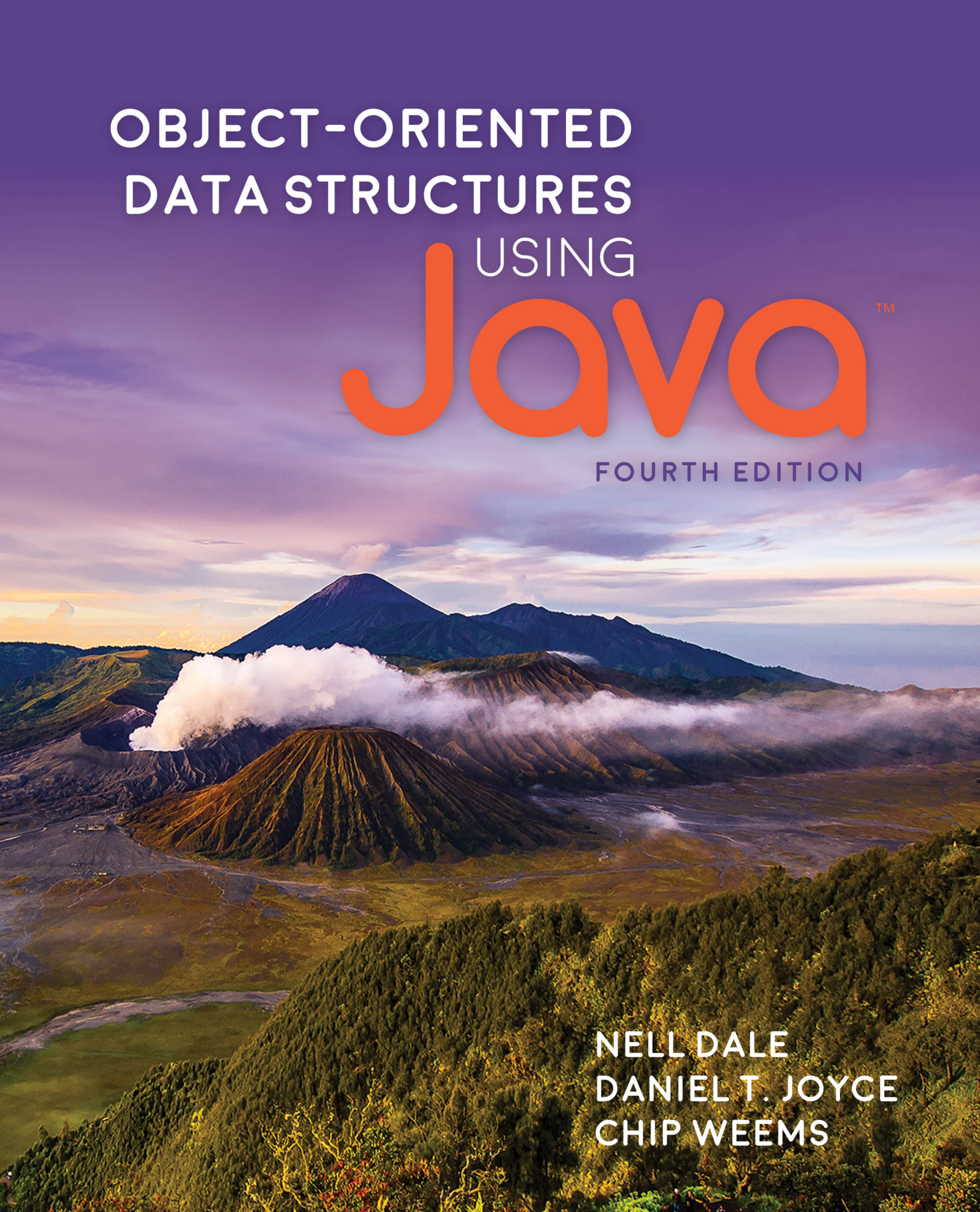
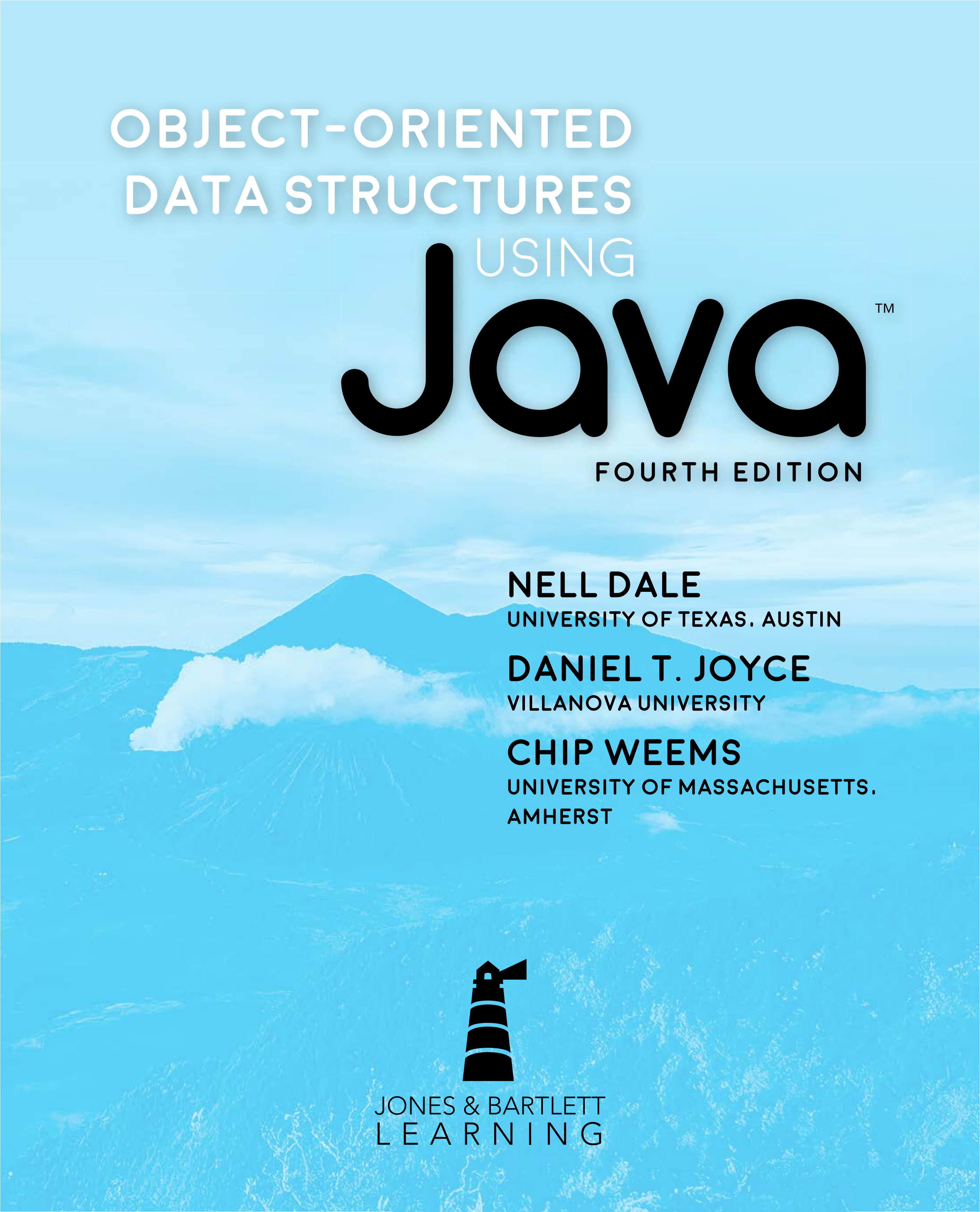

OBJECT-ORIENTED
DATA STRUCTURES
USING
Java™
FOURTH EDITION
NELL DALE
DANIEL T. JOYCE
CHIP WEEMS



OBJECT-ORIENTED DATA STRUCTURES USING Java™

FOURTH EDITION

NELL DALE
UNIVERSITY OF TEXAS, AUSTIN

DANIEL T. JOYCE
VILLANOVA UNIVERSITY

CHIP WEEMS
UNIVERSITY OF MASSACHUSETTS, AMHERST

JONES & BARTLETT LEARNING

World Headquarters
Jones & Bartlett Learning
5 Wall Street
Burlington, MA 01803
978-443-5000
info@jblearning.com
www.jblearning.com

Jones & Bartlett Learning books and products are available through most bookstores and online booksellers. To contact Jones & Bartlett Learning directly, call 800-832-0034, fax 978-443-8000, or visit our website, www.jblearning.com.

Substantial discounts on bulk quantities of Jones & Bartlett Learning publications are available to corporations, professional associations, and other qualified organizations. For details and specific discount information, contact the special sales department at Jones & Bartlett Learning via the above contact information or send an email to specialsales@jblearning.com.

Copyright © 2018 by Jones & Bartlett Learning, LLC, an Ascend Learning Company

All rights reserved. No part of the material protected by this copyright may be reproduced or utilized in any form, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without written permission from the copyright owner.

The content, statements, views, and opinions herein are the sole expression of the respective authors and not that of Jones & Bartlett Learning, LLC. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not constitute or imply its endorsement or recommendation by Jones & Bartlett Learning, LLC and such reference shall not be used for advertising or product endorsement purposes. All trademarks displayed are the trademarks of the parties noted herein. *Object-Oriented Data Structures Using Java, Fourth Edition* is an independent publication and has not been authorized, sponsored, or otherwise approved by the owners of the trademarks or service marks referenced in this product.

09820-4

Production Credits
VP, Executive Publisher: David D. Cella
Acquisitions Editor: Laura Pagluica
Editorial Assistant: Taylor Ferracane
Director of Vendor Management: Amy Rose
Marketing Manager: Amy Langlais
VP, Manufacturing and Inventory Control: Therese Connell
Composition and Project Management: S4Carlisle Publishing Services
Cover Design: Kristin E. Parker
Text Design: Scott Moden
Rights & Media Specialist: Merideth Tumasz
Media Development Editor: Shannon Sheehan
Cover Image: © Ake13bk/Shutterstock
Printing and Binding: Edwards Brothers Malloy
Cover Printing: Edwards Brothers Malloy

Library of Congress Cataloging-in-Publication Data
Names: Dale, Nell (Nell B.), author. | Joyce, Daniel T., author. | Weems, Chip., author.
Title: Object-oriented data structures using Java / Nell Dale, Daniel T. Joyce, Chip Weems.
Description: Fourth edition. | Burlington, MA : Jones & Bartlett Learning, [2017]
Identifiers: LCCN 2016025145 | ISBN 9781284089097 (casebound)
Subjects: LCSH: Object-oriented programming (Computer science) | Data structures (Computer science) | Java (Computer program language)
Classification: LCC QA76.64 .D35 2017 | DDC 005.13/3--dc23 LC record available at https://lccn.loc.gov/2016025145

6048

Printed in the United States of America
20 19 18 17 16 10 9 8 7 6 5 4 3 2 1

To Alfred G. Dale
ND

To Kathy, Tom, and Julie, thanks for the love and support
DJ

To Lisa, Charlie, and Abby, thank you . . .
CW


© Ake13bk/Shutterstock

Preface

Welcome to the fourth edition of *Object-Oriented Data Structures Using Java™*. This book presents the algorithmic, programming, and structuring techniques of a traditional data structures course in an object-oriented context. You'll find the familiar topics of linked lists, recursion, stacks, queues, collections, indexed lists, trees, maps, priority queues, graphs, sorting, searching, and complexity analysis, all covered from an object-oriented point of view using Java. We stress software engineering principles throughout, including modularization, information hiding, data abstraction, stepwise refinement, the use of visual aids, the analysis of algorithms, and software verification methods.

To the Student

You know that an algorithm is a sequence of unambiguous instructions for solving a problem. You can take a problem of moderate complexity, design a small set of classes/objects that work together to solve the problem, code the method algorithms needed to make the objects work, and demonstrate the correctness of your solution.

Algorithms describe actions. These actions manipulate data. For most interesting problems that are solved using computers, the structure of the data is just as important as the structure of the algorithms used to manipulate the data. Using this text you will discover that the way you structure data affects how efficiently you can use the data; you will see how the nature of the problem you are attempting to solve dictates your structuring decisions; and you will learn about the data structures that computer scientists have developed over the years to help solve problems.

Object-Oriented Programming with Java

Our primary goal is to present both the traditional and modern data structure topics with an emphasis on problem solving and software design. Using the Java programming language as a vehicle for problem solutions, however, presents an opportunity for students to expand their familiarity with a modern programming language and the object-oriented paradigm. As our data structure coverage unfolds, we introduce and use the appropriate Java constructs that support our primary goal. Starting early and continuing throughout the text, we introduce and expand on the use of many Java features such as classes, objects, generics, polymorphism, packages, interfaces, library classes, inheritance, exceptions, and threads. We also use Universal Modeling Language (UML) class diagrams throughout to help model and visualize our objects, classes, interfaces, applications, and their interrelationships.

Features

Data Abstraction In this text we view our data structures from three different perspectives: their specification, their application, and their implementation. The specification describes the logical or abstract level—*what* the logical relationships among the data elements are and *what* operations can be performed on the structure. The application level, sometimes called the client level, is concerned with how the data structure is used to solve a problem—*why* the operations do what they do. The implementation level involves the coding details—*how* the structures and operations are implemented. In other words we treat our data structures as abstract data types (ADTs).

Efficiency Analysis In Chapter 1 we introduce order of growth efficiency analysis using a unique approach involving the interaction of two students playing a game. Time and space analysis is consistently applied throughout the text, allowing us to compare and contrast data structure implementations and the applications that use them.

Recursion Treatment Recursion is introduced early (Chapter 3) and used throughout the remainder of the text. We present a design and analysis approach to recursion based on answering three simple questions. Answering the questions, which are based on formal inductive reasoning, leads the programmer to a solid recursive design and program.

Interesting Applications Eight primary data structures (stacks, queues, collections, indexed lists, trees, maps, priority queues, and graphs) are treated in separate chapters that include their definition, several implementations, and one or more interesting applications based on their use. Applications involve, for example, balanced expressions, postfix expressions, image generation (new!), fractals (new!), queue simulation, card decks and games (new!), text analysis (new!), tree and graph traversals, and big integers.

Robust Exercises We average more than 40 exercises per chapter. The exercises are organized by chapter sections to make them easier for you to manage. They vary in level of difficulty, including short and long programming problems (marked with "programming-required" icons—one icon to indicate short exercises and two icons for projects), the analysis of algorithms, and problems to test students' understanding of abstract concepts. In this edition we have streamlined the previous exercises, allowing us to add even more options for you to choose from. In particular we have added several larger programming exercises to many of the chapters.

Input/Output Options It is difficult to know what background the students using a data structures text will have in Java I/O. To allow all the students using our text to concentrate on the primary topic of data structures, we use the simplest I/O approach we can, namely a command line interface. However, to support those teachers and students who prefer to work with graphical user interfaces (GUIs), we provide GUIs for many of our applications. Our modular approach to program design supports this approach—our applications separate the user interface code, problem solution code, and ADT implementation code into separate classes.

Concurrency Coverage We are pleased to be one of the only data structures texts to address the topics of concurrency and synchronization, which are growing in importance as computer systems move to using more cores and threads to obtain additional performance with each new generation. We introduce this topic in Section 4.9, “Concurrency, Interference, and Synchronization,” where we start with the basics of Java threads, continue through examples of thread interference and synchronization, and culminate in a discussion of efficiency concerns.

New to the *Fourth Edition*

This edition represents a major revision of the text’s material, although the philosophy and style that our loyal adopters have grown to appreciate remain unchanged. We removed material we felt was redundant or of lesser/outdated importance to the core topic of data structures, added new key material, and reworked much of the material that we kept. Although the length of the textbook was reduced by about 10%, the coverage of data structures has been expanded. We believe this new edition is a great improvement over previous editions and hope you do, too. Major changes include:

- Simplified Architecture: We continue to use the Java interface construct to define the abstract view of our ADTs, but we have reduced the number of levels of inheritance, simplifying the architecture and making it easier to understand and use.
- New Chapters: Chapter 5, “The Collection ADT,” and Chapter 8, “The Map ADT,” are brand new. The Collection ADT material introduces the idea of a data structure as a repository and concentrates on storage and retrieval of data based on key attributes. The Map ADT has become increasingly important with the rise in popularity of scripting languages with built-in associative arrays.
- New Section: Section 1.6, “Comparing Algorithms: Order of Growth Analysis,” was completely rewritten and features an introduction to efficiency analysis driven by a game played between two students, plus analysis of sequential search, binary search, and sequential sort algorithms.
- New Sections: In response to reader’s suggestions, Chapter 3, “Recursion,” features two new sections: Section 3.3, “Recursive Processing of Arrays,” is devoted to recursive processing of arrays and Section 3.4, “Recursive Processing of Linked Lists,” is devoted to recursive processing of linked lists. These new sections provide practical examples of the use of recursion, before the reader moves on to the less practical but nevertheless popular Towers of Hanoi example covered in Section 3.5, “Towers.”
- New Section: Fractals! A fun section related to recursively generating fractal-based images now wraps up the examples of Chapter 3, “Recursion.”

- New Sections: We added "Variations" sections to the Stack, Queue, Collection, List, Tree, and Map chapters. In the primary exposition of each of these ADTs we record design decisions and specify the operations to be supported by the ADT. We also develop or at least discuss various implementation approaches, in most cases highlighting one array-based approach and one reference/linked-list-based approach. The "Variations" section discusses alternate approaches to defining/implementing the ADT and in most cases reviews the ADT counterparts available in the standard Java Library. Some of these sections also introduce related ADTs, for example, in the "Variations" section of the Collection chapter we define and discuss both the Set and Bag ADTs.
- Glossary: The text's glossary has always been available online. With this edition we make it available as Appendix E. Throughout the text we highlight important terms that might be unfamiliar to the student in **green**, the first time they are featured, to indicate that their definition can be found in the glossary.

Prerequisite Assumptions

In this book, we assume that readers are familiar with the following Java constructs:

- Built-in simple data types and the array type
- Control structures *while*, *do*, *for*, *if*, and *switch*
- Creating and instantiating objects
- Basic user-defined classes:
  - variables and methods
  - constructors, method parameters, and the *return* statement
  - visibility modifiers
- Commonly used Java Library Classes: *Integer, Math, Random, Scanner*, *String*, and *System*

Chapter Content

Chapter 1 is all about **Getting Organized**. An overview of object orientation stresses mechanisms for organizing objects and classes. The Java exception handling mechanisms, used to organize response to unusual situations, are introduced. Data structures are previewed and the two fundamental language constructs that are used to implement those structures, the array and the reference (link/pointer), are discussed. The chapter concludes with a look at efficiency analysis—how we evaluate and compare algorithms.

Chapter 2 presents **The Stack ADT**. The concept of abstract data type (ADT) is introduced. The stack is viewed from three different levels: the abstract, application, and implementation levels. The Java interface mechanism is used to support this three-tiered view. We also investigate using generics to support generally usable ADTs. The Stack ADT is implemented using both arrays and references. To support the reference-based approach we introduce the linked list structure. Sample applications include determining if a set of grouping symbols is well formed and the evaluation of postfix expressions.

Chapter 3 discusses **Recursion**, showing how recursion can be used to solve programming problems. A simple three-question technique is introduced for verifying the correctness of recursive methods. Sample applications include array processing, linked list processing, the classic Towers of Hanoi, and fractal generation. A detailed discussion of how recursion works shows how recursion can be replaced with iteration and stacks.

Chapter 4 presents **The Queue ADT**. It is also first considered from its abstract perspective, followed by a formal specification, and then implemented using both array-based and reference-based approaches. Example applications include an interactive test driver, a palindrome checker, and simulating a system of real-world queues. Finally, we look at Java's concurrency and synchronization mechanisms, explaining issues of interference and efficiency.

Chapter 5 defines **The Collection ADT**. A fundamental ADT, the Collection, supports storing information and then retrieving it later based on its content. Approaches for comparing objects for equality and order are reviewed. Collection implementations using an array, a sorted array, and a linked list are developed. A text processing application permits comparison of the implementation approaches for efficiency. The "Variations" section introduces two more well-known ADTs: the Bag and the Set.

Chapter 6 follows up with a more specific Collection ADT, **The List ADT**. In fact, the following two chapters also develop Collection ADTs. Iteration is introduced here and the use of anonymous inner classes to provide iterators is presented. As with the Collection ADT we develop array, sorted array, and linked-list–based implementations. The "Variations" section includes an example of how to "implement" a linked list within an array. Applications include a card deck model plus some card games, and a Big Integer class. This latter application demonstrates how we sometimes design specialized ADTs for specific problems.

Chapter 7 develops **The Binary Search Tree ADT**. It requires most of the chapter just to design and create our reference-based implementation of this relatively complex structure. The chapter also discusses trees in general (including breadth-first and depth-first searching) and the problem of balancing a binary search tree. A wide variety of special-purpose and self-balancing trees are introduced in the "Variations" section.

Chapter 8 presents **The Map ADT**, also known as a symbol table, dictionary, or associative array. Two implementations are developed, one that uses an *ArrayList* and the other that uses a hash table. A large part of the chapter is devoted to this latter implementation and the important concept of hashing, which provides a very efficient implementation of a Map. The "Variations" section discusses a map-based hybrid data structure plus Java's support for hashing.

Chapter 9 introduces **The Priority Queue ADT**, which is closely related to the Queue but with a different accessing protocol. This short chapter does present a sorted array-based implementation, but most of the chapter focuses on a clever, interesting, and very efficient implementation called a Heap.

Chapter 10 covers **The Graph ADT**, including implementation approaches and several important graph-related algorithms (depth-first search, breadth-first search, path existence, shortest paths, and connected components). The graph algorithms make use of stacks, queues, and priority queues, thus both reinforcing earlier material and demonstrating the general usability of these structures.

Chapter 11 presents/reviews a number of **Sorting and Searching Algorithms.** The sorting algorithms that are illustrated, implemented, and compared include straight selection sort, two versions of bubble sort, insertion sort, quick sort, heap sort, and merge sort. The sorting algorithms are compared using efficiency analysis. The discussion of algorithm analysis continues in the context of searching. Previously presented searching algorithms are reviewed and new ones are described.

Organization

Chapter Goals Sets of knowledge and skill goals are presented at the beginning of each chapter to help the students assess what they have learned.

Sample Programs Numerous sample programs and program segments illustrate the abstract concepts throughout the text.

Feature Sections Throughout the text these short sections highlight topics that are not directly part of the flow of material but nevertheless are related and important.

Boxed Notes These small boxes of information scattered throughout the text highlight, supplement, and reinforce the text material, perhaps from a slightly different point of view.

Chapter Summaries Each chapter concludes with a summary section that reviews the most important topics of the chapter and ties together related topics. Some chapter summaries include a UML diagram of the major interfaces and classes developed within the chapter.

Appendices The appendices summarize the Java reserved word set, operator precedence, primitive data types, the ASCII subset of Unicode, and provide a glossary of important terms used in the text.

Website http://go.jblearning.com/oods4e

This website provides access to the text's source code files for each chapter. Additionally, registered instructors are able to access selected answers to the text's exercises, a test item file, and presentation slides. Please contact the authors if you have material related to the text that you would like to share with others.

Acknowledgments

We would like to thank the following people who took the time to review this text: Mark Llewellyn at the University of Central Florida, Chenglie Hu at Carroll College, Val Tannen at the University of Pennsylvania, Chris Dovolis at the University of Minnesota, Mike Coe at Plano Senior High School, Mikel Petty at University of Alabama in Huntsville, Gene Sheppard at Georgia Perimeter College, Noni Bohonak at the University of South Carolina–Lancaster, Jose Cordova at the University of Louisiana–Monroe, Judy Gurka at the Metropolitan State College of Denver, Mikhail Brikman at Salem State University, Amitava Karmaker at University of Wisconsin–Stout, Guifeng Shao at Tennessee State University, Urska Cvek at Louisiana State University at Shreveport, Philip C. Doughty Jr. at Northern Virginia Community College, Jeff Kimball at Southwest Baptist University, Jeremy T. Lanman at Nova Southeastern University, Rao Li at University of South Carolina Aiken, Larry Thomas at University of Toledo, and Karen Works at Westfield State University. A special thanks to Christine Shannon at Centre College, to Phil LaMastra at Fairfield University, to Allan Gottlieb of New York University, and to J. William Cupp at Indiana Wesleyan University for specific comments leading to improvements in the text. A personal thanks to Kristen Obermyer, Tara Srihara, Sean Wilson, Christopher Lezny, and Naga Lakshmi, all of Villanova University, plus Kathy, Tom, and Julie Joyce for all of their help, support, and proofreading expertise.

A virtual bouquet of roses to the editorial and production teams who contributed so much, especially Laura Pagluica, Taylor Ferracane, Amy Rose, and Palaniappan Meyyappan.

ND
DJ
CW

Contents

© Ake13bk/Shutterstock

CHAPTER 1

Getting Organized

Knowledge Goals

You should be able to

- describe some benefits of object-oriented programming
- describe the genesis of the Unified Method
- explain the relationships among classes, objects, and applications
- explain how method calls are bound to method implementations with respect to inheritance
- describe, at an abstract level, the following structures: array, linked list, stack, queue, list, tree, map, and graph
- identify which structures are implementation dependent and which are implementation independent
- describe the difference between direct addressing and indirect addressing
- explain the subtle ramifications of using references/pointers
- explain the use of O notation to describe the amount of work done by an algorithm
- describe the sequential search, binary search, and selection sort algorithms

Skill Goals

You should be able to

- interpret a basic UML class diagram
- design and implement a Java class
- create a Java application that uses the Java class
- use packages to organize Java compilation units
- create a Java exception class
- throw Java exceptions from within a class and catch them within an application that uses the class
- predict the output of short segments of Java code that exhibit aliasing
- declare, initialize, and use one- and two-dimensional arrays in Java, including both arrays of a primitive type and arrays of objects
- given an algorithm, identify an appropriate size representation and determine its order of growth
- given a section of code determine its order of growth

Before embarking on any new project, it is a good idea to prepare carefully—to "get organized." In this first chapter that is exactly what we do. A careful study of the topics of this chapter will prepare us for the material on data structures and algorithms, using the object-oriented approach, covered in the remainder of the book.

1.1 Classes, Objects, and Applications

Software design is an interesting, challenging, and rewarding task. As a beginning student of computer science, you wrote programs that solved relatively simple problems. Much of your effort went into learning the syntax of a programming language such as Java: the language's reserved words, its data types, its constructs for selection and looping, and its input/output mechanisms.

As your programs and the problems they solve become more complex it is important to follow a software design approach that modularizes your solutions—breaks them into coherent manageable subunits. Software design was originally driven by an emphasis on actions. Programs were modularized by breaking them into subprograms or procedures/functions. A subprogram performs some calculations and returns information to the calling program, but it does not "remember" anything. In the late 1960s, researchers argued that this approach was too limiting and did not allow us to successfully represent the constructs needed to build complex systems.

Two Norwegians, Kristen Nygaard and Ole-Johan Dahl, created Simula 67 in 1967. It was the first language to support object-oriented programming. Object-oriented languages promote the object as the prime modularization mechanism. Objects represent both information and behavior and can "remember" internal information from one use to the next. This crucial difference allows them to be used in many versatile ways. In 2001, Nygaard and Dahl received the Turing Award, sometimes referred to as the "Nobel Prize of Computing," for their work.

The capability of objects to represent both information (the objects have *attributes*) and behavior (the objects have *responsibilities*) allows them to be used to represent "real-world" entities as varied as bank accounts, genomes, and hobbits. The self-contained nature of objects makes them easy to implement, modify, and test for correctness.

Object orientation is centered on classes and objects. Objects are the basic run-time entities used by applications. An object is an instantiation of a class; alternatively, a class defines the structure of its objects. In this section we review these object-oriented programming constructs that we use to organize our programs.

Classes

A class defines the structure of an object or a set of objects. A class definition includes variables (data) and methods (actions) that determine the behavior of an object. The following Java code defines a `Date` class that can be used to create and manipulate `Date` objects—for example, within a school course-scheduling application. The `Date` class can be used to create `Date` objects and to learn about the year, month, or day of any particular

Date object.[1] The class also provides methods that return the Lilian Day Number of the date (the code details have been omitted—see the feature section on Lilian Day Numbers for more information) and return a string representation of the date.

Authors' Convention

Java-reserved words (when used as such), user-defined identifiers, class and file names, and so on, appear in `this font` throughout the entire text.

```
//----------------------------------------------------------------------
// Date.java               by Dale/Joyce/Weems                 Chapter 1
//
// Defines date objects with year, month, and day attributes.
//----------------------------------------------------------------------
package ch01.dates;
public class Date
{
    protected int year, month, day;
    public static final int MINYEAR = 1583;

    // Constructor
    public Date(int newMonth, int newDay, int newYear)
    {
       month = newMonth; day = newDay; year = newYear;
    }

    // Observers
    public int getYear() { return year; }
    public int getMonth() { return month; }
    public int getDay(){ return day; }

    public int lilian()
    {
       // Returns the Lilian Day Number of this date.
       // Algorithm goes here. Code is included with the program files.
       // See Lilian Day Numbers feature section for details.
    }

    @Override[2]
    public String toString()
```

[1] The Java library includes a `Date` class, `java.util.Date`. However, the familiar properties of dates make them a natural example to use in explaining object-oriented concepts. Here we ignore the existence of the library class, as if we must design our own `Date` class.

[2] The purpose of `@Override` is discussed in Section 1.2 "Organizing Classes."

```
      // Returns this date as a String.
      {
         return(month + "/" + day + "/" + year);
      }
}
```

The `Date` class demonstrates two kinds of variables: instance variables and class variables. The instance variables of this class are `year`, `month`, and `day` declared as

```
protected int year, month, day;
```

Their values vary for each "instance" of an object of the class. Instance variables provide the internal representation of an object's attributes.

The variable `MINYEAR` is declared as

```
public static final int MINYEAR = 1583;
```

`MINYEAR` is defined as being `static`, and thus it is a class variable. It is associated directly with the `Date` class, instead of with objects of the class. A single copy of a class variable is maintained for all objects of the class.

Remember that the `final` modifier states that a variable is in its final form and cannot be modified; thus `MINYEAR` is a constant. By convention, we use only capital letters when naming constants. It is standard procedure to declare constants as class variables. Because the value of the variable cannot change, there is no need to force every object of a class to carry around its own version of the value. In addition to holding shared constants, class variables can be used to maintain information that is common to an entire class. For example, a `BankAccount` class may have a class variable that holds the number of current accounts.

In the `Date` class example, the `MINYEAR` constant represents the first full year that the widely used Gregorian calendar was in effect. The idea here is that programmers should not use the class to represent dates that predate that year. We look at ways to enforce this rule in Section 1.3 "Exceptional Situations," where we discuss handling exceptional situations.

> **Authors' Convention**
>
> We highlight important terms that might be unfamiliar to the student in **green**, the first time they are featured, to indicate that their definition can be found in the glossary in Appendix E.

The methods of the class are `Date`, `getYear`, `getMonth`, `getDay`, `lilian`, and `toString`. Note that the `Date` method has the same name as the class. Recall that this means it is a special type of method, called a class **constructor**. Constructors are used to create new instances of a class—that is, to instantiate objects of a class. The other methods are classified as **observer** methods, because they "observe" and return information based on the instance variable values. Other names for observer methods are "accessor" methods and "getters," as in accessing or getting information. Methods that simply return the value of an instance variable, such as `getYear()` in our `Date` class, are very common and always follow the same code pattern consisting of a single `return` statement. For this reason we will format such methods as a single line of code. In addition to constructors

Table 1.1 Java Access Control Modifiers

	Access Is Allowed			
	Within the Class	Within the Package	Within Subclasses	Everywhere
`public`	X	X	X	X
`protected`	X	X	X	
`package`	X	X		
`private`	X			

and observers, there is another general category of method, called a **transformer**. As you probably recall, transformers change the object in some way; for example, a method that changes the year of a `Date` object would be classified as a transformer.

You have undoubtedly noticed the use of the access modifiers `protected` and `public` within the `Date` class. Let us review the purpose and use of **access modifiers**. This discussion assumes you recall the basic ideas behind inheritance and packages. Inheritance supports the extension of one class, called the superclass, by another class, called the subclass. The subclass "inherits" properties (data and actions) from the superclass. We say that the subclass is derived from the superclass. Packages let us group related classes together into a single unit. Inheritance and packages are both discussed more extensively in the next section.

Java allows a wide spectrum of access control, as summarized in **Table 1.1**. The `public` access modifier used with the methods of `Date` makes them "publicly" available; any code that can "see" an object of the class can use its public methods. We say that these methods are "exported" from the class. Additionally, any class that is derived from the `Date` class using inheritance inherits its public methods and variables.

Public access sits at one end of the access spectrum, allowing open access. At the other end of the spectrum is private access. When you declare a class's variables and methods as `private`, they can be used only inside the class itself and are not inherited by subclasses. You should routinely use private (or protected) access within your classes to hide their data. You do not want the data values to be changed by code that is outside the class. For example, if the `month` instance variable in our `Date` class was declared to be `public`, then the application code could directly set the value of a `Date` object's `month` to strange numbers such as −12 or 27.

An exception to this guideline of hiding data within a class is shown in the `Date` example. Notice that the `MINYEAR` constant is publicly accessible. It can be accessed directly by the application code. For example, an application could include the statement

```
if (myYear < Date.MINYEAR) ...
```

Because `MINYEAR` is a final constant, its value cannot be changed by the application. Thus, even though it is publicly accessible, no other code can change its value. It is not necessary to hide it. The application code above also shows how to access a public class variable from outside the class. Because `MINYEAR` is a class variable, it is accessed through the class name, `Date`, rather than through an object of the class.

Private access affords the strongest protection. Access is allowed only within the class. However, if you plan to extend your classes using inheritance, you may want to use protected access instead.

Coding Convention

We use protected access extensively for instance variables within our classes in this text.

The `protected` access modifier used in `Date` provides visibility similar to private access, only slightly less rigid. It "protects" its data from outside access, but allows the data to be accessed from within its own package *or* from any class derived from its class. Therefore, the methods within the `Date` class can access `year`, `month`, and `day`, and if, as we will show in Section 1.2 "Organizing Classes," the `Date` class is extended, the methods in the extended class can also access those variables.

The remaining type of access is called package access. A variable or method of a class defaults to package access if none of the other three modifiers are used. Package access means that the variable or method is accessible to any other class in the same package.

Lilian Day Numbers

Various approaches to numbering days have been proposed. Most choose a particular day in history as day 1, and then number the actual sequence of days from that day forward with the numbers 2, 3, and so on. The Lilian Day Number (LDN) system uses October 15, 1582, as day 1, or LDN 1.

Our current calendar is called the Gregorian calendar. It was established in 1582 by Pope Gregory XIII. At that time 10 days were dropped from the month of October, to make up for small errors that had accumulated throughout the years. Thus, the day following October 4, 1582, in the Gregorian calendar is October 15, 1582, also known as LDN 1 in the Lilian day numbering scheme. The scheme is named after Aloysius Lilius, an advisor to Pope Gregory and one of the principal instigators of the calendar reform.

Originally, Catholic European countries adopted the Gregorian calendar. Many Protestant nations, such as England and its colonies, did not adopt the Gregorian calendar until 1752, at which

1582		OCTOBER			1582	
SUN	MON	TUE	WED	THU	FRI	SAT
	1	2	3	4	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

time they also "lost" 11 days. Today, most countries use the Gregorian calendar, at least for official international business. When comparing historical dates, one must be careful about which calendars are being used.

In our `Date` class implementation, `MINYEAR` is 1583, representing the first full year during which the Gregorian calendar was in operation. We assume that programmers will not use the `Date` class to represent dates before that time, although this rule is not enforced by the class. This assumption simplifies calculation of day numbers, as we do not have to worry about the phantom 10 days of October 1582.

To calculate LDNs, one must understand how the Gregorian calendar works. Years are usually 365 days long. However, every year evenly divisible by 4 is a leap year, 366 days long. This aligns the calendar closer to astronomical reality. To fine-tune the adjustment, if a year is evenly divisible by 100, it is not a leap year but, if it is also evenly divisible by 400, it is a leap year. Thus 2000 was a leap year, but 1900 was not.

Given a date, the `lilian` method of the `Date` class counts the number of days between that date and the hypothetical date 1/1/0—that is, January 1 of the year 0. This count is made under the assumption that the Gregorian reforms were in place during that entire time period. In other words, it uses the rules described in the previous paragraph. Let us call this number the Relative Day Number (RDN). To transform a given RDN to its corresponding LDN, we just need to subtract the RDN of October 14, 1582, from it. For example, to calculate the LDN of July 4, 1776, the method first calculates its RDN (648,856) and then subtracts from it the RDN of October 14, 1582 (578,100), giving the result of 70,756.

Code for the `lilian` method is included with the program code files.

The Unified Method

The object-oriented approach to programming is based on implementing models of reality. But how do you go about this? Where do you start? How do you proceed? The best plan is to follow an organized approach called a **methodology**.

In the late 1980s, many people proposed object-oriented methodologies. By the mid-1990s, three proposals stood out: the Object Modeling Technique, the Objectory Process, and the Booch Method. Between 1994 and 1997, the primary authors of these proposals got together and consolidated their ideas. The resulting methodology was dubbed the Unified Method. It is now, by far, the most popular organized approach to creating object-oriented systems.

The Unified Method features three key elements:

1. It is use-case driven. A use-case is a description of a sequence of actions performed by a user within the system to accomplish some task. The term "user" here should be interpreted in a broad sense and could represent another system.
2. It is architecture-centric. The word "architecture" refers to the overall structure of the target system, the way in which its components interact.

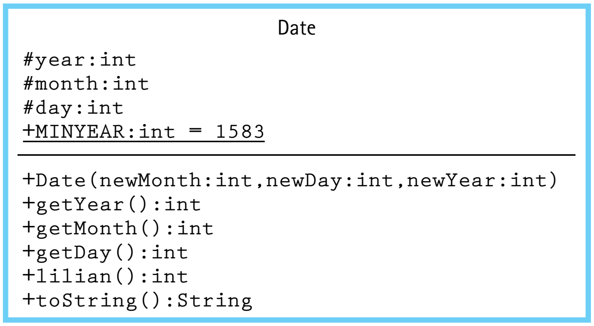


Figure 1.1 UML class diagram for the `Date` class

3. It is iterative and incremental. The Unified Method involves a series of development cycles, with each one building upon the foundation established by its predecessors.

One of the main benefits of the Unified Method is improved communication among the people involved in the project. The Unified Method includes a set of diagrams for this purpose, called the **Unified Modeling Language (UML)**.[3] UML diagrams have become a de facto industry standard for modeling software. They are used to specify, visualize, construct, and document the components of a software system. We use UML class diagrams throughout this text to model our classes and their interrelationships.

A diagram representing the `Date` class is shown in **Figure 1.1**. The diagram follows the standard UML class notation approach. The name of the class appears in the top section of the diagram, the variables (attributes) appear in the next section, and the methods (operations) appear in the final section. The diagram includes information about the nature of the variables and method parameters; for example, we can see at a glance that `year`, `month`, and `day` are all of type `int`. Note that the variable `MINYEAR` is underlined; this indicates that it is a class variable rather than an instance variable. The diagram also indicates the visibility or protection associated with each part of the class (+ = public, # = protected).

Objects

Objects are created from classes at run time. They can contain and manipulate data. Multiple objects can be created from the same class definition. Once a class such as `Date` has been defined, a program can create and use objects of that class. The effect is similar to expanding the language's set of standard types to include a `Date` type. To create an object in Java we use the `new` operator, along with the class constructor, as follows:

```
Date myDate = new Date(6, 24, 1951);
Date yourDate = new Date(10, 11, 1953);
Date ourDate = new Date(6, 15, 1985);
```

[3] The official definition of the UML is maintained by the Object Management Group. Detailed information can be found at http://www.uml.org/.

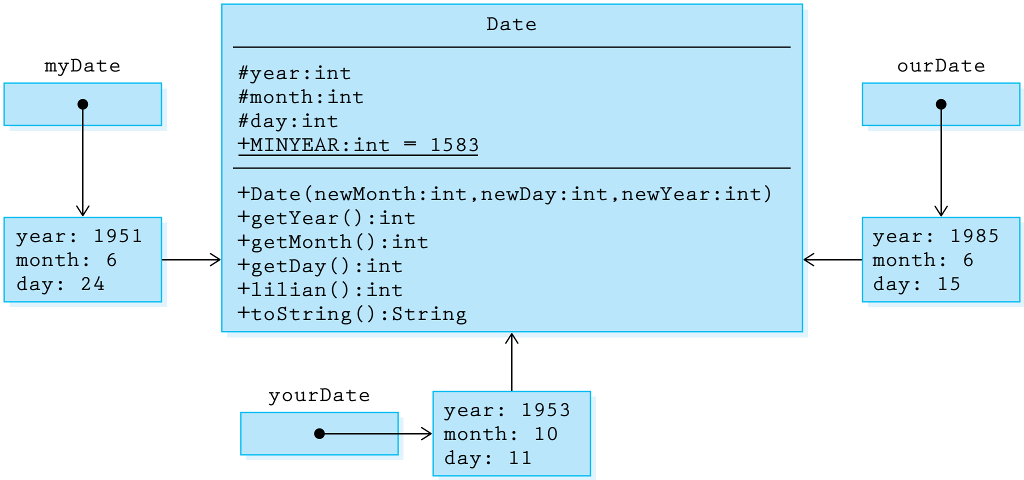


Figure 1.2 Class diagram showing `Date` objects

We say that the variables `myDate`, `yourDate`, and `ourDate` reference "objects of the class `Date`" or simply "objects of type `Date`." We could also refer to them as "`Date` objects."

Figure 1.2 extends our previous diagram (shown in Figure 1.1) to show the relationship between the instantiated `Date` objects and the `Date` class. As you can see, the objects are associated with the class, as represented by arrows from the objects to the class in the diagram. Notice that the `myDate`, `yourDate`, and `ourDate` variables are not objects, but actually hold references to the objects. The references are shown by the arrows from the variable boxes to the objects. In reality, references are memory addresses. The memory address of the instantiated object is stored in the memory location assigned to the variable. If no object has been instantiated for a particular variable, then its memory location holds a `null` reference.

Methods are invoked through the object upon which they are to act. For example, to assign the return value of the `getYear` method of the `ourDate` object to the integer variable `theYear`, a programmer would code

```
theYear = ourDate.getYear();
```

Recall that the `toString` method is invoked in a special way. Just as Java automatically changes an integer value, such as that returned by `getDay`, to a string in the statement

```
System.out.println("The big day is " + ourDate.getDay());
```

it automatically changes an object, such as `ourDate`, to a string in the statement

```
System.out.println("The party will be on " + ourDate);
```

The output from these statements would be

The big day is 15
The party will be on 6/15/1985