 3, 4, 5, 6, 7};
for (int i = 4; i < anArray.length - 1; i++)
    anArray[i + 1] = anArray[i];
```

and the following:

```
int[] anArray = {0, 1, 2, 3, 4, 5, 6, 7};
for (int i = anArray.length - 1; i > 4; i--)
    anArray[i] = anArray[i - 1];
```

What are the contents of `anArray` after the execution of each loop?

2. Write statements to remove the middle object from a `KWArrayList` and place it at the end.

PROGRAMMING

1. Implement the `indexOf` method of the `KWArrayList<E>` class.
2. Provide a constructor for class `KWArrayList<E>` that accepts an `int` argument that represents the initial array capacity. Use this instead of `INITIAL_CAPACITY`.

2.5 Single-Linked Lists

The `ArrayList` has the limitation that the `add` and `remove` methods operate in linear ($O(n)$) time because they require a loop to shift elements in the underlying array (see Figures 2.8 and 2.9). In this section, we introduce a data structure, the linked list, that overcomes this limitation by providing the ability to add or remove items anywhere in the list in constant ($O(1)$) time. A linked list is useful when you need to insert and remove elements at arbitrary locations (not just at the end) and when you do frequent insertions and removals.

One example would be maintaining an alphabetized list of students in a course at the beginning of a semester while students are adding and dropping courses. If you were using an `ArrayList`, you would have to shift all names that follow the new person's name down one position before you could insert a new student's name. Figure 2.10 shows this process. The names in gray were all shifted down when Barbara added the course. Similarly, if a student drops the course, the names of all students after the one who dropped (in gray in Figure 2.11) would be shifted up one position to close up the space.

Another example would be maintaining a list of students who are waiting to register for a course. Instead of having the students waiting in an actual line, you can give each student a number, which is the student's position in the line. If someone drops out of the line, everyone with a higher number gets a new number that is 1 lower than before. If someone cuts into the line because they "need the course to graduate," everyone after this person gets a new number, which is 1 higher than before. The person maintaining the list is responsible for giving everyone their new number after a change. Figure 2.12 illustrates what happens when Alice is inserted and given the number 1: Everyone whose number is ≥ 1 gets a new number. This process is analogous to maintaining the names in an `ArrayList`; each person's number is that person's position in the list, and some names in the list are shifted after every change.

FIGURE 2.10 Adding a Student to a Class List

FIGURE 2.11 Removing a Student from a Class List

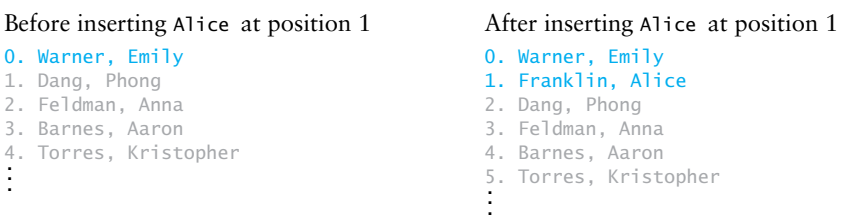


FIGURE 2.12 Inserting into a Numbered List of Students Waiting to Register

A better way to do this would be to give each person the name of the next person in line, instead of his or her own position in the line (which can change frequently). To start the registration process, the person who is registering students calls the person who is at the head of the line. After he or she finishes registration, the person at the head of the line calls the next person, and so on. Now what if person A lets person B cut into the line after her? Because B will now register after A, person A must call B. Also, person B must call the person who originally followed A. Figure 2.13 illustrates what needs to be done to insert Alice in the list after Emily. Only the two highlighted entries need to be changed (Emily must call Alice instead of Phong, and Alice must call Phong). Although Alice is shown at the bottom of Figure 2.13 (third column), she is really the second student in the list. The first four students in the list are Emily Warner, Alice Franklin, Phong Dang, and Anna Feldman.

What happens if someone drops out of our line? In this case, the name of the person who follows the one who drops out must be given to the person who comes before the one who drops out. This is illustrated in Figure 2.14. If Aaron drops out, only one entry needs to be changed (Anna must call Kristopher instead of Aaron).

Using a linked list is analogous to the process just discussed and illustrated in Figures 2.13 and 2.14 for storing our list of student names. After we find the position of a node to be inserted or removed, the actual insertion or removal is done in constant time and no shifts are required. Each element in a linked list, called a node, stores information and a link to the next node in the list. For example, for our list of students in Figure 2.14, the information

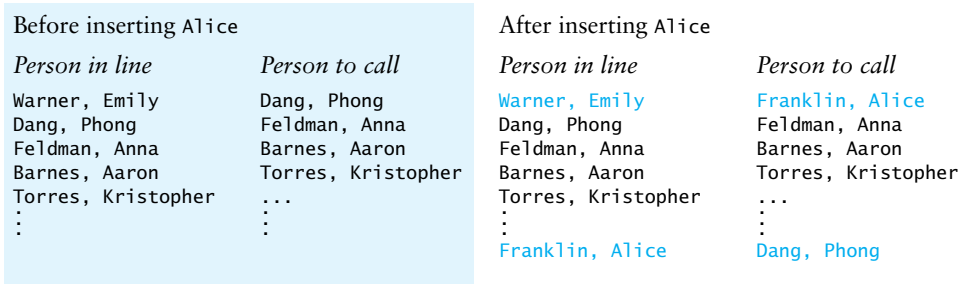


FIGURE 2.13 Inserting into a List Where Each Student Knows Who Is Next

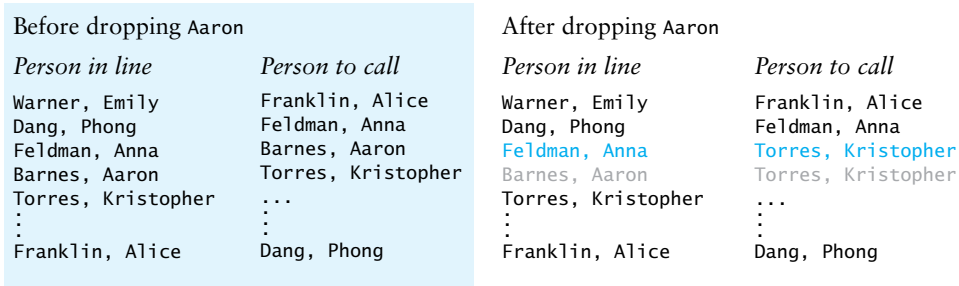


FIGURE 2.14 Removing a Student from a List Where Each Student Knows Who Is Next

"Warner, Emily" would be stored in the first node, and the link to the next node would reference a node whose information part was "Franklin, Alice". Here are the first three nodes of this list:

```
"Warner, Emily" ==> "Franklin, Alice" ==> "Dang, Phong"
```

We discuss how to represent and manipulate a linked list next.

A List Node

A *node* is a data structure that contains a data item and one or more links. A *link* is a reference to a node. A UML (unified modeling language) diagram of this relationship is shown in Figure 2.15. This shows that a Node contains a data field named data of type E and a reference (as indicated by the open diamond) to a Node. The name of the reference is next, as shown on the line from the Node to itself.

FIGURE 2.15
Node and Link

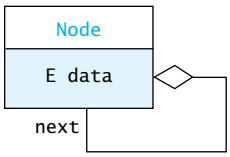


Figure 2.16 shows four nodes linked together to form the list "Tom" ==> "Dick" ==> "Harry" ==> "Sam". In this figure, we show that data references a String object. In subsequent figures, we will show the string value inside the Node. We will explain the purpose of the box in the left margin when we define class KWSingleLinkedList.

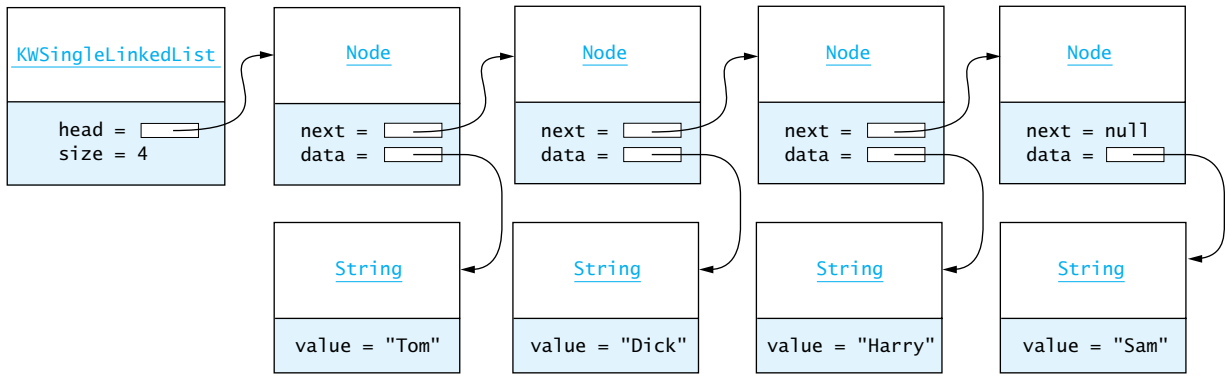
Next, we define a class Node<E> (see Listing 2.1) as an inner class that can be placed inside a generic list class. When each node is created, the type parameter E specifies the type of data stored in the node.

LISTING 2.1
An Inner Class Node

```
/** A Node is the building block for a single-linked list. */
private static class Node<E> {
    // Data Fields
    /** The reference to the data. */
    private E data;
    /** The reference to the next node. */
    private Node<E> next;

    // Constructors
    /** Creates a new node with a null next field.
        @param dataItem The data stored
     */
    private Node(E dataItem) {
        data = dataItem;
        next = null;
    }

    /** Creates a new node that references another node.
        @param dataItem The data stored
        @param nodeRef The node referenced by new node
     */
    private Node(E dataItem, Node<E> nodeRef) {
        data = dataItem;
        next = nodeRef;
    }
}
```

The keyword static in the class header indicates that the Node<E> class will not reference its outer class. (It can't because it has no methods other than constructors.) In the Java API documentation, static inner classes are also called *nested classes*.

FIGURE 2.16
Nodes in a Linked List



Generally, we want to keep the details of the `Node` class private. Thus, the qualifier `private` is applied to the class as well as to the data fields and the constructor. However, the data fields and methods of an inner class are visible anywhere within the enclosing class (also called the *parent class*).

The first constructor stores the data passed to it in the instance variable `data` of a new node. It also sets the `next` field to `null`. The second constructor sets the `next` field to reference the same node as its second argument. We didn't define a default constructor because none is needed.

Connecting Nodes

We can construct the list shown in Figure 2.16 using the following sequence of statements:

```
var tom = new Node<String>("Tom");
var dick = new Node<String>("Dick");
var harry = new Node<String>("Harry");
var sam = new Node<String>("Sam");
tom.next = dick;
dick.next = harry;
harry.next = sam;
```

The assignment statement

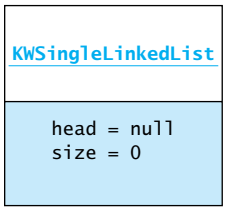
```
tom.next = dick;
```

stores a reference (link) to the node with data `"Dick"` in the variable `next` of node `tom`.

A Single-Linked List Class

Java does not have a class that implements single-linked lists. Instead, it has a more general double-linked list class, which will be discussed in the next section. However, we will create a `KWSingleLinkedList` class to show you how these operations could be implemented.

```
/** Class to represent a linked list with a link from each node to the next
    node. SingleLinkedList does not implement the List interface.
 */
public class KWSingleLinkedList<E> {
    /** Reference to list head. */
    private Node<E> head = null;
    /** The number of items in the list */
    private int size = 0;
    ...
```



A new `KWSingleLinkedList` object has two data fields, `head` and `size`, with initial values null and 0, respectively, as shown in the diagram in the margin. The data field `head` will reference the first list node called the *list head*. Method `addFirst` below inserts one element at a time to the front of the list, thereby changing the node pointed to by `head`. In the call to the constructor for `Node`, the argument `head` references the current first list node. A new node is created, which is referenced by `head` and is linked to the previous list `head`. Variable `data` of the new list head references `item`.

```
/** Add an item to the front of the list.
    @param item The item to be added
 */
public void addFirst(E item) {
    head = new Node<>(item, head);
    size++;
}
```

The following fragment creates a linked list names and builds the list shown in Figure 2.16 using method `addFirst`:

```
KWSingleLinkedList<String> names = new KWSingleLinkedList<>();
names.addFirst("Sam");
names.addFirst("Harry");
names.addFirst("Dick");
names.addFirst("Tom");
```

Inserting a Node in a List

If we have a reference harry to node `"Harry"`, we can insert a new node, `"Bob"`, into the list after `"Harry"` as follows:

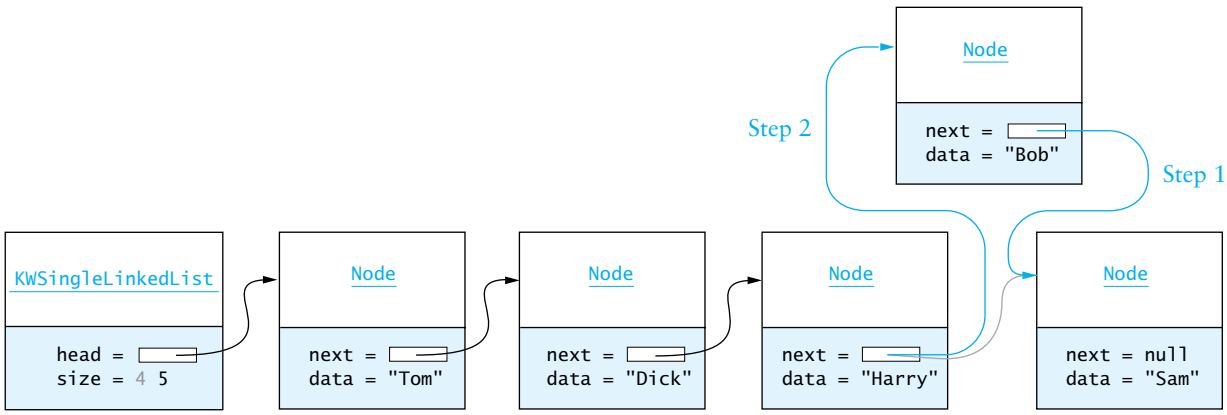
```
var bob = new Node<String>("Bob");
bob.next = harry.next; // Step 1
harry.next = bob; // Step 2
```

The linked list now is as shown in Figure 2.17. We show the number of the step that created each link alongside it. In Figures 2.17 and 2.18 the gray arrows show old links and the blue arrows show new links.

We can generalize this by writing the method `addAfter` as follows:

```
/** Add a node after a given node
    @param node The node preceding the new item
    @param item The item to insert
 */
```

FIGURE 2.17
After Inserting `"Bob"`



```
private void addAfter(Node<E> node, E item) {
    node.next = new Node<>(item, node.next);
    size++;
}
```

We declare this method `private` since it should not be called from outside the class. This is because we want to keep the internal structure of the class hidden. Such private methods are known as helper methods because they will help implement the public methods of the class. Later we will see how `addAfter` is used to implement the public add methods.

Removing a Node

If we have a reference, `tom`, to the node that contains `"Tom"`, we can remove the node that follows `"Tom"`:

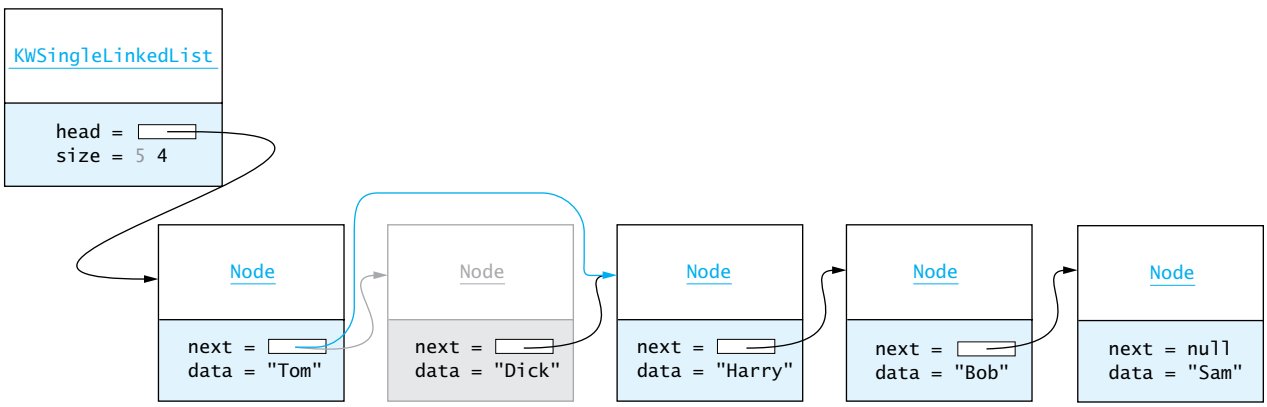
```
tom.next = tom.next.next;
```

The list is now as shown in Figure 2.18. Note that we did not start with a reference to `"Dick"` but instead began with a reference to `"Tom"`. To delete a node, we need a reference to the node that precedes it, not the node being deleted. (Recall from our registration list example that the person in front of the one dropping out of line must be told to call the person who follows the one who is dropping out.)

Again, we can generalize this by writing the `removeAfter` method:

```
/** Remove the node after a given node
    @param node The node before the one to be removed
    @return The data from the removed node, or null
            if there is no node to remove
 */
private E removeAfter(Node<E> node) {
    Node<E> temp = node.next;
    if (temp != null) {
        node.next = temp.next;
        size--;
        return temp.data;
    } else {
       return null;
    }
}
```

FIGURE 2.18
After Removing `"Dick"`



The `removeAfter` method works on all nodes except for the first one. For that, we need a special method, `removeFirst`:

```
/** Remove the first node from the list
    @return The removed node's data or null if the list is empty
 */
private E removeFirst() {
    Node<E> temp = head;
    if (head != null) {
        head = head.next;
    }
    // Return data at old head or null if list is empty
    if (temp != null) {
        size--;
        return temp.data;
    } else {
        return null;
    }
}
```

Completing the `KWSingleLinkedList` Class

We conclude our illustration of the single-linked list data structure by showing how it can be used to implement a limited subset of the methods required by the `List` interface (see Table 2.5). Specifically, we will write the `get`, `set`, and `add` methods. Methods `size`, `indexOf`, and `remove` are left as exercises. Recall from the `ArrayList` class that each of these methods takes an index parameter, but we showed above that the methods to add and remove a node need a reference to a node. We need an additional helper method to get a node at a given index.

```
/** Find the node at a specified position
    @param index The position of the node sought
    @return The node at index or null if it does not exist
 */
private Node<E> getNode(int index) {
    Node<E> node = head;
    for (int i = 0; i < index && node != null; i++) {
        node = node.next;
    }
    return node;
}
```

TABLE 2.5
Selected Methods of Interface `java.util.List<E>`

Method	Behavior
`E get (int index)`	Returns the data in the element at position `index`
`E set(int index, E anEntry)`	Stores a reference to `anEntry` in the element at position `index`. Returns the data formerly at position `index`
`int size()`	Gets the current size of the `List`
`boolean add(E anEntry)`	Adds a reference to `anEntry` at the end of the `List`. Always returns `true`
`void add (int index, E anEntry)`	Adds a reference to `anEntry`, inserting it before the item at position `index`
`int indexOf(E target)`	Searches for `target` and returns the position of the first occurrence, or –1 if it is not in the `List`
`E remove (int index)`	Removes the entry formerly at position `index` and returns it

The get and set Methods

Using the getNode method, the get and set methods are straightforward:

```
/** Get the data at index
    @param index The position of the data to return
    @return The data at index
    @throws IndexOutOfBoundsException if index is out of range
 */
public E get(int index) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException(Integer.toString(index));
    }
    Node<E> node = getNode(index);
    return node.data;
}

/** Store a reference to anEntry in the element at position index.
    @param index The position of the item to change
    @param newValue The new data
    @return The data previously at index
    @throws IndexOutOfBoundsException if index is out of range
 */
public E set(int index, E newValue) {
    if (index < 0 || index >= size) {
        throw new IndexOutOfBoundsException(Integer.toString(index));
    }
    Node<E> node = getNode(index);
    E result = node.data;
    node.data = newValue;
    return result;
}
```

The add Methods

After verifying that the index is in range, the index is checked for the special case of adding at the first element. If the index is zero, then the addFirst method is used to insert the new item; otherwise the addAfter method is used. Note that getNode (called before addAfter) returns a reference to the predecessor of the node to be inserted.

```
/** Insert the specified item at index
    @param index The position where item is to be inserted
    @param item The item to be inserted
    @throws IndexOutOfBoundsException if index is out of range
 */
public void add(int index, E item) {
    if (index < 0 || index > size) {
        throw new IndexOutOfBoundsException(Integer.toString(index));
    }

    if (index == 0) {
        addFirst(item);
    } else {
        Node<E> node = getNode(index-1);
        addAfter(node, item);
    }
}
```

The List interface also specifies an add method without an index that adds (appends) an item to the end of a list. It can be easily implemented by calling the add(int index, E item) method using size as the index.

```
/** Append item to the end of the list
    @param item The item to be appended
    @return true (as specified by the Collection interface)
 */
public boolean add(E item) {
    add(size, item);
    return true;
}
```

EXERCISES FOR SECTION 2.5

SELF-CHECK

1. What is the big-O for the single-linked list `get` operation?
2. What is the big-O for the `set` operation?
3. What is the big-O for each `add` method?
4. Draw a single-linked list of `Integer` objects containing the integers 5, 10, 7, and 30 and referenced by `head`. Complete the following fragment, which adds all `Integer` objects in a list. Your fragment should walk down the list, adding all integer values to `sum`.

```
int sum = 0;
Node<Integer> nodeRef = ____________________;
while (nodeRef != null) {
    int next = ____________________;
    sum += next;
    nodeRef = ____________________;
}
```

5. For the single-linked list in Figure 2.16, data field `head` (type `Node<string>`) references the first node. Explain the effect of each statement in the following fragments.
   a. `head = new Node<>("Shakira", head.next);`
   b.
   ```
   Node<String> nodeRef = head.next;
   nodeRef.next = nodeRef.next.next;
   ```
   c.
   ```
   Node<String> nodeRef = head;
   while (nodeRef.next != null)
       nodeRef = nodeRef.next;
       nodeRef.next = new Node<>("Tamika");
   ```
   d.
   ```
   Node<String> nodeRef = head;
   while (nodeRef != null && !nodeRef.data.equals("Harry"))
       nodeRef = nodeRef.next;
   if (nodeRef != null) {
       nodeRef.data = "Sally";
       nodeRef.next = new Node<>("Harry", nodeRef.next.next);
   }
   ```

PROGRAMMING

1. Using the single-linked list shown in Figure 2.16, and assuming that `head` references the first `Node` and `tail` references the last `Node`, write statements to do each of the following:
   a. Insert `"Bill"` before `"Tom"`.
   b. Insert `"Sue"` before `"Sam"`.
   c. Remove `"Bill"`.
   d. Remove `"Sam"`.
2. Write method `size`.

3. Write method `indexOf`.
4. Write method `remove`. Use helper methods `getNode`, `removeFirst`, and `removeAfter`. Method `remove` should throw and exception if `index` is out-of-bounds.
5. Write the remove method whose method heading follows.

```
/** Remove the first occurrence of element item.
    @param item The item to be removed
    @return true if item is found and removed; otherwise, return false.
 */
public boolean remove(E item)
```

6. Write the following method `add` for class `KWSingleLinkedList<E>` without using any helper methods.

```
/** Insert a new item before the one at position index,
    starting at 0 for the list head. The new item is inserted
    between the one at position index-1 and the one formerly at position index.
    @param index The index where the new item is to be inserted
    @param item The item to be inserted
    @throws IndexOutOfBoundsException if the index is out of range
 */
public void add(int index, E item)
```

2.6 Double-Linked Lists and Circular Lists

Our single-linked list data structure has some limitations:

- Insertion at the front of the list is O(1). Insertion at other positions is O(n), where n is the size of the list.
- We can insert a node only after a node for which we have a reference. For example, to insert `"Bob"` in Figure 2.17, we needed a reference to the node containing `"Harry"`. If we wanted to insert `"Bob"` before `"Sam"` but did not have a reference to `"Harry"`, we would have to start at the beginning of the list and search until we found a node whose next node was `"Sam"`.
- We can remove a node only if we have a reference to its predecessor node. For example, to remove `"Dick"` in Figure 2.18, we needed a reference to the node containing `"Tom"`. If we wanted to remove `"Dick"` without having this reference, we would have to start at the beginning of the list and search until we found a node whose next node was `"Dick"`.
- We can move in only one direction, starting at the list head, whereas with an `ArrayList` we can move forward (or backward) by incrementing (or decrementing) the index.

We can overcome these limitations by adding a reference to the previous `node` in the `Node` class, as shown in the UML class diagram in Figure 2.19. The open diamond indicates that both `prev` and `next` are references whose values can be changed. Our *double-linked list* is shown in Figure 2.20.

FIGURE 2.19
Double-Linked List Node

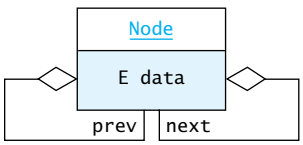
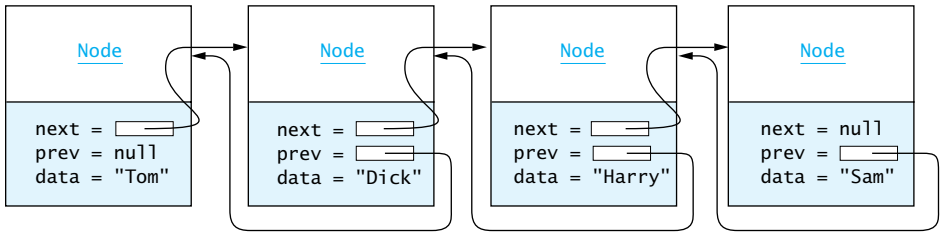


FIGURE 2.20
A Double-Linked List



PITFALL

Falling Off the End of a List

If `nodeRef` is at the last list element and you execute the statement

```
nodeRef = nodeRef.next;
```

`nodeRef` will be set to `null`, and you will fall off the end of the list. This is not an error. However, if you execute this statement again, you will get a `NullPointerException`, because `nodeRef.next` is undefined when `nodeRef` is `null`.

The Node Class

The `Node` class for a double-linked list has references to the data and to the next and previous nodes. The declaration of this class follows.

```
/** A Node is the building block for a double-linked list. */
private static class Node<E> {
    /** The data value. */
    private E data;
    /** The link to the next node. */
    private Node<E> next = null;
    /** The link to the previous node. */
    private Node<E> prev = null;

    /** Construct a node with the given data value.
        @param dataItem The data value
     */
    private Node(E dataItem) {
        data = dataItem;
    }
}
```

Inserting into a Double-Linked List

If `sam` is a reference to the node containing `"Sam"`, we can insert a new node containing `"Sharon"` into the list before `"Sam"` using the following statements. Before the insertion, we can refer to the predecessor of `sam` as `sam.prev`. After the insertion, this node will be referenced by `sharon.prev`.

```
Node<String> sharon = new Node<>("Sharon");
// Link new node to its neighbors.
    sharon.next = sam; // Step 1
    sharon.prev = sam.prev; // Step 2
```

```
// Link old predecessor of sam to new predecessor.
    sam.prev.next = sharon; // Step 3
// Link to new predecessor.
    sam.prev = sharon; // Step 4
```

The three nodes affected by the insertion are shown in Figures 2.21 and 2.22. The old links are shown in black or gray if they were changed, and the new links are shown in blue. Next to each blue link we show the number of the step that creates it. Figure 2.21 shows the links after Steps 1 and 2, and Figure 2.22 shows the links after Steps 3 and 4.

Removing from a Double-Linked List

If we have a reference, `harry`, to the node that contains `"Harry"`, we can remove that node without having a named reference to its predecessor:

```
harry.prev.next = harry.next; // Step 1
harry.next.prev = harry.prev; // Step 2
```

The list is now as shown in Figure 2.23.

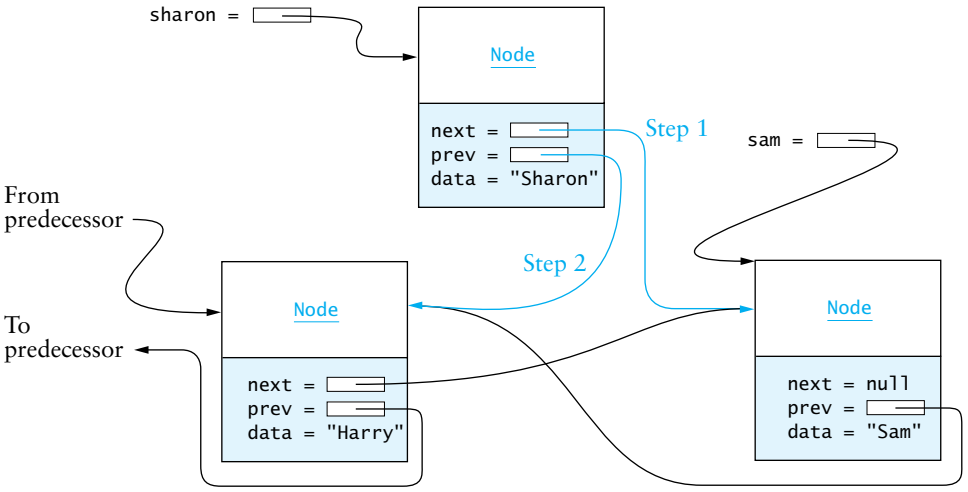


FIGURE 2.21 Steps 1 and 2 in Inserting `"Sharon"`

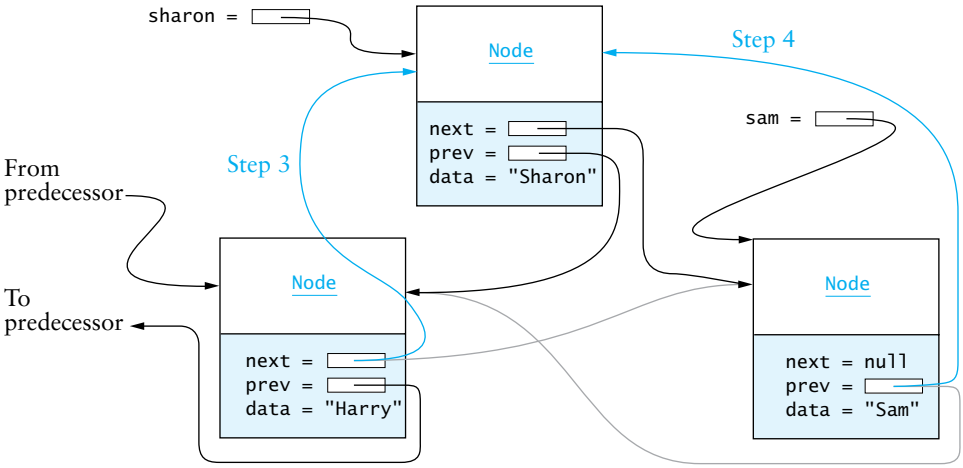


FIGURE 2.22 After Inserting `"Sharon"` before `"Sam"`

FIGURE 2.23
Removing `"Harry"` from a Double-Linked List

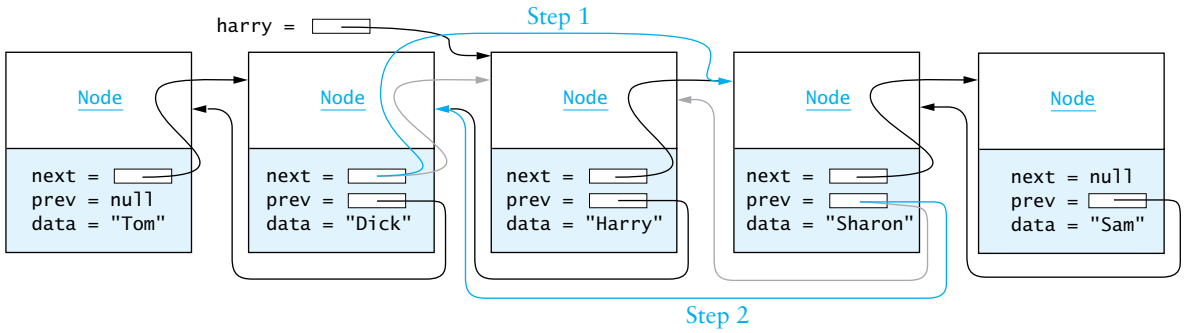
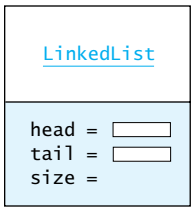


FIGURE 2.24
A Double-Linked List Object



A Double-Linked List Class

So far we have shown just the internal `Nodes` for a linked list. A double-linked list object would consist of a separate object with data fields `head` (a reference to the first list `Node`), `tail` (a reference to the last list `Node`), and `size` (the number of `Nodes`) (see Figure 2.24). Because both ends of the list are directly accessible, now insertion at either end is O(1); insertion elsewhere is still O(n).

Circular Lists

We can create a *circular* list from a double-linked list by linking the last node to the first node (and the first node to the last one). If `head` references the first list node and `tail` references the last list node, the statements

```
head.prev = tail;
tail.next = head;
```

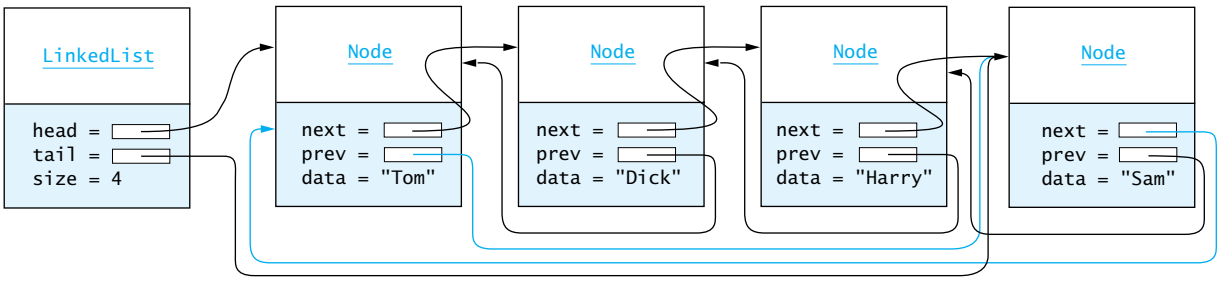
would accomplish this (see Figure 2.25).

You could also create a circular list from a single-linked list by executing just the statement

```
tail.next = head;
```

This statement connects the last list element to the first list element. If you keep a reference to only the last list element, `tail`, you can access the last element and the first element (`tail.next`) in O(1)time.

FIGURE 2.25
A Circular Linked List



One advantage of a circular list is that you can continue to traverse in the forward (or reverse) direction even after you have passed the last (or first) list node. This enables you to visit all the list elements from any starting point. In a list that is not circular, you would have to start at the beginning or at the end if you wanted to visit all the list elements. A second advantage of a circular list is that you can never fall off the end of the list. There is a disadvantage: You must be careful not to set up an infinite loop.

EXERCISES FOR SECTION 2.6

SELF-CHECK

1. Answer the following questions about lists.
   a. Each node in a single-linked list has a reference to _____ and _____.
   b. In a double-linked list, each node has a reference to _____, _____, and _____.
   c. To remove an item from a single-linked list, you need a reference to _____.
   d. To remove an item from a double-linked list, you need a reference to _____.
2. For the double-linked list in Figure 2.20, explain the effect of each statement in the following fragments.
   a.
```
Node<String> nodeRef = tail.prev;
nodeRef.prev.next = tail;
tail.prev = nodeRef.prev;
```
   b.
```
Node<String> nodeRef = head;
head = new Node<>("Tamika");
head.next = nodeRef;
nodeRef.prev = head;
```
   c.
```
Node<String> nodeRef = new Node<>("Shakira");
nodeRef.prev = head;
nodeRef.next = head.next;
head.next.prev = nodeRef;
head.next = nodeRef;
```

PROGRAMMING

1. For the double-linked list shown in Figure 2.20, assume `head` references the first list node and `tail` references the last list node. Write statements to do each of the following:
   a. Insert `"Bill"` before `"Tom"`.
   b. Insert `"Sue"` before `"Sam"`.
   c. Remove `"Bill"`.
   d. Remove `"Sam"`.

2.7 The LinkedList Class and the Iterator, ListIterator, and Iterable Interfaces

The LinkedList Class

The `LinkedList` class, part of the Java API package `java.util`, is a double-linked list that implements the `List` interface. A selected subset of the methods from this Java API is shown

TABLE 2.6
Selected Methods of the `java.util.LinkedList<E>` Class

Method	Behavior
`public void add(int index, E obj)`	Inserts object `obj` into the list at position `index`
`public void addFirst(E obj)`	Inserts object `obj` as the first element of the list
`public void addLast(E obj)`	Adds object `obj` to the end of the list
`public E get(int index)`	Returns the item at position `index`
`public E getFirst()`	Gets the first element in the list. Throws `NoSuchElementException` if the list is empty
`public E getLast()`	Gets the last element in the list. Throws `NoSuchElementException` if the list is empty
`public boolean remove(E obj)`	Removes the first occurrence of object `obj` from the list. Returns true if the list contained object `obj`; otherwise, returns `false`
`public int size()`	Returns the number of objects contained in the list

in Table 2.6. Because the `LinkedList` class, like the `ArrayList` class, implements the `List` interface, it contains many of the methods found in the `ArrayList` class as well as some additional methods.

The `Iterator`

Let's say we want to process each element in a `LinkedList`. We can use the following loop to access the list elements in sequence, starting with the one at index 0.

```
// Access each list element.
for (int index = 0; index < aList.size(); index++) {
    E nextElement = aList.get(index);
    // Do something with the element at position index (nextElement)
    ...
}
```

The loop is executed `aList.size()` times; thus, it is O(n). During each iteration, we call the method `get` to retrieve the element at position `index`.

If we assume that the method `get` begins at the first list node (`head`), each call to method `get` must advance a local reference (`nodeRef`) to the node at position `index` using a loop such as:

```
// Advance nodeRef to the element at position index.
Node<E> nodeRef = head;
for (int j = 0; j < index; j++) {
    nodeRef = nodeRef.next;
}
```

This loop (in method `get`) executes `index` times, so it is also O(n). Therefore, the performance of the nested loops used to process each element in a `LinkedList` is O(n^2) and is very inefficient. We would like to have an alternative way to access the elements in a linked list sequentially.

We can use the concept of an iterator to accomplish this. Think of an *iterator* as a moving place marker that keeps track of the current position in a particular linked list. The

Iterator object for a list starts at the first element in the list. The programmer can use the Iterator object's next method to retrieve the next element. Each time it does a retrieval, the Iterator object advances to the next list element, where it waits until it is needed again. We can also ask the Iterator object to determine whether the list has more elements left to process (method hasNext). Iterator objects throw a NoSuchElementException if they are asked to retrieve the next element after all elements have been processed.

EXAMPLE 2.10 Assume iter is declared as an Iterator object for LinkedList myList. We can replace the fragment shown at the beginning of this section with the following:

```
// Access each list element.
while (iter.hasNext()) {
    E nextElement = iter.next();
    // Do something with the next element (nextElement).
    ...
}
```

This fragment is O(n) instead of O(n^2). All that remains is to determine how to declare iter as an Iterator for LinkedList object myList. We show how to do this in the next section and discuss Iterator a bit more formally.

The Iterator Interface

The interface Iterator<E> is defined as part of API package java.util. Table 2.7 summarizes the methods declared by this interface.

The List interface declares the method iterator, which returns an Iterator object that will iterate over the elements of that list. (The requirement for the iterator method is actually in the Iterable interface, which is the superinterface for the Collection interface, which is a superinterface for the List interface. We discuss the Collection interface in Section 2.10.)

In the following loop, we process all items in List<Integer> aList through an Iterator.

```
// Obtain an Iterator to the list aList.
Iterator<Integer> itr = aList.iterator();
while (itr.hasNext()) {
    int value = itr.next());
    // Do something with value.
    ...
}
```

An Iterator does not refer to or point to a particular object at any given time. Rather, you should think of an Iterator as pointing between objects within a list. The method

TABLE 2.7
The java.util.Iterator<E> Interface

Method	Behavior
boolean hasNext()	Returns true if the next method returns a value
E next()	Returns the next element. If there are no more elements, throws the NoSuchElementException
void remove()	Removes the last element returned by the next method

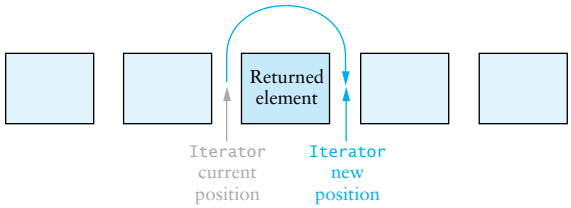


FIGURE 2.26 Advancing an `Iterator` via the `next` Method

`hasNext` tells us whether or not calling the `next` method will succeed. If `hasNext` returns **`true`**, then a call to `next` will return the next object in the list and advance the `Iterator` (see Figure 2.26). If `hasNext` returns **`false`**, a call to `next` will cause the `NoSuchElementException` to be thrown.

You can use the `Iterator remove` method to remove elements from a list as you access them. You can remove only the element that was most recently accessed by **`next`**. Each call to `remove` must be preceded by a call to **`next`** to retrieve the next element.

EXAMPLE 2.11 We wish to remove all elements from `aList` (type `LinkedList<Integer>`) that are divisible by a particular value. The following method will accomplish this:

```
/** Remove the items divisible by div. */
    @pre LinkedList aList contains Integer objects.
    @post Elements divisible by div have been removed.
 */
public static void removeDivisibleBy(LinkedList<Integer> aList, int div) {
    Iterator<Integer> iter = aList.iterator();
    while (iter.hasNext()) {
        int nextInt = iter.next();
        if (nextInt % div == 0)
            iter.remove();
    }
}
```

The method call `iter.next` retrieves the next `Integer` in the list. Its value is unboxed, and if it is divisible by `div`, the statement

```
iter.remove();
```

removes the element just retrieved from the list.

PITFALL

Improper Use of remove

If a call to `remove` is not preceded by a call to `next`, `remove` will throw an `IllegalStateException`. If you want to remove two consecutive elements in a list, a separate call to `next` must occur before each call to `remove`.

PROGRAM STYLE

Removal Using `Iterator.remove` versus `List.remove`

You could also use method `LinkedList.remove` to remove elements from a list. However, it is more efficient to remove multiple elements from a list using `Iterator.remove` than it would be to use `LinkedList.remove`. The `LinkedList.remove` method removes only one element at a time, so you would need to start at the beginning of the list each time and advance down the list to each element that you wanted to remove (**O**(n^2) process). With the `Iterator.remove` method, you can remove elements as they are accessed by the `Iterator` object without having to go back to the beginning of the list (**O**(n) process).

The Enhanced for Loop

The enhanced **`for`** loop (also called the **`for`** each loop) makes it easier to sequence through arrays. It also enables sequential access to `List` objects without the need to create and manipulate an iterator. If `myList` is a `List` of strings, a **`for`** each loop that begins with the following line will store each of its strings in variable `nextStr`, starting with the first one.

```
for (String nextStr : myList) {
```

The enhanced **`for`** loop creates an `Iterator` object and implicitly calls its `hasNext` and `next` methods. This `Iterator` object is not accessible within the body of the loop, so other `Iterator` methods, such as `remove`, are not available. The enhanced **`for`** loop can be used with arrays and any class that implements the `Iterable` interface (see Section 2.10).

The `ListIterator` Interface

The `Iterator` has some limitations. It can traverse the `List` only in the forward direction. It also provides only a `remove` method, not an `add` method. Also, to start an `Iterator` somewhere other than at the first `List` element, you must write your own loop to advance the `Iterator` to the desired starting position.

The Java API also contains the `ListIterator<E>` interface, which is an extension of the `Iterator<E>` interface that overcomes these limitations. Like the `Iterator`, the `ListIterator` should be thought of as being positioned between elements of the linked list. The positions are assigned an index from 0 to `size`, where the position just before the first element has index 0 and the position just after the last element has index `size`. The `next` method moves the iterator forward and returns the element that was jumped over. The `previous` method moves the iterator backward and also returns the element that was jumped over. This is illustrated in Figure 2.27, where *i* is the current position of the iterator. The methods defined by the `ListIterator` interface are shown in Table 2.8.

FIGURE 2.27 Moving a `ListIterator` via `next` or `previous`

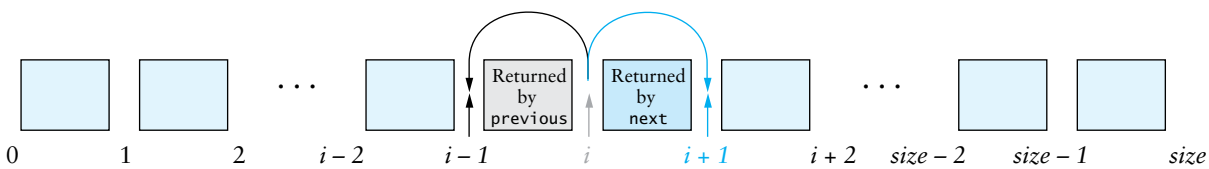


TABLE 2.8
The `java.util.ListIterator<E>` Interface

Method	Behavior
`void add(E obj)`	Inserts object `obj` into the list just before the item that would be returned by the next call to method `next` and after the item that would have been returned by method previous. If the method `previous` is called after `add`, the newly inserted object will be returned
`boolean hasNext()`	Returns `true` if `next` will not throw an exception
`boolean hasPrevious()`	Returns `true` if `previous` will not throw an exception
`E next()`	Returns the next object and moves the iterator forward. If the iterator is at the end, the `NoSuchElementException` is thrown
`int nextIndex()`	Returns the index of the item that will be returned by the next call to `next`. If the iterator is at the end, the list size is returned
`E previous()`	Returns the previous object and moves the iterator backward. If the iterator is at the beginning of the list, the `NoSuchElementException` is thrown
`int previousIndex()`	Returns the index of the item that will be returned by the next call to `previous`. If the iterator is at the beginning of the list, `-1` is returned
`void remove()`	Removes the last item returned from a call to `next` or `previous`. If a call to `remove` is not preceded by a call to `next` or `previous`, the `IllegalStateException` is thrown
`void set(E obj)`	Replaces the last item returned from a call to `next` or `previous` with `obj`. If a call to `set` is not preceded by a call to `next` or `previous`, the `IllegalStateException` is thrown

TABLE 2.9
Methods in `java.util.LinkedList<E>` That Return `ListIterators`

Method	Behavior
`ListIterator<E> listIterator()`	Returns a `ListIterator` that begins just before the first list element
`ListIterator<E> listIterator(int index)`	Returns a `ListIterator` that begins just before the position `index`

To obtain a `ListIterator`, you call the `listIterator` method of the `LinkedList` class. This method has two forms, as shown in Table 2.9.

EXAMPLE 2.12 If `myList` is type `LinkedList<String>`, the statement

```
ListIterator<String> myIter = myList.listIterator(3);
```

would create a `ListIterator` object `myIter` positioned between the elements at positions 2 and 3 of the linked list. The method call

```
myIter.next()
```

would return a reference to the `String` object at position 3 and move the iterator forward; the method call

```
myIter.nextIndex()
```

would return 4. The method call

```
myIter.previous()
```

would return a reference to the `String` object at position 3 and move the iterator back to its original position. The method call

```
myIter.previousIndex()
```

would return 2. The method call

```
myIter.hasNext()
```

would return `true` if the list has at least four elements; the method call

```
myIter.hasPrevious()
```

would return `true`.

EXAMPLE 2.13 The fragment

```
ListIterator<String> myIter = myList.listIterator();
while (myIter.hasNext()) {
    if (target.equals(myIter.next())) {
        myIter.set(newItem);
        break; // Exit loop
    }
}
```

searches for `target` in list `myList` (type `List<String>`) and, if `target` is present, replaces its first occurrence with `newItem`.

Comparison of `Iterator` and `ListIterator`

Because the interface `ListIterator<E>` is a subinterface of `Iterator<E>`, classes that implement `ListIterator` must provide all of the capabilities of both. The `Iterator` interface requires fewer methods and can be used to iterate over more general data structures—structures for which an index is not meaningful and ones for which traversing in only the forward direction is required. It is for this reason that the `Iterator` is required by the `Collection` interface (more general), whereas the `ListIterator` is required only by the `List` interface (more specialized). We will discuss the `Collection` interface in Section 2.10.

Conversion between a `ListIterator` and an Index

The `ListIterator` has the methods `nextIndex` and `previousIndex`, which return the index values associated with the items that would be returned by a call to the `next` or `previous` methods. The `LinkedList` class has the method `listIterator(int index)`, which returns a `ListIterator` whose next call to `next` will return the item at position `index`. Thus, you can convert between an index and a `ListIterator`. However, remember that the `listIterator(int index)` method returns the desired `ListIterator` by creating a new `ListIterator` that starts at the beginning and then walks along the list until the desired position is found. There is a special case where `index` is equal to `size()`, but all others are an O(n) operation.

The `Iterable` Interface

Next, we show the `Iterable` interface. This interface requires only that a class that implements it provides an `iterator` method. As mentioned above, the `Collection` interface extends the `Iterable` interface, so all classes that implement the `List` interface (a subinterface of `Collection`) must provide an `iterator` method.

```
public interface Iterable<E> {
    /** Returns an iterator over the elements in this collection. */
    Iterator<E> iterator();
}
```

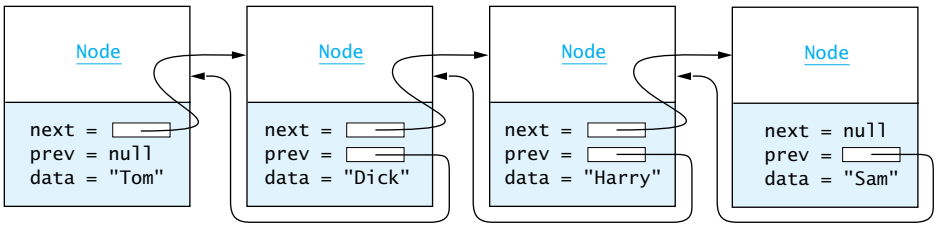
EXERCISES FOR SECTION 2.7

SELF-CHECK

1. The method `indexOf`, part of the `List` interface, returns the index of the first occurrence of an object in a `List`. What does the following code fragment do?

```
int indexOfSam = myList.indexOf("Sam");
ListIterator<String> iteratorToSam = myList.listIterator(indexOfSam);
iteratorToSam.previous();
iteratorToSam.remove();
```

   where the internal nodes of `myList` (type `LinkedList<String>`) are shown in the figure below:



2. In Question 1, what if we change the statement

```
iteratorToSam.previous();
```

   to

```
iteratorToSam.next();
```

3. In Question 1, what if we omit the statement

```
iteratorToSam.previous();
```

PROGRAMMING

1. Write the method `indexOf` as specified in the `List` interface by adapting the code shown in Example 2.13 to return the index of the first occurrence of an object.
2. Write the method `lastIndexOf` specified in the `List` interface by adapting the code shown in Example 2.13 to return the index of the last occurrence of an object.
3. Write a method `indexOfMin` that returns the index of the minimum item in a `List`, assuming that each item in the list implements the `Comparable` interface.

2.8 Application of the `LinkedList` Class

In this section, we introduce a case study that uses the Java `LinkedList` class to solve a common problem: maintaining an ordered list. We will develop an `OrderedList` class.

CASE STUDY Maintaining an Ordered List

Problem As discussed in Section 2.5, we can use a linked list to maintain a list of students who are registered for a course. We want to maintain this list so that it will be in alphabetical order even after students have added and dropped the course.

Analysis Instead of solving this problem just for a list of students, we will develop a generic `OrderedList` class that can be used to store any group of objects that can be compared. Java classes whose object types can be compared implement the `Comparable` interface, which is defined as follows:

```
/** Instances of classes that realize this interface can be compared.
 */
public interface Comparable<E> {
    /** Method to compare this object to the argument object.
        @param obj The argument object
        @return Returns a negative integer if this object < obj;
                zero if this object equals obj;
                a positive integer if this object > obj
     */
    int compareTo(E obj);
}
```

Therefore, a class that implements the `Comparable` interface must provide a `compareTo` method that returns an `int` value that indicates the relative ordering of two instances of that class. The result is negative if this object < argument; zero if this object equals argument; and positive if this object > argument.

We can either extend the Java `LinkedList` class to create a new class `OrderedList`, or create an `OrderedList` class that uses a `LinkedList` to store the items. If we implement our `OrderedList` class as an extension of `LinkedList`, a client will be able to use methods in the `List` interface that can insert new elements or modify existing elements in such a way that the items are no longer in order. Therefore, we will use the `LinkedList` class as a component of the `OrderedList` class and we will implement only those methods that preserve the order of the items.

Design The class diagram in Figure 2.28 shows the relationships among the `OrderedList` class, the `LinkedList` class, and the `Comparable` interface. The filled diamond indicates that the `LinkedList` is a component of the `OrderedList`, and the open diamond indicates that the `LinkedList` will contain `Comparable` objects. We explain the meaning of the text `E extends Comparable<E>` shortly.

Because we want to be able to make insertions and deletions in the ordered linked list, we must implement `add` and `remove` methods. We also provide a `get` method, to access

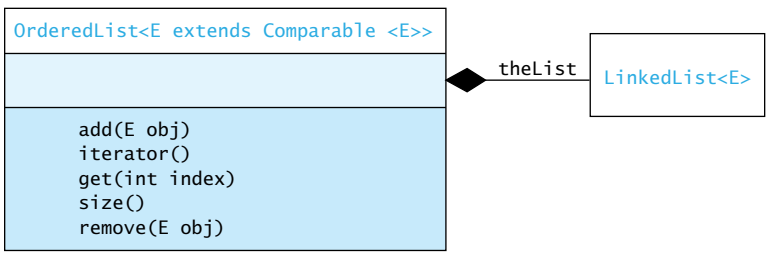


FIGURE 2.28 `OrderedList` Class Diagram

TABLE 2.10
Class `OrderedList<E extends Comparable<E>>`

Data Field	Attribute
`private LinkedList<E> theList`	A linked list to contain the data
Method	**Behavior**
`public void add(E obj)`	Inserts `obj` into the list preserving the list's order
`public Iterator iterator()`	Returns an `Iterator` to the list
`public E get(int index)`	Returns the object at the specified position
`public int size()`	Returns the size of the list
`public boolean remove(E obj)`	Removes first occurrence of `obj` from the list. Returns `true` if the list contained object `obj`; otherwise, returns `false`

the element at a particular position, and an `iterator` method, to provide the user with the ability to access all of the elements in sequence efficiently. Table 2.10 describes the class. Although not shown in Figure 2.28, class `OrderedList<E>` implements `Iterable<E>` because it has an `iterator` method. The following one is the start of its definition.

```
import java.util.*;

/** A class to represent an ordered list. The data is stored in
    a linked list data field.
 */

public class OrderedList<E extends Comparable<E>>
             implements Iterable<E> {
    /** A list to contain the data. */
    private List<E> theList = new LinkedList<>();
```

Because we want our ordered list to contain only objects that implement the `Comparable` interface, we need to tell the compiler that only classes that meet this criterion should be bound to the type parameter `E`. We do this by declaring our ordered list as `OrderedList<E extends Comparable<E>>`.

SYNTAX Specifying Requirements on Generic Types

FORM:
`class` *`ClassName<TypeParameter`* `extends` *`ClassOrInterfaceName<TypeParameter>>`*

EXAMPLE:
```
class OrderedList<E extends Comparable<E>>
```

MEANING:
When we declare actual objects of type *`ClassName<TypeParameter>`*, class *`TypeParameter`* must extend class *`ClassOrInterfaceName`* or implement interface *`ClassOrInterfaceName`*.

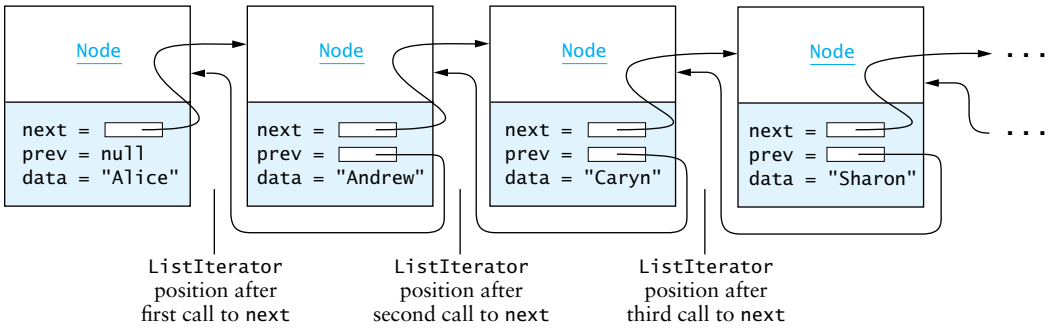


FIGURE 2.29 Inserting "Bill" before "Caryn" in an Ordered List

Implementation

Let's say we have an ordered list that contains the data: "Alice", "Andrew", "Caryn", "Sharon", and we want to insert "Bill" (see Figure 2.29). If we start at the beginning of the list and access "Alice", we know that "Bill" must follow "Alice", but we can't insert "Bill" yet. If we access "Andrew", we know that "Bill" must follow "Andrew", but we can't insert "Bill" yet. However, when we access "Caryn", we know we must insert "Bill" before "Caryn". Therefore, to insert an element in an ordered list, we need to access the first element whose data is larger than the data in the element to be inserted. Once we have accessed the successor of our new node, we can insert a new node just before it. (Note that in order to access "Caryn" using the method next, we have advanced the iterator just past "Caryn".)

Algorithm for Insertion

The algorithm for insertion is as follows:

1. Find the first item in the list that is greater than the item to be inserted.
2. Insert the new item before this one.

We can refine this algorithm as follows:

1.1 Create a ListIterator that starts at the beginning of the list.

1.2 while the ListIterator is not at the end and the item to be inserted is greater than or equal to the next item.

1.3 Advance the ListIterator.

2. Insert the new item before the current ListIterator position.

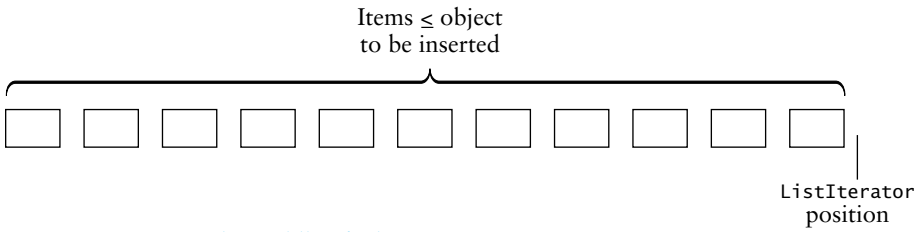
The add Method

A straightforward coding of the insertion algorithm would be the following:

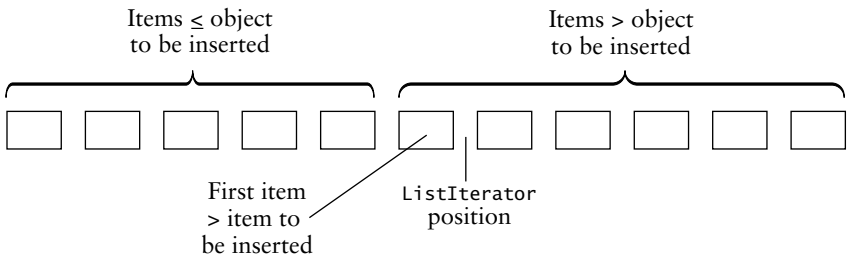
```
// WARNING - THIS DOES NOT WORK.
ListIterator<E> iter = theList.listIterator();
while (iter.hasNext()
      && obj.compareTo(iter.next()) >= 0) {
    // iter was advanced - check new position.
}
iter.add(obj);
```

FIGURE 2.30
Attempted Insertion into an Ordered List

Case 1: Inserting at the end of a list



Case 2: Inserting in the middle of a list



Unfortunately, this does not work. When the `while` loop terminates, either we are at the end of the list or the `ListIterator` has just skipped over the first item that is greater than the item to be inserted (see Figure 2.30). In the first case, the `add` method will insert the item at the end of the list, just as we want, but in the second case, it will insert the item just after the position where it belongs. Therefore, we must separate the two cases and code the `add` method as follows:

```
/** Insert obj into the list preserving the list's order.
    @pre The items in the list are ordered.
    @post obj has been inserted into the list
          such that the items are still in order.
    @param obj The item to be inserted
 */
public void add(E obj) {
    ListIterator<E> iter = theList.listIterator();
    // Find the insertion position and insert.
    while (iter.hasNext()) {
        if (obj.compareTo(iter.next()) < 0) {
            // Iterator has stepped over the first element
            // that is greater than the element to be inserted.
            // Move the iterator back one.
            iter.previous();
            // Insert the element.
            iter.add(obj);
            // Exit the loop and return.
            return;
        }
    }
    // assert: All items were examined and no item is larger than
    // the element to be inserted.
    // Add the new item to the end of the list.
    iter.add(obj);
}
```

Using Delegation to Implement the Other Methods

The other methods in Table 2.10 are implemented via delegation to the `LinkedList` class. They merely call the corresponding method in the `LinkedList`. For example, the `get` and `iterator` methods are coded as follows:

```
/** Returns the element at the specified position.
    @param index The specified position
    @return The element at position index
 */
public E get(int index) {
    return theList.get(index);
}

/** Returns an iterator to this OrderedList.
    @return The iterator, positioning it before the first element
 */
public Iterator<E> iterator() {
    return theList.iterator();
}
```

PITFALL

Omitting <E> After `Listiterator<E>` or `Iterable<E>`

If you omit `<E>` in the declaration for `ListIterator<E> iter` in method `add` of `OrderedList`:

```
ListIterator<E> iter = theList.listIterator();
```

you will get an `incompatible types` syntax error when the `if` statement

```
if (obj.compareTo(iter.next()) < 0) {
```

is compiled. The reason is that the object returned by `iter.next()` will be type `Object` instead of type `E`, so the argument of `compareTo` will not be type `E` as required.

Similarly, if you omit the `<E>` after `Iterable` in the header for class `OrderedList`:

```
public class OrderedList<E extends <Comparable<E>> implements Iterable<E> {
```

you will get an `incompatible types` syntax error if an `Iterator` method or an enhanced `for` loop is compiled. The reason is that the data type of the object returned by an `Iterator` method would be `Object`, not type `E` as required.

Testing Class OrderedList

Next, we illustrate the design of a class that tests our implementation of the `OrderedList`. The next chapter provides a thorough discussion of testing.

Testing

You can test the `OrderedList` class by storing a collection of randomly generated positive integers in an `OrderedList`. You can then insert a negative integer and an integer larger than any integer in the list. This tests the two special cases of inserting at the beginning and at the end of the list. You can then create an iterator and use it to traverse the list, displaying an error message if the current integer is smaller than the previous integer, which is an indication that the list is not ordered. You can also display the list during the traversal so that you

can inspect it to verify that it is in order. Finally, you can remove the first element, the last element, and an element in the middle and repeat the traversal to show that removal does not affect the ordering. Listing 2.2 shows a class with methods that performs these tests.

Method `traverseAndShow` traverses an ordered list passed as an argument using an enhanced `for` statement to access the list elements. Each `Integer` is stored in `thisItem`. The `if` statement displays an error message if the previous value is greater than the current value (`prevItem > thisItem is true`). Method `main` calls `traverseAndShow` after all elements are inserted and after the three elements are removed. In method `main`, the loop

```
for (int i = 0; i < START_SIZE; i++) {
    int anInteger = random.nextInt(MAX_INT);
    testList.add(anInteger);
}
```

fills the ordered list with randomly generated values between `0` and `MAX_INT-1`. Variable `random` is an instance of class `Random` (in API `java.util`), which contains methods for generating pseudorandom numbers. Method `Random.nextInt` generates random integers between 0 and its argument. Chapter 3 provides a thorough discussion of testing.

LISTING 2.2

Class TestOrderedList

```
import java.util.*;
public class TestOrderedList {
    /** Traverses ordered list and displays each element.
        Displays an error message if an element is out of order.
        @param testList An ordered list of integers
     */
    public static void traverseAndShow(OrderedList<Integer> testList) {
        int prevItem = testList.get(0);

        // Traverse ordered list and display any value that
        // is out of order.
       for (Integer thisItem : testList) {
           System.out.println(thisItem);
           if (prevItem > thisItem)
               System.out.println("*** FAILED, value is " + thisItem);
           prevItem = thisItem;
       }
    }

    public static void main(String[] args) {
        OrderedList<Integer> testList = new OrderedList<>();
        final int MAX_INT = 500;
        final int START_SIZE = 100;

        // Create a random number generator.
        Random random = new Random();
        // Fill list with START_SIZE random values.
        for (int i = 0; i < START_SIZE; i++) {
            int anInteger = random.nextInt(MAX_INT);
            testList.add(anInteger);
        }

        // Add to beginning and end of list.
        testList.add(-1);
        testList.add(MAX_INT + 1);
        traverseAndShow(testList); // Traverse and display.
```

```
        // Remove first, last, and middle elements.
        Integer first = testList.get(0);
        Integer last = testList.get(testList.size() - 1);
        Integer middle = testList.get(testList.size() / 2);
        testList.remove(first);
        testList.remove(last);
        testList.remove(middle);
        traverseAndShow(testList); // Traverse and display.
    }
}
```

EXERCISES FOR SECTION 2.8

SELF-CHECK

1. Why don't we implement the `OrderedList` by extending `LinkedList`? What would happen if someone called the `add` method? How about the `set` method?
2. What other methods in the `List` interface could we include in the `OrderedList` class? See the Java API documentation for a complete list of methods.
3. How can we provide a `listIterator` method for the `OrderedList` class?

PROGRAMMING

1. Write the code for the other methods of the `OrderedList` class that are listed in Table 2.10.
2. Rewrite the `OrderedList.add` method to start at the end of the list and iterate using the `ListIterator`'s `previous` method.

2.9 Implementation of a Double-Linked List Class

In this section, we describe the class `KWLinkedList` that implements some of the methods of the `List` interface using a double-linked list. We will not provide a complete implementation because we expect you to use the standard `LinkedList` class provided by the Java API (in package `java.util`). The data fields for the `KWLinkedList` class are shown in Table 2.11. They are declared as shown here.

```
import java.util.*;

/** Class KWLinkedList implements a double-linked list and
    a ListIterator. */
public class KWLinkedList<E> {
    // Data Fields
    /** A reference to the head of the list. */
    private Node<E> head = null;
    /** A reference to the end of the list. */
    private Node<E> tail = null;
    /** The size of the list. */
    private int size = 0;
```

TABLE 2.11
Data Fields for Class `KWLinkedList<E>`

Data Field	Attribute
`private Node<E> head`	A reference to the first item in the list
`private Node<E> tail`	A reference to the last item in the list
`private int size`	A count of the number of items in the list

Implementing the `KWLinkedList` Methods

We need to implement the methods shown in Table 2.6 for the `LinkedList` class. The algorithm for the `add(int index, E obj)` method is

1. Obtain a reference, `nodeRef`, to the node at position `index`.
2. Insert a new `Node` containing `obj` before the `Node` referenced by `nodeRef`.

Similarly, the algorithm for the `get(int index)` method is

1. Obtain a reference, `nodeRef`, to the node at position `index`.
2. Return the contents of the `Node` referenced by `nodeRef`.

We also have the `listIterator(int index)` method with the following algorithm:

1. Obtain a reference, `nodeRef`, to the node at position `index`.
2. Return a `ListIterator` that is positioned just before the `Node` referenced by `nodeRef`.

These three methods all have the same first step. Therefore, we want to use a common method to perform this step.

If we look at the requirements for the `ListIterator`, we see that it has an `add` method that inserts a new item before the current position of the iterator. Thus, we can refine the algorithm for the `KWLinkedList.add(int index, E obj)` method to

1. Obtain an iterator that is positioned just before the `Node` at position `index`.
2. Insert a new `Node` containing `obj` before the `Node` currently referenced by this iterator.

Thus, the `KWLinkedList<E>` method `add` can be coded as

```
/** Add an item at position index.
    @param index The position at which the object is to be
           inserted
    @param obj The object to be inserted
    @throws IndexOutOfBoundsException if the index is out
            of range (i < 0 || i > size())
 */
public void add(int index, E obj) {
    listIterator(index).add(obj);
}
```

Note that it was not necessary to declare a local `ListIterator` object in the `KWLinkedList` method `add`. The method call `listIterator(index)` returns an anonymous `ListIterator` object, to which we apply the `ListIterator.add` method.

Similarly, we can code the `get` method as

```
/** Get the element at position index.
    @param index Position of item to be retrieved
    @return The item at index
 */
```

```
public E get(int index) {
    return listIterator(index).next();
}
```

Other methods in Table 2.6 (`addFirst`, `addLast`, `getFirst`, `getLast`) can be implemented by delegation to methods `add` and `get` above.

A Class That Implements the `ListIterator` Interface

We can implement most of the `KWLinkedList` methods by delegation to the class `KWListIter`, which will implement the `ListIterator` interface (see Table 2.8). Because it is an inner class of `KWLinkedList`, its methods will be able to reference the data fields and members of the parent class (and also the other inner class, `Node`). The data fields for class `KWListIter` are shown in Table 2.12.

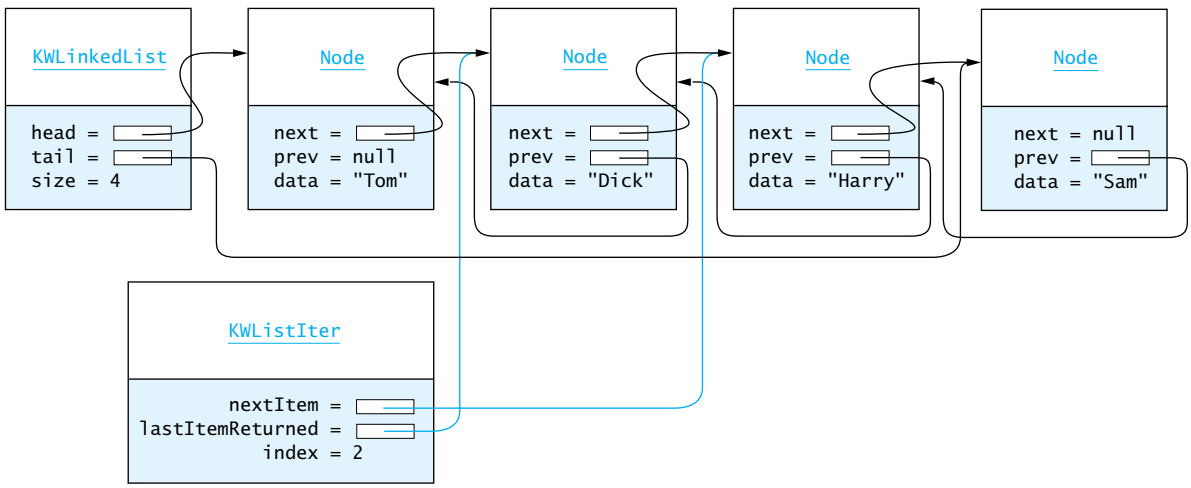
```
/** Inner class to implement the ListIterator interface. */
private class KWListIter implements ListIterator<E> {
    /** A reference to the next item. */
    private Node<E> nextItem;
    /** A reference to the last item returned. */
    private Node<E> lastItemReturned;
    /** The index of the current item. */
    private int index = 0;
```

Figure 2.31 shows an example of a `KWLinkedList` object and a `KWListIter` object. The `next` method would return `"Harry"`, and the `previous` method would return `"Dick"`. The `nextIndex` method would return 2, and the `previousIndex` method would return 1.

TABLE 2.12
Data Fields of Class `KWListIter`

`private Node<E> nextItem`	A reference to the next item
`private Node<E> lastItemReturned`	A reference to the node that was last returned by `next` or `previous`
`private int index`	The iterator is positioned just before the item at `index`

FIGURE 2.31
Double-Linked List with `KWListIter`



The Constructor

The KWListIter constructor takes as a parameter the index of the Node at which the iteration is to begin. A test is made for the special case where the index is equal to the size; in that case, the iteration starts at the tail. Otherwise, a loop starting at the head walks along the list until the node at index is reached.

```
/** Construct a KWListIter that will reference the ith item.
    @param i The index of the item to be referenced
 */
public KWListIter(int i) {
    // Validate i parameter.
    if (i < 0 || i > size) {
        throw new IndexOutOfBoundsException("Invalid index " + i);
    }
    lastItemReturned = null; // No item returned yet.
    // Special case of last item.
    if (i == size) {
        index = size;
        nextItem = null;
    } else { // Start at the beginning
        nextItem = head;
        for (index = 0; index < i; index++) {
             nextItem = nextItem.next;
        }
    }
}
```

The hasNext and next Methods

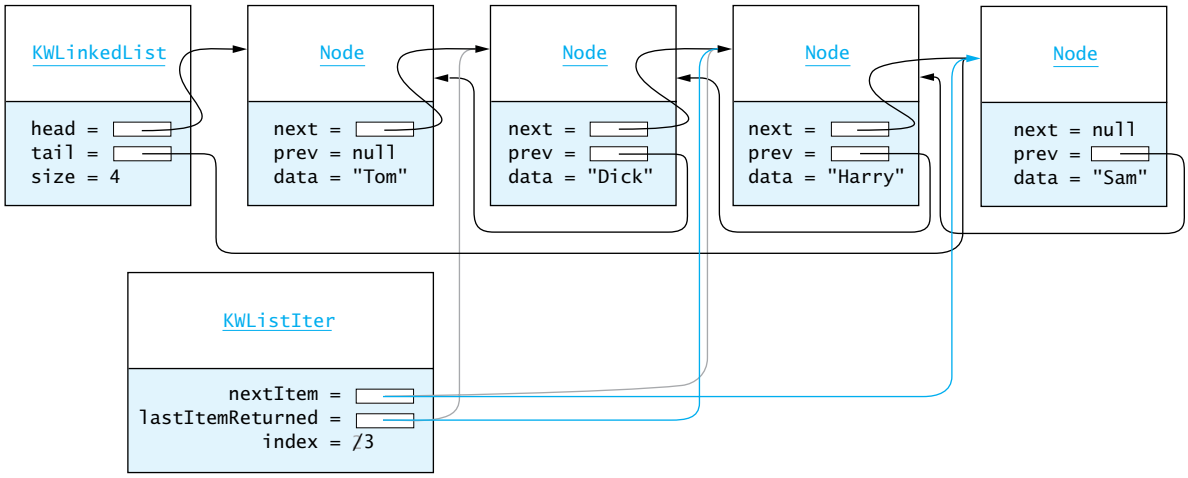
The data field nextItem will always reference the Node that will be returned by the next method. Therefore, the hasNext method merely tests to see whether nextItem is null.

```
/** Indicate whether movement forward is defined.
    @return true if call to next will not throw an exception
 */
public boolean hasNext() {
    return nextItem != null;
}
```

The next method begins by calling hasNext. If the result is false, the NoSuchElemenentException is thrown. Otherwise, lastItemReturned is set to nextItem; then nextItem is advanced to the next node, and index is incremented. The data field of the node referenced by lastItemReturned is returned. As shown in Figure 2.32, the previous iterator position is indicated by the gray arrows and the new position by the blue arrows.

```
/** Move the iterator forward and return the next item.
    @return The next item in the list
    @throws NoSuchElementException if there is no such object
 */
public E next() {
    if (!hasNext()) {
        throw new NoSuchElementException();
    }
    lastItemReturned = nextItem;
    nextItem = nextItem.next;
    index++;
    return lastItemReturned.data;
}
```

FIGURE 2.32
Advancing a `KWListIter`



The hasPrevious and previous Methods

The `hasPrevious` method is a little trickier. When the iterator is at the end of the list, `nextItem` is `null`. In this case, we can determine that there is a previous item by checking the size—a nonempty list will have a previous item when the iterator is at the end. If the iterator is not at the end, then `nextItem` is not `null`, and we can check for a previous item by examining `nextItem.prev`.

```
/** Indicate whether movement backward is defined.
    @return true if call to previous will not throw an exception
 */
public boolean hasPrevious() {
    return (nextItem == null && size != 0)
            || nextItem.prev != null;
}
```

The `previous` method begins by calling `hasPrevious`. If the result is `false`, the `NoSuchElementException` is thrown. Otherwise, if `nextItem` is `null`, the iterator is past the last element, so `nextItem` is set to `tail` because the previous element must be the last list element. If `nextItem` is not `null`, `nextItem` is set to `nextItem.prev`. Either way, `lastItemReturned` is set to `nextItem`, and index is decremented. The `data` field of the node referenced by `lastItemReturned` is returned.

```
/** Move the iterator backward and return the previous item.
    @return The previous item in the list
    @throws NoSuchElementException if there is no such object
 */
public E previous() {
    if (!hasPrevious()) {
        throw new NoSuchElementException();
    }
    if (nextItem == null) { // Iterator is past the last element
        nextItem = tail;
    } else {
        nextItem = nextItem.prev;
    }
    lastItemReturned = nextItem;
    index--;
    return lastItemReturned.data;
}
```

The add Method

The add method inserts a new node before the node referenced by nextItem. There are four cases: add to an empty list, add to the head of the list, add to the tail of the list, and add to the middle of the list. We next discuss each case separately; you can combine them to write the method.

An empty list is indicated by head equal to null. In this case, a new Node is created, and both head and tail are set to reference it. This is illustrated in Figure 2.33.

```
/** Add a new item between the item that will be returned
    by next and the item that will be returned by previous.
    If previous is called after add, the element added is
    returned.
    @param obj The item to be inserted
 */
public void add(E obj) {
    if (head == null) { // Add to an empty list.
        head = new Node<>(obj);
        tail = head;
    ...
```

The KWListIter object in Figure 2.33 shows a value of null for lastItemReturned and 1 for index. These data fields are set at the end of the method. In all cases, data field nextItem is not changed by the insertion. It must reference the successor of the item that was inserted, or null if there is no successor.

If nextItem equals head, then the insertion is at the head. The new Node is created and is linked to the beginning of the list.

```
} else if (nextItem == head) { // Insert at head.
    // Create a new node.
    Node<E> newNode = new Node<>(obj);
    // Link it to the nextItem.
    newNode.next = nextItem;  // Step 1
    // Link nextItem to the new node.
    nextItem.prev = newNode;  // Step 2
    // The new node is now the head.
    head = newNode;           // Step 3
```

This is illustrated in Figure 2.34. In Figures 2.34–2.36, the gray arrows show old links and the blue arrows show new links.

If nextItem is null, then the insertion is at the tail. The new Node is created and linked to the tail.

```
} else if (nextItem == null) { // Insert at tail.
    // Create a new node.
    Node<E> newNode = new Node<>(obj);
    // Link the tail to the new node.
    tail.next = newNode; // Step 1
    // Link the new node to the tail.
    newNode.prev = tail; // Step 2
    // The new node is the new tail.
    tail = newNode;      // Step 3
```

This is illustrated in Figure 2.35.

If none of the previous cases is true, then the addition is into the middle of the list. The new node is created and inserted before the node referenced by nextItem.

```
} else { // Insert into the middle.
    // Create a new node.
```

FIGURE 2.33
Adding to an Empty List

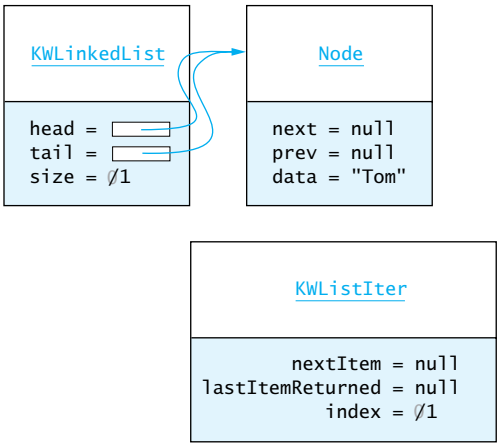
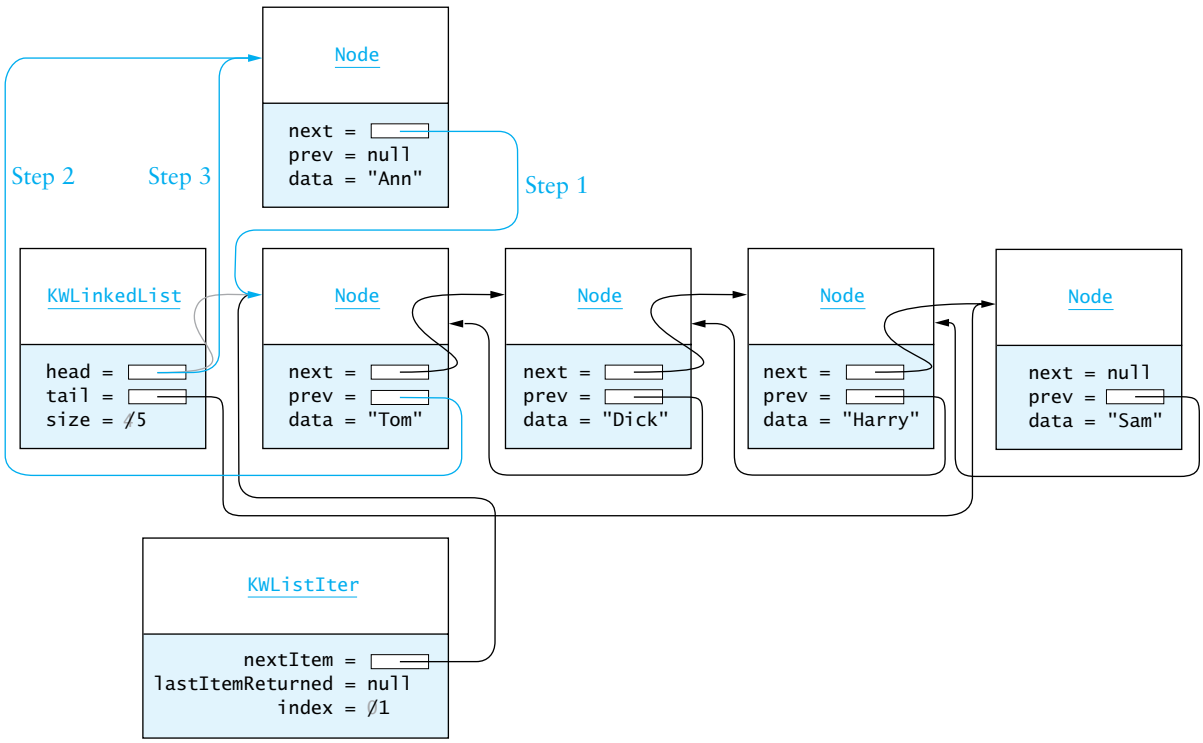
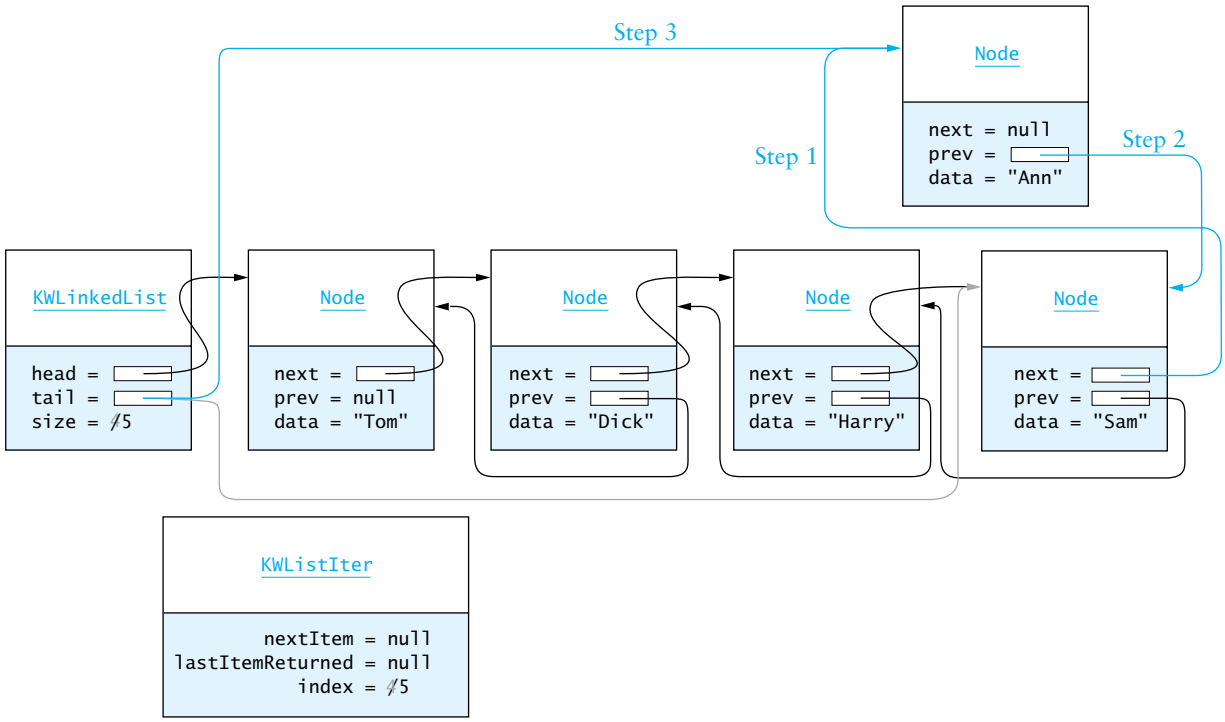


FIGURE 2.34
Adding to the Head of the List




```
        Node<E> newNode = new Node<>(obj);
        // Link it to nextItem.prev.
        newNode.prev = nextItem.prev; // Step 1
        nextItem.prev.next = newNode; // Step 2
        // Link it to the nextItem.
        newNode.next = nextItem; // Step 3
        nextItem.prev = newNode; // Step 4
    }
```

FIGURE 2.35
Adding to the Tail of the List



This is illustrated in Figure 2.36.

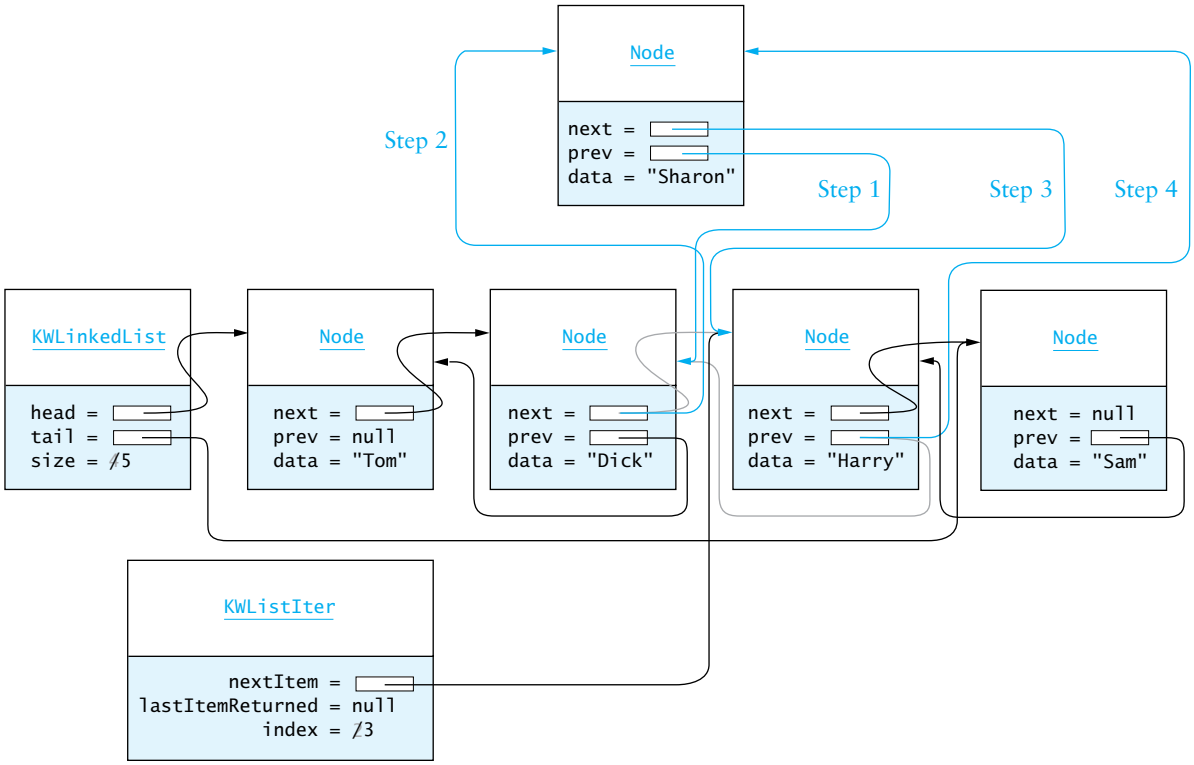
After the new node is inserted, both `size` and `index` are incremented and `lastItemReturned` is set to `null`.

```
        // Increase size and index and set lastItemReturned.
        size++;
        index++;
        lastItemReturned = null;
    } // End of method add.
```

Inner Classes: Static and Nonstatic

There are two inner classes in class `KWLinkedList<E>`: class `Node<E>` and class `KWListIter`. We declare `Node<E>` to be static because there is no need for its methods to access the data fields of its parent class (`KWLinkedList<E>`). We can't declare `KWListIter` to be static because its methods access and modify the data fields of the `KWLinkedList` object that creates the `KWListIter` object. An inner class that is not static contains an implicit reference to its parent object, just as it contains an implicit reference to itself. Because `KWListIter` is not static and can reference data fields of its parent class `KWLinkedList<E>`, the type parameter `<E>` is considered to be previously defined; therefore, it cannot appear as part of the class name.

FIGURE 2.36
Adding to the Middle of the List



PITFALL

Defining `KWListIter` as a Generic Inner Class

If you define class `KWListIter` as

```
private class KWListIter<E>. . .
```

you will get an `incompatible` types syntax error when you attempt to reference data field `head` or `tail` (type `Node<E>`) inside class `KWListIter`.

EXERCISES FOR SECTION 2.9

SELF-CHECK

1. Why didn't we write the `hasPrevious` method as follows?

```
public boolean hasPrevious() {
    return nextItem.prev != null
           || (nextItem == null && size != 0);
}
```

2. Why must we call `next` or `previous` before we call `remove`?
3. What happens if we call `remove` after we call `add`? What does the Java API documentation say? What does our implementation do?

PROGRAMMING

1. Implement the `KWListIter.remove` method.
2. Implement the `KWListIter.set` method.
3. Implement the `KWLinkedList listIterator` and `iterator` methods.
4. Implement the `KWLinkedList addFirst`, `addLast`, `getFirst`, and `getLast` methods.

2.10 The Collections Framework Design

The `Collection` Interface

The `Collection` interface specifies a subset of the methods specified in the `List interface`. Specifically, the `add(int, E)`, `get(int)`, `remove(int)`, `set(int, E)`, and related methods (all of which have an `int` parameter that represents a position) are not in the `Collection` interface, but the `add(E)` and `remove(Object)` methods, which do not specify a position, are included. The `iterator` method is also included in the `Collection` interface. Thus, you can use an `Iterator` to access all of the items in a `Collection`, but the order in which they are retrieved is not necessarily related to the order in which they were inserted.

The `Collection` interface is part of the Collections Framework as shown in Figure 2.37. This interface has three subinterfaces: the `List` interface, the `Queue` interface (Chapter 4), and the `Set` interface (Chapter 7). The Java API does not provide any direct implementation of the `Collection` interface. The interface is used to reference collections of data in the most general way.

Common Features of Collections

Because it is the superinterface of `List`, `Queue`, and `Set`, the `Collection` interface specifies a set of common methods. If you look at the documentation for the Java API `java.util.Collection`, you will see that this is a fairly large set of methods and other requirements. A few features can be considered fundamental:

- Collections grow as needed.
- Collections hold references to objects.
- Collections have at least two constructors: one to create an empty collection and one to make a copy of another collection.

Table 2.13 shows selected methods defined in the `Collection` interface. We have already seen and described these methods in the discussions of the `ArrayList` and `LinkedList`. The `Iterator` provides a common way to access all of the elements in a `Collection`. For collections implementing the `List` interface, the order of the elements is determined by the index of the elements. In the more general `Collection`, the order is not specified.

FIGURE 2.37
The Collections Framework

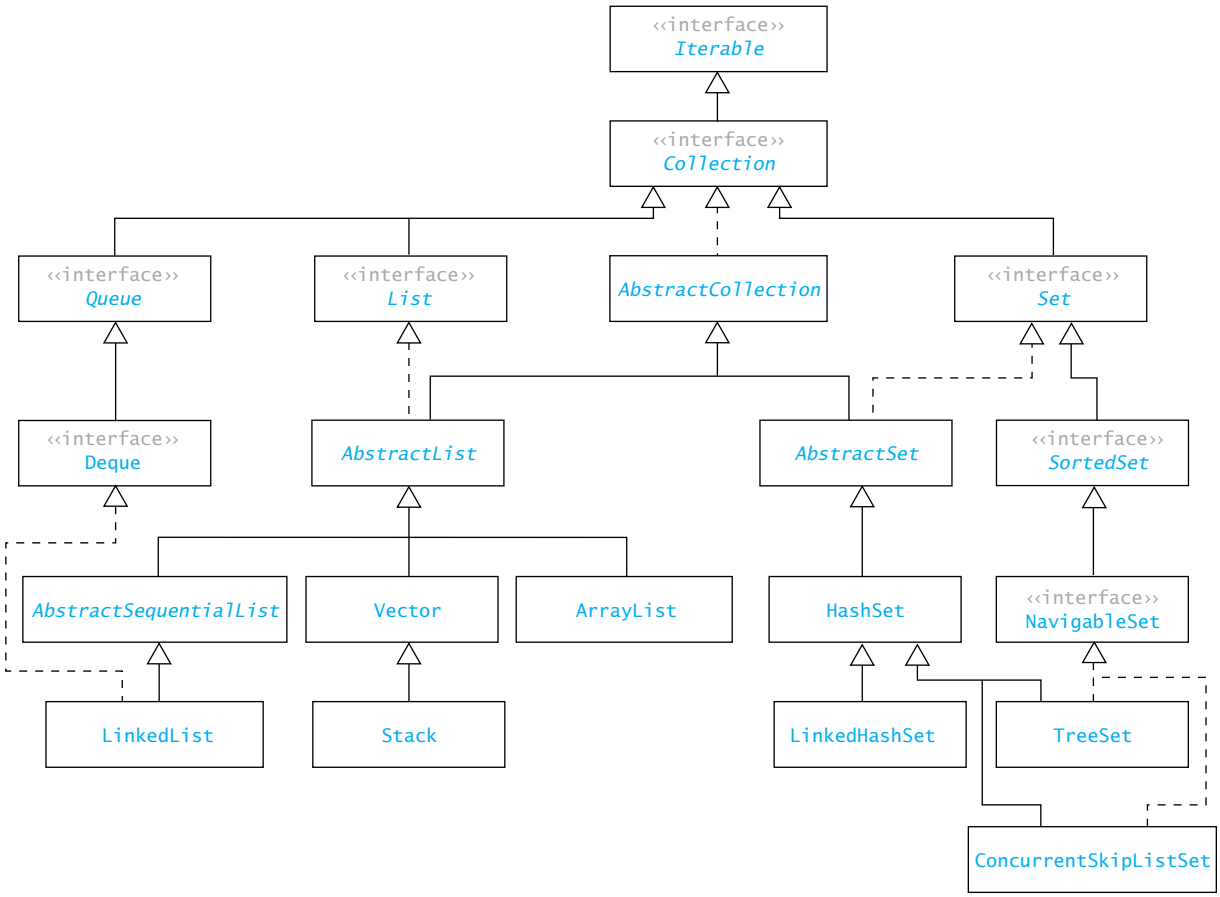


TABLE 2.13
Selected Methods of the `java.util.Collection<E>` Interface

Method	Behavior
`boolean add(E obj)`	Ensures that the collection contains the `object obj`. Returns `true` if the collection was modified
`boolean contains(E obj)`	Returns `true` if the collection contains the object `obj`
`Iterator<E> iterator()`	Returns an `Iterator` to the collection
`int size()`	Returns the `size` of the collection

In the `ArrayList` and `LinkedList`, the `add(E)` method always inserts the object at the end and always returns `true`. In the more general `Collection`, the position where the object is inserted is not specified. The `Set` interface extends the `Collection` by requiring that the `add` method not insert an object that is already present; instead, in that case it returns `false`. The `Set` interface is discussed in Chapter 7.

The AbstractCollection, AbstractList, and AbstractSequentialList Classes

If you look at the Java API documentation, you will see that the Collection and List interfaces specify a large number of methods. To help implement these interfaces, the Java API includes the AbstractCollection and AbstractList classes. You can think of these classes as a kit (or as a cake mix) that can be used to build implementations of their corresponding interface. Most of the methods are provided, but you need to add a few to make it complete.

To implement the Collection interface completely, you need to only extend the AbstractCollection class, provide an implementation of the add, size, and iterator methods, and supply an inner class to implement the Iterator interface. To implement the List interface, you can extend the AbstractList class and provide an implementation of the add(int, E), get(int), remove(int), set(int, E), and size() methods. Since we provided these methods in our KWArrayList, we can make it a complete implementation of the List interface by changing the class declaration to

```
public class KWArrayList<E> extends AbstractList<E> implements List<E>
```

Note that the AbstractList class implements the iterator and listIterator methods using the index associated with the elements.

Another way to implement the List interface is to extend the AbstractSequentialList class, implement the listIterator and size methods, and provide an inner class that implements the ListIterator interface. This was the approach we took in our KWLinkedList. Thus, by changing the class declaration to

```
public class KWLinkedList<E> extends AbstractSequentialList<E>
                             implements List<E>
```

it becomes a complete implementation of the List interface. Our KWLinkedList class included the add, get, remove, and set methods. These are provided by the AbstractSequentialList, so we could remove them from our KWLinkedList class and still have a complete List implementation.

The List and RandomAccess Interfaces (Advanced)

The ArrayList and the LinkedList implement the List interface that we described in Section 2.2. Both the ArrayList and LinkedList represent a collection of objects that can be referenced using an index. This may not be the best design because accessing elements of a LinkedList using an index requires an O(n) traversal of the list until the item selected by the index is located. Unfortunately, the Java designers cannot easily change the design of the API since a lot of programs have been written and the users of Java do not want to go back and change their code. Also, there are other implementations of the List interface in which the indexed operations get and set are approximately O(n) instead of O(1).

The RandomAccess interface is applied only to those implementations in which indexed operations are efficient (e.g., ArrayList). An algorithm can then test to see if a parameter of type List is also of type RandomAccess and, if not, copy its contents into an ArrayList temporarily so that the indexed operations can proceed more efficiently. After the indexed operations are completed, the contents of the ArrayList are copied back to the original.

EXERCISES FOR SECTION 2.10

SELF-CHECK

1. Look at the `AbstractCollection` definition in the Java API documentation. What methods are abstract? Could we use the `KWArrayList` and extend the `AbstractCollection`, but not the `AbstractList`, to develop an implementation of the `Collection` interface? How about using the `KWLinkedList` and the `AbstractCollection`, but not the `AbstractSequentialList`?

PROGRAMMING

1. Using either the `KWArrayList` or the `KWLinkedList` as the base, develop an implementation of the `Collection` interface by extending the `AbstractCollection`. Test it by ensuring that the following statements compile:

```
Collection<String> testCollection = new KWArrayList<>();
Collection<String> testCollection = new KWLinkedList<>();
```

2. While constructors cannot be specified in interfaces, members of the Collection Framework are required to provide two constructors: one to create an empty collection and the other to make a copy of an existing collection. Add the following copy constructors to `KWArrayList` and `KWLinkedList`:

```
public KWArrayList<E>(Collection<E> c)
public KWLinkedList<E>(Collection<E> c)
```

Chapter Review

- We use big-O notation to describe the performance of an algorithm. Big-O notation specifies how the performance increases with the number of data items being processed by an algorithm. The best performance is O(1), which means the performance is constant regardless of the number of data items processed.
- The `List` is a generalization of an array. As in the array, elements of a `List` are accessed by means of an index. Unlike the array, the `List` can grow or shrink. Items may be inserted or removed from any position.
- The Java API provides the `ArrayList<E>` class, which uses an array as the underlying structure to implement the `ArrayList`. We provided an example of how this might be implemented by allocating an array that is larger than the number of items in the list. As items are inserted into the list, the items with higher indices are moved up to make room for the inserted item, and as items are removed, the items with higher indices are moved down to fill in the emptied space. When the array capacity is reached, a new array is allocated that is twice the size and the old array is copied to the new one. By doubling the capacity, the cost of the copy is spread over each insertion so that the copies can be considered to have a constant time contribution to the cost of each insertion.

- A linked list data structure consists of a set of nodes, each of which contains its data and a reference to the next node in the list. In a double-linked list, each node contains a reference to both the next node and the previous node in the list. Insertion into and removal from a linked list is a constant-time operation.
- To access an item at a position indicated by an index in a linked list requires walking along the list from the beginning until the item at the specified index is reached. Thus, traversing a linked list using an index would be an $O(n^2)$ operation because we need to repeat the walk each time the index changes. The `Iterator` provides a general way to traverse a list so that traversing a linked list using an iterator is an $O(n)$ operation.
- An iterator provides us with the ability to access the items in a `List` sequentially. The `Iterator` interface defines the methods available to an iterator. The `List` interface defines the `iterator` method, which returns an `Iterator` to the list. The `Iterator.hasNext` method tells whether there is a next item, and the `Iterator.next` method returns the next item and advances the iterator. The `Iterator` also provides the `remove` method, which lets us remove the last item returned by the `next` method.
- The `ListIterator` interface extends the `Iterator` interface. The `ListIterator` provides us with the ability to traverse the list either forward or backward. In addition to the `hasNext` and `next` methods, the `ListIterator` has the `hasPrevious` and `previous` methods. Also, in addition to the `remove` method, it has an `add` method that inserts a new item into the list just before the current iterator position.
- The `Iterable` interface is implemented by the `Collection` interface. It imposes a requirement that its implementers (all classes that implement the `Collection` interface) provide an `iterator` method that returns an `Iterator` to an instance of that collection class. The enhanced `for` loop makes it easier to iterate through these collections without explicitly manipulating an iterator and also to iterate through an array object without manipulating an array index.
- The Java API provides the `LinkedList` class, which uses a double-linked list to implement the `List` interface. We show an example of how this might be implemented. Because the class that realizes the `ListIterator` interface provides the `add` and `remove` operations, the corresponding methods in the linked list class can be implemented by constructing an iterator (using the `listIterator(int)` method) that references the desired position and then calling on the iterator to perform the insertion or removal.
- The `Collection` interface is the root of the Collections Framework. The `Collection` is more general than the `List` because the items in a `Collection` are not indexed. The `add` method inserts an item into a `Collection` but does not specify where it is inserted. The `Iterator` is used to traverse the items in a `Collection`, but it does not specify the order of the items.
- The `Collection` interface and the `List` interface define a large number of methods that make these abstractions useful for many applications. In our discussion of both the `ArrayList` and `LinkedList`, we showed how to implement only a few key methods. The Collections Framework includes the `AbstractCollection`, `AbstractList`, and `AbstractSequentialList` classes. These classes implement their corresponding interface except for a few key methods; these are the same methods for which we showed implementations.

Java API Interfaces and Classes Introduced in This Chapter

```
java.uti1.AbstractCollection
java.uti1.AbstractList
java.uti1.AbstractSequentialList
java.util.ArrayList
java.util.Collection
java.uti1.Iterable
java.util.Iterator
java.util.LinkedList
java.util.List
java.util.ListIterator
java.util.RandomAccess
```

User-Defined Interfaces and Classes in this Chapter

KWArrayList	Node
KWLinkedList	OrderedList
KWListIter	SingleLinkedList

Quick-Check Exercises

1. Elements of a `List` are accessed by means of _____.
2. A `List` can _____ or _____as items are added or removed.
3. When we allocate a new array for an `ArrayList` because the current capacity is exceeded, we make the new array at least_____. This allows us to_____.
4. Determine the order of magnitude (big-O) for an algorithm whose running time is given by the equation $T(n) = 3n^4 - 2n^2 + 100n + 37$.
5. In a single-linked list, if we want to remove a list element, which list element do we need to access? If `nodeRef` references this element, what statement removes the desired element?
6. Suppose a single-linked list contains three nodes with data `"him"`, `"her"`, and `"it"`, and `head` references the first element. What is the effect of the following fragment?

```
Node<String> nodeRef = head.next;
nodeRef.data = "she";
```

7. Answer Question 5 for the following fragment.

```
Node<String> nodeRef = head.next;
head.next = nodeRef.next;
```

8. Answer Question 5 for the following fragment.

```
head = new Node<String>("his", head);
```

9. An `Iterator` allows us to access items of a `List`_____.
10. A `ListIterator` allows us to access the elements _____.
11. The Java `LinkedList` class uses a _____to implement the `List` interface.
12. The `Collection` is a _____of the `List`.

Review Questions

1. What is the difference between the size and the capacity of an `ArrayList`? Why might we have a constructor that lets us set the initial capacity?
2. What is the difference between the `remove(Object obj)` and `remove(int index)` methods?
3. When we insert an item into an `ArrayList`, why do we start shifting at the last element?
4. The `Vector` and `ArrayList` both provide the same methods, since they both implement the `List` interface. The `Vector` has some additional methods with the same functionality but different names. For example, the `Vector addElement` and `add` methods have the same functionality. There are some methods that are unique to `Vector`. Look at the Java API documentation and make a list of the methods that are in `Vector` that have equivalent methods in `ArrayList` and ones that are unique. Can the unique methods be implemented using the methods available in `ArrayList`?
5. If a loop processes n items and n changes from 1024 to 2048, how does that affect the running time of a loop that is $O(n^2)$? How about a loop that is $O(\log n)$? How about a loop that is $O(n \log n)$?
6. What is the advantage of a double-linked list over a single-linked list? What is the disadvantage?
7. Why is it more efficient to use an iterator to traverse a linked list?
8. What is the difference between the `Iterator` and `ListIterator` interfaces?

9. How would you make a copy of a `ListIterator`? Consider the following:

```
ListIterator copyOfIter =
    myList.ListIterator(otherIterator.previousIndex());
```

   Is this an efficient approach? How would you modify the `KWLinkedList` class to provide an efficient method to copy a `ListIterator`?

10. What is a `Collection`? Are there any classes in the Java API that completely implement the `Collection` interface?

Programming Projects

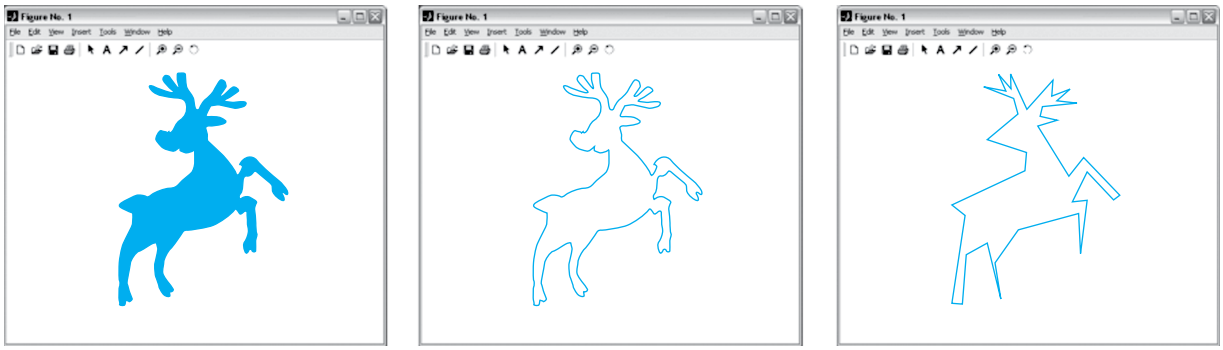
1. Develop a program to maintain a list of homework assignments. When an assignment is assigned, add it to the list, and when it is completed, remove it. You should keep track of the due date. Your program should provide the following services:
   - Add a new assignment.
   - Remove an assignment.
   - Provide a list of the assignments in the order they were assigned.
   - Find the assignment(s) with the earliest due date.
2. We can represent a polynomial as a list of terms, where the terms are in decreasing order by exponent. You should define a class `Term` that contains data fields `coef` and `exponent`. For example, $-5x^4$ has a `coef` value of -5 and an `exponent` value of 4. To add two polynomials, you traverse both lists and examine the two terms at the current iterator position. If the exponent of one is smaller than the exponent of the other, then insert the larger one into the result and advance that list's iterator. If the exponents are equal, then create a new term with that exponent and the sum of the two coefficients, and advance both iterators. For example:

$$3x^4 + 2x^2 + 3x + 7 \text{ added to } 2x^3 + -5x + 5 \text{ is } 3x^4 + 2x^3 + 2x^2 + -2x + 12$$

   Write a `polynomial` class with an inner class `Term`. The `polynomial` class should have a data field `terms` that is of type `LinkedList<Term>`. Provide an `add` method and a `readPoly` method. Method `readyPoly` reads a string representing a polynomial such as `"2x^3 - 4x^2"` and returns a polynomial list with two terms. You also need a `toString` method for class `Term` and `Polynomial` that would display this stored polynomial `2x^3 - 4x^2`.
3. Provide a multiply method for your polynomial class. To multiply, you iterate through polynomial A and then multiply all terms of polynomial B by the current term of polynomial A. You then add each term you get by multiplying two terms to the polynomial result. *Hint*: To multiply two terms, multiply their coefficients and add their exponents. For example, $2x^3 \times 4x^2$ is $8x^5$.
4. Write a program to manage a list of students waiting to register for a course as described in Section 2.5. Operations should include adding a new student at the end of the list, adding a new student at the beginning of the list, removing the student from the beginning of the list, and removing a student by name.
5. A circular-linked list has no need of a `head` or `tail`. Instead, you need only a reference to a current node, which is the `nextNode` returned by the `Iterator`. Implement such a `CircularList` class. For a nonempty list, the `Iterator.hasNext` method will always return `true`.
6. The Josephus problem is named after the historian Flavius Josephus, who lived between the years 37 and 100 CE. Josephus was also a reluctant leader of the Jewish revolt against the Roman Empire. When it appeared that Josephus and his band were to be captured, they resolved to kill themselves. Josephus persuaded the group by saying, "Let us commit our mutual deaths to determination by lot. He to whom the first lot falls, let him be killed by him that hath the second lot, and thus fortune shall make its progress through us all; nor shall any of us perish by his own right hand, for it would be unfair if, when the rest are gone, somebody should repent

and save himself" (Flavius Josephus, *The Wars of the Jews*, Book III, Chapter 8, Verse 7, tr. William Whiston, 1737). Yet that is exactly what happened; Josephus was left for last, and he and the person he was to kill surrendered to the Romans. Although Josephus does not describe how the lots were assigned, the following approach is generally believed to be the way it was done. People form a circle and count around the circle some predetermined number. When this number is reached, that person receives a lot and leaves the circle. The count starts over with the next person. Using the circular-linked list developed in Exercise 4, simulate this problem. Your program should take two parameters: n, the number of people who start, and m, the number of counts. For example, try $n = 20$ and $m = 12$. Where does Josephus need to be in the original list so that he is the last one chosen?

7. A two-dimensional shape can be defined by its boundary-polygon, which is simply a list of all coordinates ordered by a traversal of its outline. See the following figure for an example.

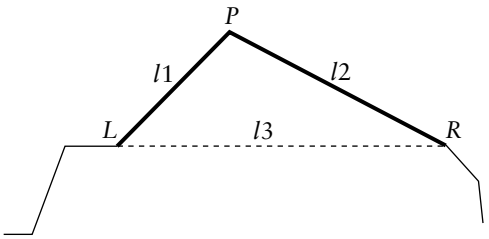


The left picture shows the original shape; the middle picture, the outline of the shape. The rightmost picture shows an abstracted boundary, using only the "most important" vertices. We can assign an importance measure to a vertex P by considering its neighbors L and R. We compute the distances LP, PR, and LR. Call these distances $l1$, $l2$, and $l3$. Define the importance as $l1 + l2 - l3$.

Use the following algorithm to find the n most important points.

1. `while` the number of points is greater than n
    2. Compute the importance of each point.
    3. Remove the least important one.

The Student Source Code on the textbook's website contains a program to read a data file, display the image and display the simplified image. It also shows a slider that lets you set the number of points. You need to provide the method `DrawingProcessor.simplifyDrawing`. *Note*: This problem and the algorithm for its solution are based on the paper L. J. Latecki and R. Lakämper, "Convexity Rule for Shape Decomposition Based on Discrete Contour Evolution," *Computer Vision and Image Understanding (CVIU)* 73(1999): 441–454.



Answers to Quick-Check Exercises

1. an index.
2. grow, shrink.
3. twice the size, spread out the cost of the reallocation so that it is effectively a constant-time operation.
4. O(n^4)
5. The predecessor of this node. `nodeRef.next = nodeRef.next.next;`
6. Replaces `"her"` with `"she"`.
7. Deletes the second list element (`"she"`).
8. Insert a new first element containing `"his"`.
9. sequentially.
10. both forward and backward.
11. double-linked list.
12. superinterface.