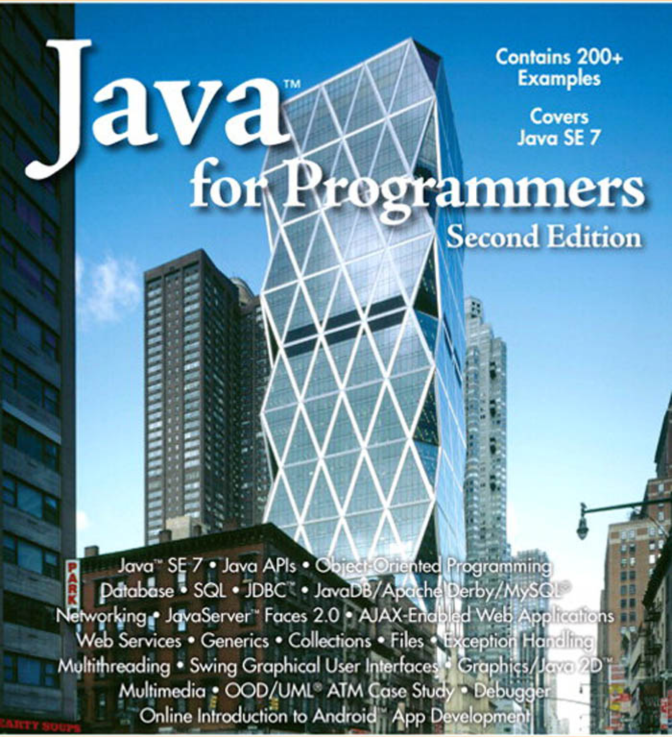
DEITEL® DEVELOPER SERIES



Java™ for Programmers

Second Edition

Contains 200+ Examples

Covers Java SE 7

Java™ SE 7 • Java APIs • Object-Oriented Programming
Database • SQL • JDBC™ • JavaDB/Apache Derby/MySQL®
Networking • JavaServer™ Faces 2.0 • AJAX-Enabled Web Applications
Web Services • Generics • Collections • Files • Exception Handling
Multithreading • Swing Graphical User Interfaces • Graphics/Java 2D™
Multimedia • OOD/UML® ATM Case Study • Debugger
Online Introduction to Android™ App Development

PAUL DEITEL • HARVEY DEITEL

Java™ for Programmers
Second Edition
Deitel® Developer Series

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U. S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the U. S., please contact:

International Sales
international@pearsoned.com

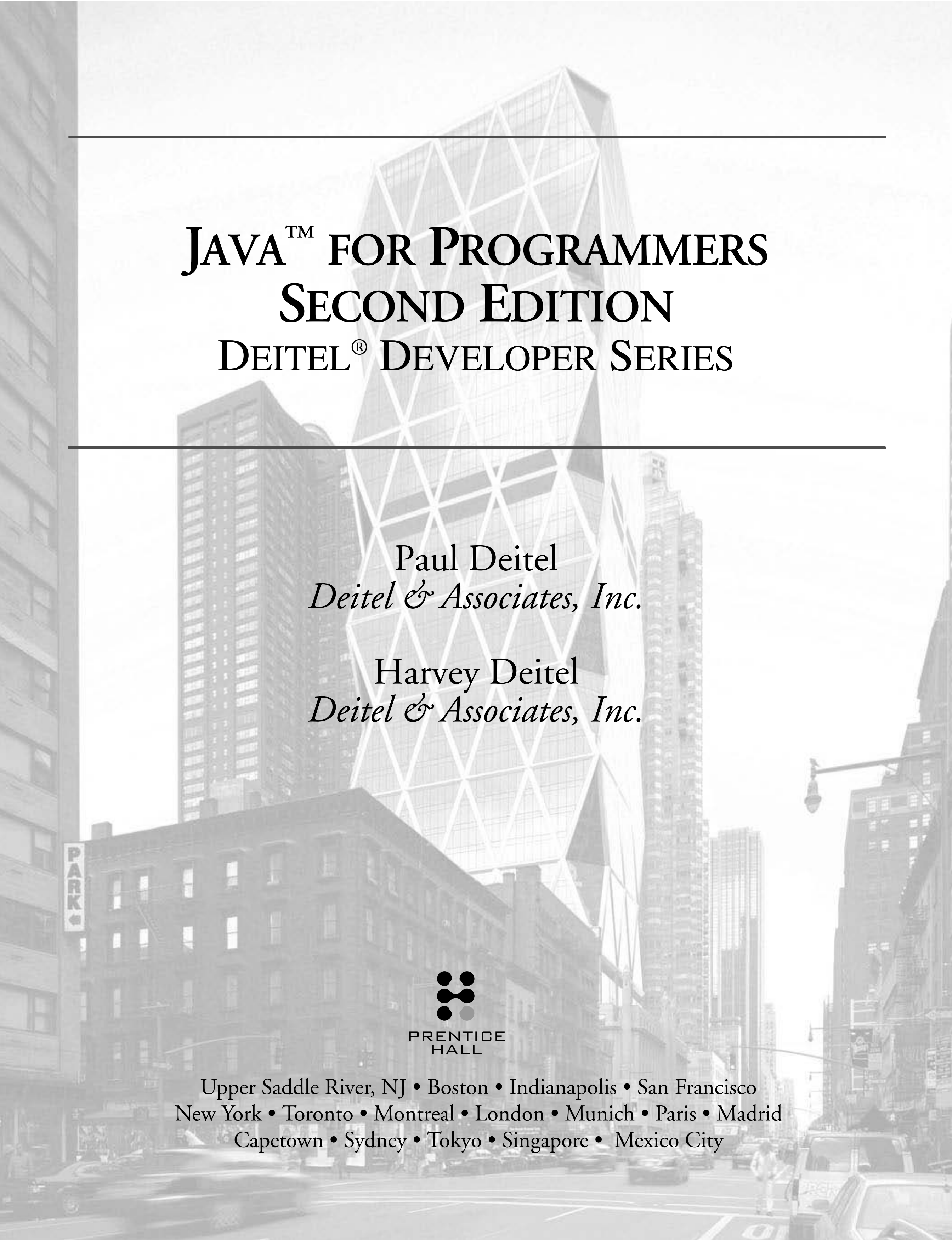
Visit us on the Web: informit.com/ph

Library of Congress Cataloging-in-Publication Data

On file

ISBN-13: 978-0-13282154-4
ISBN-10: 0-13-282154-0

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, March 2012



Java™ for Programmers Second Edition

Deitel® Developer Series

Paul Deitel
Deitel & Associates, Inc.

Harvey Deitel
Deitel & Associates, Inc.

PRENTICE HALL

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Trademarks

DEITEL, the double-thumbs-up bug and DIVE INTO are registered trademarks of Deitel and Associates, Inc.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Microsoft, Internet Explorer and the Windows logo are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

UNIX is a registered trademark of The Open Group.

Apache is a trademark of The Apache Software Foundation.

CSS, XHTML and XML are registered trademarks of the World Wide Web Consortium.

Firefox is a registered trademark of the Mozilla Foundation.

Google is a trademark of Google, Inc.

Web 2.0 is a service mark of CMP Media.

Throughout this book, trademarks are used. Rather than put a trademark symbol in every occurrence of a trademarked name, we state that we are using the names in an editorial fashion only and to the benefit of the trademark owner, with no intention of infringement of the trademark.

In memory of Clifford "Spike" Stephens,
A dear friend who will be greatly missed.

Paul and Harvey Deitel

This page intentionally left blank

Contents

This page intentionally left blank

Preface

Live in fragments no longer, only connect.
—Edgar Morgan Foster

Welcome to Java and *Java for Programmers, Second Edition*! This book presents leading-edge computing technologies for software developers.

We focus on software engineering best practices. At the heart of the book is the Deitel signature "live-code approach"—concepts are presented in the context of complete working programs, rather than in code snippets. Each complete code example is accompanied by live sample executions. All the source code is available at

```
www.deitel.com/books/javafp2/
```

As you read the book, if you have questions, send an e-mail to `deitel@deitel.com`; we'll respond promptly. For updates on this book, visit the website shown above, follow us on Facebook (`www.facebook.com/DeitelFan`) and Twitter (`@deitel`), and subscribe to the *Deitel® Buzz Online* newsletter (`www.deitel.com/newsletter/subscribe.html`).

Features

Here are the key features of *Java for Programmers, 2/e*:

Java Standard Edition (SE) 7

- ***Easy to use as a Java SE 6 or Java SE 7 book.*** We cover the new Java SE 7 features in modular sections. Here's some of the new functionality: `String`s in `switch` statements, the `try`-with-resources statement for managing `AutoClosable` objects, multi-`catch` for defining a single exception handler to replace multiple exception handlers that perform the same task and inferring the types of generic objects from the variable they're assigned to by using the <> notation. We also overview the new concurrency API features.
- ***Java SE 7's `AutoClosable` versions of `Connection`, `Statement` and `ResultSet`.*** With the source code for Chapter 25, Accessing Databases with JDBC, we provide a version of the chapter's first example that's implemented using Java SE 7's `AutoClosable` versions of `Connection`, `Statement` and `ResultSet`. `AutoClosable` objects reduce the likelihood of resource leaks when you use them with Java SE 7's `try`-with-resources statement, which automatically closes the `AutoClosable` objects allocated in the parentheses following the `try` keyword.

Object Technology

- ***Object-oriented programming and design.*** We review the basic concepts and terminology of object technology in Chapter 1. Readers develop their first customized classes and objects in Chapter 3.

- ***Exception handling.*** We integrate basic exception handling early in the book and cover it in detail in Chapter 11, Exception Handling: A Deeper Look.
- ***Class `Arrays` and `ArrayList`.*** Chapter 7 covers class `Arrays`—which contains methods for performing common array manipulations—and class `ArrayList`—which implements a dynamically resizable array-like data structure.
- ***OO case studies.*** The early classes and objects presentation features `Time`, `Employee` and `GradeBook` class case studies that weave their way through multiple sections and chapters, gradually introducing deeper OO concepts.
- ***Case Study: Using the UML to Develop an Object-Oriented Design and Java Implementation of an ATM.*** The UML™ (Unified Modeling Language™) is the industry-standard graphical language for modeling object-oriented systems. Chapters 12–13 include a case study on object-oriented design using the UML. We design and implement the software for a simple automated teller machine (ATM). We analyze a typical requirements document that specifies the system to be built. We determine the classes needed to implement that system, the attributes the classes need to have, the behaviors the classes need to exhibit and specify how the classes must interact with one another to meet the system requirements. From the design we produce a *complete* Java implementation. Readers often report having a "light-bulb moment"—the case study helps them "tie it all together" and really understand object orientation in Java.
- ***Reordered generics presentation.*** We begin with generic class `ArrayList` in Chapter 7. Because *you'll understand basic generics concepts early in the book*, our later data structures discussions provide a deeper treatment of generic collections—showing how to use the built-in collections of the Java API. We then show how to implement generic methods and classes.

Database and Web Development

- ***JDBC 4.*** Chapter 25, Accessing Databases with JDBC, covers JDBC 4 and uses the Java DB/Apache Derby and MySQL database management systems. The chapter features an OO case study on developing a database-driven address book that demonstrates prepared statements and JDBC 4's automatic driver discovery.
- ***Java Server Faces (JSF) 2.0.*** Chapters 26–27 have been updated with JavaServer Faces (JSF) 2.0 technology, which greatly simplifies building JSF web applications. Chapter 26 includes examples on building web application GUIs, validating forms and session tracking. Chapter 27 discusses data-driven and Ajax-enabled JSF applications. The chapter features a database-driven multitier web address book that allows users to add and search for contacts.
- ***Web services.*** Chapter 28, Web Services, demonstrates creating and consuming SOAP- *and* REST-based web services. Case studies include developing blackjack and airline reservation web services.
- ***Java Web Start and the Java Network Launch Protocol (JNLP).*** We introduce Java Web Start and JNLP, which enable applets *and* applications to be launched via a web browser. Users can install locally for later execution. Programs can also request the user's permission to access local system resources such as files—en-

abling you to develop more robust applets and applications that execute safely using Java's sandbox security model, which applies to downloaded code.

Multithreading

- ***Multithreading.*** We completely reworked Chapter 23, Multithreading [special thanks to the guidance of Brian Goetz and Joseph Bowbeer—two of the co-authors of *Java Concurrency in Practice,* Addison-Wesley, 2006].
- ***`SwingWorker` class.*** We use class `SwingWorker` to create *multithreaded user interfaces*.

GUI and Graphics

- ***GUI and graphics presentation.*** Chapters 14, 15 and 22, and Appendix H present Java GUI and Graphics programming.
- ***`GroupLayout` layout manager.*** We discuss the `GroupLayout` layout manager in the context of the GUI design tool in the NetBeans IDE.
- ***`JTable` sorting and filtering capabilities.*** Chapter 25 uses these capabilities to sort the data in a `JTable` and filter it by regular expressions.

Other Features

- ***Android.*** Because of the tremendous interest in Android-based smartphones and tablets, we've included a three-chapter introduction to Android app development online at `www.deitel.com/books/javafp`. These chapters are from our new Deitel Developer Series book *Android for Programmers: An App-Driven Approach*. After you learn Java, you'll find it straightforward to develop and run Android apps on the free Android emulator that you can download from `developer.android.com`.
- ***Software engineering community concepts***. We discuss agile software development, refactoring, design patterns, LAMP, SaaS (Software as a Service), PaaS (Platform as a Service), cloud computing, open-source software and more.

Teaching Approach

Java for Programmers, 2/e, contains hundreds of complete working examples. We stress program clarity and concentrate on building well-engineered software.

Syntax Shading. For readability, we syntax shade the code, similar to the way most integrated-development environments and code editors syntax color the code. Our syntax-shading conventions are:

```
comments appear like this
keywords appear like this
constants and literal values appear like this
all other code appears in black
```

Code Highlighting. We place gray rectangles around each program's key code.

Using Fonts for Emphasis. We place the key terms and the index's page reference for each defining occurrence in **bold** text for easier reference. On-screen components are emphasized in the **bold Helvetica** font (e.g., the **File** menu) and Java program text in the `Lucida` font (e.g., `int x = 5;`).

Web Access. All of the source-code examples can be downloaded from:

```
www.deitel.com/books/javafp2
www.pearsonhighered.com/deitel
```

Objectives. The chapter opening quotations are followed by a list of chapter objectives.

Illustrations/Figures. Abundant tables, line drawings, UML diagrams, programs and program outputs are included.

Programming Tips. We include programming tips to help you focus on important aspects of program development. These tips and practices represent the best we've gleaned from a combined eight decades of programming and teaching experience.

Good Programming Practice

The Good Programming Practices *call attention to techniques that will help you produce programs that are clearer, more understandable and more maintainable.*

Common Programming Error

Pointing out these Common Programming Errors *reduces the likelihood that you'll make the same errors.*

Error-Prevention Tip

These tips contain suggestions for exposing and removing bugs from your programs; many of the tips describe aspects of Java that prevent bugs from getting into programs.

Performance Tip

These tips highlight opportunities for making your programs run faster or minimizing the amount of memory that they occupy.

Portability Tip

The Portability Tips *help you write code that will run on a variety of platforms.*

Software Engineering Observation

The Software Engineering Observations *highlight architectural and design issues that affect the construction of software systems, especially large-scale systems.*

Look-and-Feel Observation

These observations help you design attractive, user-friendly graphical user interfaces that conform to industry norms.

Thousands of Index Entries. We've included a comprehensive index, which is especially useful when you use the book as a reference.

Software Used in *Java for Programmers, 2/e*

All the software you'll need for this book is available free for download from the web. See the Before You Begin section that follows the Preface for links to each download.

We wrote most of the examples in *Java for Programmers, 2/e*, using the free Java Standard Edition Development Kit (JDK) 6. For the Java SE 7 modules, we used the OpenJDK's early access version of JDK 7 (`download.java.net/jdk7/`). In Chapters 26–28, we also used the Netbeans IDE, and in Chapter 25, we used MySQL and MySQL Connector/J. You can find additional resources and software downloads in our Java Resource Centers at:

```
www.deitel.com/ResourceCenters.html
```

Discounts on *Deitel Developer Series* Books

If you'd like to receive information on professional *Deitel Developer Series* titles, including *Android for Programmers: An App-Driven Approach*, please register your copy of *Java for Programmers, 2/e* at `informit.com/register`. You'll receive information on how to purchase *Android for Programmers* at a discount.

Java Fundamentals: Parts I, II and III, Second Edition LiveLessons Video Training Product

Our *Java Fundamentals: Parts I, II and III, Second Edition* LiveLessons video training product shows you what you need to know to start building robust, powerful software with Java. It includes 20+ hours of expert training synchronized with *Java for Programmers, 2/e*. Check out our growing list of LiveLessons video products:

- *Java Fundamentals I and II*
- *C# 2010 Fundamentals I, II, and III*
- *C# 2008 Fundamentals I and II*
- *C++ Fundamentals I and II*
- *iPhone App-Development Fundamentals I and II*
- *JavaScript Fundamentals I and II*
- *Visual Basic 2010 Fundamentals I and II*

Coming Soon

- *Java Fundamentals I, II and III, Second Edition*
- *C Fundamentals I and II*
- *Android App Development Fundamentals I and II*
- *iPhone and iPad App-Development Fundamentals I and II, Second Edition*

For additional information about Deitel LiveLessons video products, visit:

```
www.deitel.com/livelessons
```

Acknowledgments

We'd like to thank Abbey Deitel and Barbara Deitel for long hours devoted to this project. Barbara devoted long hours to Internet research to support our writing efforts. Abbey wrote the new engaging Chapter 1 and the new cover copy. We're fortunate to have worked on this project with the dedicated team of publishing professionals at Pearson. We appreciate

the guidance, savvy and energy of Mark Taub, Editor-in-Chief of Computer Science. John Fuller managed the book's production. Sandra Schroeder did the cover design.

Reviewers

We wish to acknowledge the efforts of the reviewers who contributed to the recent editions of this content. They scrutinized the text and the programs and provided countless suggestions for improving the presentation: Lance Andersen (Oracle), Soundararajan Angusamy (Sun Microsystems), Joseph Bowbeer (Consultant), William E. Duncan (Louisiana State University), Diana Franklin (University of California, Santa Barbara), Edward F. Gehringer (North Carolina State University), Huiwei Guan (Northshore Community College), Ric Heishman (George Mason University), Dr. Heinz Kabutz (JavaSpecialists.eu), Patty Kraft (San Diego State University), Lawrence Premkumar (Sun Microsystems), Tim Margush (University of Akron), Sue McFarland Metzger (Villanova University), Shyamal Mitra (The University of Texas at Austin), Peter Pilgrim (Java Champion, Consultant), Manjeet Rege, Ph.D. (Rochester Institute of Technology), Manfred Riem (Java Champion, Consultant, Robert Half), Simon Ritter (Oracle), Susan Rodger (Duke University), Amr Sabry (Indiana University), José Antonio González Seco (Parliament of Andalusia), Sang Shin (Sun Microsystems), S. Sivakumar (Astra Infotech Private Limited), Raghavan "Rags" Srinivas (Intuit), Monica Sweat (Georgia Tech), Vinod Varma (Astra Infotech Private Limited) and Alexander Zuev (Sun Microsystems).

Well, there you have it! As you read the book, we'd appreciate your comments, criticisms, corrections and suggestions for improvement. Please address all correspondence to:

```
deitel@deitel.com
```

We'll respond promptly. We hope you enjoy working with *Java for Programmers, 2/e*. Good luck!

Paul and Harvey Deitel

About the Authors

Paul J. Deitel, CEO and Chief Technical Officer of Deitel & Associates, Inc., is a graduate of MIT, where he studied Information Technology. He holds the Sun (now Oracle) Certified Java Programmer and Certified Java Developer certifications, and is an Oracle Java Champion. Through Deitel & Associates, Inc., he has delivered Java, C#, Visual Basic, C++, C and Internet programming courses to industry clients, including Cisco, IBM, Sun Microsystems, Dell, Siemens, Lucent Technologies, Fidelity, NASA at the Kennedy Space Center, the National Severe Storm Laboratory, White Sands Missile Range, Rogue Wave Software, Boeing, SunGard Higher Education, Stratus, Cambridge Technology Partners, One Wave, Hyperion Software, Adra Systems, Entergy, CableData Systems, Nortel Networks, Puma, iRobot, Invensys and many more. He and his co-author, Dr. Harvey M. Deitel, are the world's best-selling programming-language textbook/professional book authors.

Dr. Harvey M. Deitel, Chairman and Chief Strategy Officer of Deitel & Associates, Inc., has 50 years of experience in the computer field. Dr. Deitel earned B.S. and M.S. degrees from MIT and a Ph.D. from Boston University. He has extensive industry and academic experience, including earning tenure and serving as the Chairman of the Computer Science Department at Boston College before founding Deitel & Associates, Inc., with his son, Paul J. Deitel. He and Paul are the co-authors of dozens of books and multimedia packages and they are writing many more. With translations published in Japanese, German, Russian, Chinese, Spanish, Korean, French, Polish, Italian, Portuguese, Greek, Urdu and Turkish, the Deitels' texts have earned international recognition. Dr. Deitel has delivered hundreds of professional seminars to major corporations, academic institutions, government organizations and the military.

About Deitel & Associates, Inc.

Deitel & Associates, Inc., founded by Paul Deitel and Harvey Deitel, is an internationally recognized authoring, corporate training and software development organization specializing in computer programming languages, object technology, Android and iPhone app development, and Internet and web software technology. The company offers instructor-led training courses delivered at client sites worldwide on major programming languages and platforms, such as Java™, C, C++, Visual C#®, Visual Basic®, Objective-C, and iPhone and iPad app development, Android app development, XML®, Python®, object technology, Internet and web programming, and a growing list of additional programming and software development courses. The company's clients include many of the world's largest companies, government agencies, branches of the military, and academic institutions.

Through its 35-year publishing partnership with Prentice Hall/Pearson, Deitel & Associates, Inc., publishes leading-edge programming professional books, college textbooks, and *LiveLessons* DVD- and web-based video courses. Deitel & Associates, Inc. and the authors can be reached at:

```
deitel@deitel.com
```

To learn more about Deitel's *Dive Into® Series* Corporate Training curriculum, visit:

```
www.deitel.com/training/
```

subscribe to the free *Deitel® Buzz Online* e-mail newsletter at:

```
www.deitel.com/newsletter/subscribe.html
```

and follow the authors on Facebook

```
www.facebook.com/DeitelFan
```

and Twitter

```
@deitel
```

To request a proposal for on-site, instructor-led training at your company or organization, e-mail

```
deitel@deitel.com
```

Individuals wishing to purchase Deitel books and *LiveLessons* DVD training courses can do so through `www.deitel.com`. Bulk orders by corporations, the government, the military and academic institutions should be placed directly with Pearson. For more information, visit `www.pearsoned.com/professional/index.htm`.

This page intentionally left blank

Before You Begin

This section contains information you should review before using this book and instructions to ensure that your computer is set up properly for use with this book. We'll post updates (if any) to the Before You Begin section on the book's website:

```
www.deitel.com/books/javafp2/
```

Font and Naming Conventions

We use fonts to distinguish between on-screen components (such as menu names and menu items) and Java code or commands. Our convention is to emphasize on-screen components in a sans-serif bold **Helvetica** font (for example, **File** menu) and to emphasize Java code and commands in a sans-serif `Lucida` font (for example, `System.out.println()`).

Software Used in the Book

All the software you'll need for this book is available free for download from the web.

Java SE Software Development Kit (JDK) 6 and 7

We wrote most of the examples in *Java for Programmers, 2/e*, using the free Java Standard Edition Development Kit (JDK) 6, which is available from:

```
www.oracle.com/technetwork/java/javase/downloads/index.html
```

For the Java SE 7 modules, we used the OpenJDK's early access version of JDK 7, which is available from:

```
dlc.sun.com.edgesuite.net/jdk7/binaries-/index.html
```

Java DB, MySQL and MySQL Connector/J

In Chapter 25, we use the Java DB and MySQL Community Edition database management systems. Java DB is part of the JDK installation. At the time of this writing, the JDK's 64-bit installer was not properly installing Java DB. If you are using the 64-bit version of Java, you may need to install Java DB separately. You can download Java DB from:

```
www.oracle.com/technetwork/java/javadb/downloads/index.html
```

At the time of this writing, the latest release of MySQL Community Edition was 5.5.8. To install MySQL Community Edition on Windows, Linux or Mac OS X, see the installation overview for your platform at:

- Windows: `dev.mysql.com/doc/refman/5.5/en/windows-installation.html`
- Linux: `dev.mysql.com/doc/refman/5.5/en/linux-installation-rpm.html`
- Mac OS X: `dev.mysql.com/doc/refman/5.5/en/macosx-installation.html`

Carefully follow the instructions for downloading and installing the software on your platform. The downloads are available from:

```
dev.mysql.com/downloads/mysql/
```

You also need to install MySQL Connector/J (the J stands for Java), which allows programs to use JDBC to interact with MySQL. MySQL Connector/J can be downloaded from

```
dev.mysql.com/downloads/connector/j/
```

At the time of this writing, the current generally available release of MySQL Connector/J is 5.1.14. The documentation for Connector/J is located at

```
dev.mysql.com/doc/refman/5.5/en/connector-j.html
```

To install MySQL Connector/J, carefully follow the installation instructions at:

```
dev.mysql.com/doc/refman/5.5/en/connector-j-installing.html
```

We *do not* recommend modifying your system's `CLASSPATH` environment variable, which is discussed in the installation instructions. Instead, we'll show you how use MySQL Connector/J by specifying it as a command-line option when you execute your applications.

Obtaining the Code Examples

The examples for *Java for Programmers, 2/e* are available for download at

```
www.deitel.com/books/javafp2/
```

If you're not already registered at our website, go to `www.deitel.com` and click the **Register** link below our logo in the upper-left corner of the page. Fill in your information. There's no charge to register, and we do not share your information with anyone. We send you only account-management e-mails unless you register separately for our free *Deitel® Buzz Online* e-mail newsletter at `www.deitel.com/newsletter/subscribe.html`. After registering for the site, you'll receive a confirmation e-mail with your verification code. *Click the link in the confirmation e-mail to complete your registration*. Configure your e-mail client to allow e-mails from `deitel.com` to ensure that the confirmation email is not filtered as junk mail.

Next, go to `www.deitel.com` and sign in using the **Login** link below our logo in the upper-left corner of the page. Go to `www.deitel.com/books/javafp2/`. You'll find the link to download the examples under the heading **Download Code Examples and Other Premium Content for Registered Users**. Write down the location where you choose to save the ZIP file on your computer. We assume the examples are located at `C:\Examples` on your computer.

Setting the PATH Environment Variable

The `PATH` environment variable on your computer designates which directories the computer searches when looking for applications, such as the applications that enable you to compile and run your Java applications (called `javac` and `java`, respectively). *Carefully follow the installation instructions for Java on your platform to ensure that you set the* `PATH` *environment variable correctly.*

If you do not set the PATH variable correctly, when you use the JDK's tools, you'll receive a message like:

```
'java' is not recognized as an internal or external command,
operable program or batch file.
```

In this case, go back to the installation instructions for setting the PATH and recheck your steps. If you've downloaded a newer version of the JDK, you may need to change the name of the JDK's installation directory in the PATH variable.

Setting the CLASSPATH Environment Variable

If you attempt to run a Java program and receive a message like

```
Exception in thread "main" java.lang.NoClassDefFoundError: YourClass
```

then your system has a CLASSPATH environment variable that must be modified. To fix the preceding error, follow the steps in setting the PATH environment variable, to locate the CLASSPATH variable, then edit the variable's value to include the local directory—typically represented as a dot (.). On Windows add

```
.;
```

at the beginning of the CLASSPATH's value (with no spaces before or after these characters). On other platforms, replace the semicolon with the appropriate path separator characters—often a colon (:)

Java's Nimbus Look-and-Feel

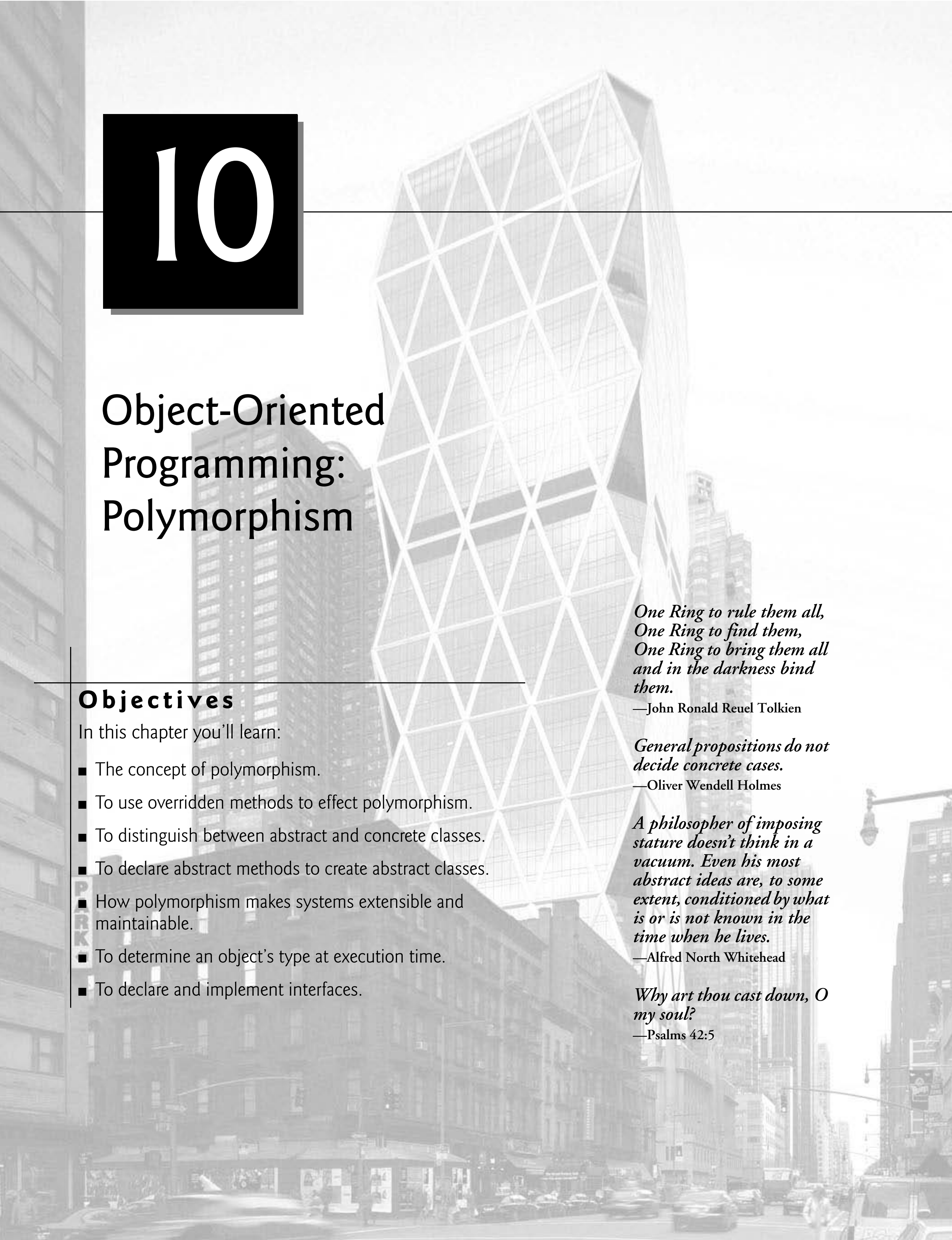
Java comes bundled with an elegant, cross-platform look-and-feel known as Nimbus. For programs with graphical user interfaces, we've configured our systems to use Nimbus as the default look-and-feel.

To set Nimbus as the default for all Java applications, you must create a text file named swing.properties in the lib folder of both your JDK installation folder and your JRE installation folder. Place the following line of code in the file:

```
swing.defaultlaf=com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel
```

For more information on locating these installation folders visit java.sun.com/javase/6/webnotes/install/index.html. [*Note:* In addition to the standalone JRE, there's a JRE nested in your JDK's installation folder. If you're using an IDE that depends on the JDK (e.g., NetBeans), you may also need to place the swing.properties file in the nested jre folder's lib folder.]

This page intentionally left blank



10

Object-Oriented Programming: Polymorphism

One Ring to rule them all,
One Ring to find them,
One Ring to bring them all
and in the darkness bind them.
—John Ronald Reuel Tolkien

General propositions do not decide concrete cases.
—Oliver Wendell Holmes

A philosopher of imposing stature doesn't think in a vacuum. Even his most abstract ideas are, to some extent, conditioned by what is or is not known in the time when he lives.
—Alfred North Whitehead

Why art thou cast down, O my soul?
—Psalms 42:5

Objectives

In this chapter you'll learn:

- The concept of polymorphism.
- To use overridden methods to effect polymorphism.
- To distinguish between abstract and concrete classes.
- To declare abstract methods to create abstract classes.
- How polymorphism makes systems extensible and maintainable.
- To determine an object's type at execution time.
- To declare and implement interfaces.

Outline

10.1 Introduction

We continue our study of object-oriented programming by explaining and demonstrating **polymorphism** with inheritance hierarchies. Polymorphism enables you to "program in the general" rather than "program in the specific." In particular, polymorphism enables you to write programs that process objects that share the same superclass (either directly or indirectly) as if they're all objects of the superclass; this can simplify programming.

Consider the following example of polymorphism. Suppose we create a program that simulates the movement of several types of animals for a biological study. Classes `Fish`, `Frog` and `Bird` represent the types of animals under investigation. Imagine that each class extends superclass `Animal`, which contains a method `move` and maintains an animal's current location as *x-y* coordinates. Each subclass implements method `move`. Our program maintains an `Animal` array containing references to objects of the various `Animal` subclasses. To simulate the animals' movements, the program sends each object the *same* message once per second—namely, `move`. Each specific type of `Animal` responds to a `move` message in its own way—a `Fish` might swim three feet, a `Frog` might jump five feet and a `Bird` might fly ten feet. Each object knows how to modify its *x-y* coordinates appropriately for its *specific* type of movement. Relying on each object to know how to "do the right thing" (i.e., do what is appropriate for that type of object) in response to the same method call is the key concept of polymorphism. The same message (in this case, `move`) sent to a variety of objects has "many forms" of results—hence the term polymorphism.

Implementing for Extensibility

With polymorphism, we can design and implement systems that are easily extensible—new classes can be added with little or no modification to the general portions of the program, as long as the new classes are part of the inheritance hierarchy that the program processes generically. The only parts of a program that must be altered are those that require direct knowledge of the new classes that we add to the hierarchy. For example, if we extend class `Animal` to create class `Tortoise` (which might respond to a `move` message by crawling one inch), we need to write only the `Tortoise` class and the part of the simulation that instantiates a `Tortoise` object. The portions of the simulation that tell each `Animal` to move generically can remain the same.

Chapter Overview

First, we discuss common examples of polymorphism. We then provide a simple example demonstrating polymorphic behavior. We use superclass references to manipulate *both* superclass objects and subclass objects polymorphically.

We then present a case study that revisits the employee hierarchy of Section 9.4.5. We develop a simple payroll application that polymorphically calculates the weekly pay of several different types of employees using each employee's `earnings` method. Though the earnings of each type of employee are calculated in a specific way, polymorphism allows us to process the employees "in the general." In the case study, we enlarge the hierarchy to include two new classes—`SalariedEmployee` (for people paid a fixed weekly salary) and `HourlyEmployee` (for people paid an hourly salary and "time-and-a-half" for overtime). We declare a common set of functionality for all the classes in the updated hierarchy in an "abstract" class, `Employee`, from which "concrete"classes `SalariedEmployee`, `HourlyEmployee` and `CommissionEmployee` inherit directly and "concrete" class `BasePlusCommissionEmployee` inherits indirectly. As you'll soon see, *when we invoke each employee's `earnings` method off a superclass `Employee` reference, the correct earnings subclass calculation is performed,* due to Java's polymorphic capabilities.

Programming in the Specific

Occasionally, when performing polymorphic processing, we need to program "in the specific." Our `Employee` case study demonstrates that a program can determine the type of an object at *execution time* and act on that object accordingly. In the case study, we've decided that `BasePlusCommissionEmployees` should receive 10% raises on their base salaries. So, we use these capabilities to determine whether a particular employee object *is a* `BasePlusCommissionEmployee`. If so, we increase that employee's base salary by 10%.

Interfaces

The chapter continues with an introduction to Java interfaces. An interface describes a set of methods that can be called on an object, but does *not* provide concrete implementations for all the methods. You can declare classes that **implement** (i.e., provide concrete implementations for the methods of) one or more interfaces. Each interface method must be declared in all the classes that explicitly implement the interface. Once a class implements an interface, all objects of that class have an *is-a* relationship with the interface type, and all objects of the class are guaranteed to provide the functionality described by the interface. This is true of all subclasses of that class as well.

Interfaces are particularly useful for assigning common functionality to possibly *unrelated* classes. This allows objects of unrelated classes to be processed polymorphically—objects of classes that implement the same interface can respond to all of the interface method calls. To demonstrate creating and using interfaces, we modify our payroll application to create a general accounts payable application that can calculate payments due for company employees and invoice amounts to be billed for purchased goods. As you'll see, interfaces enable polymorphic capabilities similar to those possible with inheritance.

10.2 Polymorphism Examples

We now consider several additional examples of polymorphism.

Quadrilaterals

If class `Rectangle` is derived from class `Quadrilateral`, then a `Rectangle` object is a more specific version of a `Quadrilateral`. Any operation (e.g., calculating the perimeter or the area) that can be performed on a `Quadrilateral` can also be performed on a `Rectangle`. These operations can also be performed on other `Quadrilaterals`, such as `Squares`, `Parallelograms` and `Trapezoids`. The polymorphism occurs when a program invokes a method through a superclass `Quadrilateral` variable—at execution time, the correct subclass version of the method is called, based on the type of the reference stored in the superclass variable. You'll see a simple code example that illustrates this process in Section 10.3.

Space Objects in a Video Game

Suppose we design a video game that manipulates objects of classes `Martian`, `Venusian`, `Plutonian`, `SpaceShip` and `LaserBeam`. Imagine that each class inherits from the superclass `SpaceObject`, which contains method `draw`. Each subclass implements this method. A screen manager maintains a collection (e.g., a `SpaceObject` array) of references to objects of the various classes. To refresh the screen, the screen manager periodically sends each object the same message—namely, `draw`. However, each object responds its own way, based on its class. For example, a `Martian` object might draw itself in red with green eyes and the appropriate number of antennae. A `SpaceShip` object might draw itself as a bright silver flying saucer. A `LaserBeam` object might draw itself as a bright red beam across the screen. Again, the *same* message (in this case, `draw`) sent to a variety of objects has "many forms" of results.

A screen manager might use polymorphism to facilitate adding new classes to a system with minimal modifications to the system's code. Suppose that we want to add `Mercurian` objects to our video game. To do so, we'd build a class `Mercurian` that extends `SpaceObject` and provides its own `draw` method implementation. When `Mercurian` objects appear in the `SpaceObject` collection, the screen manager code *invokes method `draw`, exactly as it does for every other object in the collection, regardless of its type.* So the new `Mercurian` objects simply "plug right in" without any modification of the screen manager code by the programmer. Thus, without modifying the system (other than to build new classes and modify the code that creates new objects), you can use polymorphism to conveniently include additional types that were not envisioned when the system was created.

Software Engineering Observation 10.1

Polymorphism enables you to deal in generalities and let the execution-time environment handle the specifics. You can command objects to behave in manners appropriate to those objects, without knowing their types (as long as the objects belong to the same inheritance hierarchy).

Software Engineering Observation 10.2

Polymorphism promotes extensibility: Software that invokes polymorphic behavior is independent of the object types to which messages are sent. New object types that can respond to existing method calls can be incorporated into a system without modifying the base system. Only client code that instantiates new objects must be modified to accommodate new types.

10.3 Demonstrating Polymorphic Behavior

Section 9.4 created a class hierarchy, in which class `BasePlusCommissionEmployee` inherited from `CommissionEmployee`. The examples in that section manipulated `CommissionEmployee` and `BasePlusCommissionEmployee` objects by using references to them to invoke their methods—we aimed superclass variables at superclass objects and subclass variables at subclass objects. These assignments are natural and straightforward—superclass variables are *intended* to refer to superclass objects, and subclass variables are *intended* to refer to subclass objects. However, as you'll soon see, other assignments are possible.

In the next example, we aim a *superclass* reference at *a subclass* object. We then show how invoking a method on a subclass object via a superclass reference invokes the *subclass* functionality—the type of the *referenced object*, not the type of the *variable*, determines which method is called. This example demonstrates that *an object of a subclass can be treated as an object of its superclass,* enabling various interesting manipulations. A program can create an array of superclass variables that refer to objects of many subclass types. This is allowed because each subclass object *is an* object of its superclass. For instance, we can assign the reference of a `BasePlusCommissionEmployee` object to a superclass `CommissionEmployee` variable, because a `BasePlusCommissionEmployee` *is a* `CommissionEmployee`—we can treat a `BasePlusCommissionEmployee` as a `CommissionEmployee`.

As you'll learn later in the chapter, you *cannot treat a superclass object as a subclass object,* because a superclass object is *not* an object of any of its subclasses. For example, we cannot assign the reference of a `CommissionEmployee` object to a subclass `BasePlusCommissionEmployee` variable, because a `CommissionEmployee` is *not* a `BasePlusCommissionEmployee`—a `CommissionEmployee` does *not* have a `baseSalary` instance variable and does *not* have methods `setBaseSalary` and `getBaseSalary`. The *is-a* relationship applies only *up the hierarchy* from a subclass to its direct (and indirect) superclasses, and *not* vice versa (i.e., not down the hierarchy from a superclass to its subclasses).

The Java compiler *does* allow the assignment of a superclass reference to a subclass variable if we explicitly cast the superclass reference to the subclass type—a technique we discuss in Section 10.5. Why would we ever want to perform such an assignment? A superclass reference can be used to invoke only the methods declared in the superclass—attempting to invoke subclass-only methods through a superclass reference results in compilation errors. If a program needs to perform a subclass-specific operation on a subclass object referenced by a superclass variable, the program must first cast the superclass reference to a subclass reference through a technique known as **downcasting**. This enables the program to invoke subclass methods that are not in the superclass. We show a downcasting example in Section 10.5.

The example in Fig. 10.1 demonstrates three ways to use superclass and subclass variables to store references to superclass and subclass objects. The first two are straightforward—as in Section 9.4, we assign a superclass reference to a superclass variable, and a subclass reference to a subclass variable. Then we demonstrate the relationship between subclasses and superclasses (i.e., the *is-a* relationship) by assigning a subclass reference to a superclass variable. This program uses classes `CommissionEmployee` and `BasePlusCommissionEmployee` from Fig. 9.10 and Fig. 9.11, respectively.

In Fig. 10.1, lines 10–11 create a `CommissionEmployee` object and assign its reference to a `CommissionEmployee` variable. Lines 14–16 create a `BasePlusCommissionEmployee` object and assign its reference to a `BasePlusCommissionEmployee` variable. These assign-

```java
// Fig. 10.1: PolymorphismTest.java
// Assigning superclass and subclass references to superclass and
// subclass variables.

public class PolymorphismTest
{
   public static void main( String[] args )
   {
      // assign superclass reference to superclass variable
      CommissionEmployee commissionEmployee = new CommissionEmployee(
         "Sue", "Jones", "222-22-2222", 10000, .06 );

      // assign subclass reference to subclass variable
      BasePlusCommissionEmployee basePlusCommissionEmployee =
         new BasePlusCommissionEmployee(
         "Bob", "Lewis", "333-33-3333", 5000, .04, 300 );

      // invoke toString on superclass object using superclass variable
      System.out.printf( "%s %s:\n\n%s\n\n",
         "Call CommissionEmployee's toString with superclass reference ",
         "to superclass object", commissionEmployee.toString() );

      // invoke toString on subclass object using subclass variable
      System.out.printf( "%s %s:\n\n%s\n\n",
         "Call BasePlusCommissionEmployee's toString with subclass",
         "reference to subclass object",
         basePlusCommissionEmployee.toString() );

      // invoke toString on subclass object using superclass variable
      CommissionEmployee commissionEmployee2 =
         basePlusCommissionEmployee;
      System.out.printf( "%s %s:\n\n%s\n",
         "Call BasePlusCommissionEmployee's toString with superclass",
         "reference to subclass object", commissionEmployee2.toString() );
   } // end main
} // end class PolymorphismTest
```

```
Call CommissionEmployee's toString with superclass reference to superclass
object:

commission employee: Sue Jones
social security number: 222-22-2222
gross sales: 10000.00
commission rate: 0.06

Call BasePlusCommissionEmployee's toString with subclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

Fig. 10.1 | Assigning superclass and subclass references to superclass and subclass variables. (Part 1 of 2.)

```
Call BasePlusCommissionEmployee's toString with superclass reference to
subclass object:

base-salaried commission employee: Bob Lewis
social security number: 333-33-3333
gross sales: 5000.00
commission rate: 0.04
base salary: 300.00
```

Fig. 10.1 | Assigning superclass and subclass references to superclass and subclass variables. (Part 2 of 2.)

ments are natural—for example, a `CommissionEmployee` variable's primary purpose is to hold a reference to a `CommissionEmployee` object. Lines 19–21 use `commissionEmployee` to invoke `toString` explicitly. Because `commissionEmployee` refers to a `CommissionEmployee` object, superclass `CommissionEmployee`'s version of `toString` is called. Similarly, lines 24–27 use `basePlusCommissionEmployee` to invoke `toString` explicitly on the `BasePlusCommissionEmployee` object. This invokes subclass `BasePlusCommissionEmployee`'s version of `toString`.

Lines 30–31 then assign the reference of subclass object `basePlusCommissionEmployee` to a superclass `CommissionEmployee` variable, which lines 32–34 use to invoke method `toString`. *When a superclass variable contains a reference to a subclass object, and that reference is used to call a method, the subclass version of the method is called.* Hence, `commissionEmployee2.toString()` in line 34 actually calls class `BasePlusCommissionEmployee`'s `toString` method. The Java compiler allows this "crossover" because an object of a subclass *is an* object of its superclass (but not vice versa). When the compiler encounters a method call made through a variable, the compiler determines if the method can be called by checking the variable's class type. If that class contains the proper method declaration (or inherits one), the call is compiled. At execution time, the type of the object to which the variable refers determines the actual method to use. This process, called *dynamic binding*, is discussed in detail in Section 10.5.

10.4 Abstract Classes and Methods

When we think of a class, we assume that programs will create objects of that type. Sometimes it's useful to declare classes—called **abstract classes**—for which you *never* intend to create objects. Because they're used only as superclasses in inheritance hierarchies, we refer to them as **abstract superclasses**. These classes cannot be used to instantiate objects, because, as we'll soon see, abstract classes are *incomplete*. Subclasses must declare the "missing pieces" to become "concrete" classes, from which you can instantiate objects. Otherwise, these subclasses, too, will be abstract. We demonstrate abstract classes in Section 10.5.

Purpose of Abstract Classes

An abstract class's purpose is to provide an appropriate superclass from which other classes can inherit and thus share a common design. In the `Shape` hierarchy of Fig. 9.3, for example, subclasses inherit the notion of what it means to be a `Shape`—perhaps common attributes such as `location`, `color` and `borderThickness`, and behaviors such as `draw`, `move`, `resize` and `changeColor`. Classes that can be used to instantiate objects are called **concrete**

classes. Such classes provide implementations of *every* method they declare (some of the implementations can be inherited). For example, we could derive concrete classes `Circle`, `Square` and `Triangle` from abstract superclass `TwoDimensionalShape`. Similarly, we could derive concrete classes `Sphere`, `Cube` and `Tetrahedron` from abstract superclass `ThreeDimensionalShape`. Abstract superclasses are *too general* to create real objects—they specify only what is common among subclasses. We need to be more *specific* before we can create objects. For example, if you send the `draw` message to abstract class `TwoDimensionalShape`, the class knows that two-dimensional shapes should be drawable, but it does not know what specific shape to draw, so it cannot implement a real `draw` method. Concrete classes provide the specifics that make it reasonable to instantiate objects.

Not all hierarchies contain abstract classes. However, you'll often write client code that uses only abstract superclass types to reduce the client code's dependencies on a range of subclass types. For example, you can write a method with a parameter of an abstract superclass type. When called, such a method can receive an object of any concrete class that directly or indirectly extends the superclass specified as the parameter's type.

Abstract classes sometimes constitute several levels of a hierarchy. For example, the `Shape` hierarchy of Fig. 9.3 begins with abstract class `Shape`. On the next level of the hierarchy are *abstract* classes `TwoDimensionalShape` and `ThreeDimensionalShape`. The next level of the hierarchy declares *concrete* classes for `TwoDimensionalShapes` (`Circle`, `Square` and `Triangle`) and for `ThreeDimensionalShapes` (`Sphere`, `Cube` and `Tetrahedron`).

Declaring an Abstract Class and Abstract Methods

You make a class abstract by declaring it with keyword **`abstract`**. An abstract class normally contains one or more **abstract methods**. An abstract method is one with keyword `abstract` in its declaration, as in

```
public abstract void draw(); // abstract method
```

Abstract methods do *not* provide implementations. A class that contains *any* abstract methods must be explicitly declared `abstract` even if that class contains some concrete (nonabstract) methods. Each concrete subclass of an abstract superclass also must provide concrete implementations of each of the superclass's abstract methods. Constructors and `static` methods cannot be declared `abstract`. Constructors are not inherited, so an `abstract` constructor could never be implemented. Though non-`private static` methods are inherited, they cannot be overridden. Since `abstract` methods are meant to be overridden so that they can process objects based on their types, it would not make sense to declare a `static` method as `abstract`.

Software Engineering Observation 10.3

An abstract class declares common attributes and behaviors (both abstract and concrete) of the various classes in a class hierarchy. An abstract class typically contains one or more abstract methods that subclasses must override if they are to be concrete. The instance variables and concrete methods of an abstract class are subject to the normal rules of inheritance.

Common Programming Error 10.1

Attempting to instantiate an object of an abstract class is a compilation error.

Common Programming Error 10.2

Failure to implement a superclass's abstract methods in a subclass is a compilation error unless the subclass is also declared `abstract`*.*

Using Abstract Classes to Declare Variables

Although we cannot instantiate objects of abstract superclasses, you'll soon see that we *can* use abstract superclasses to declare variables that can hold references to objects of any concrete class derived from those abstract superclasses. Programs typically use such variables to manipulate subclass objects polymorphically. You also can use abstract superclass names to invoke `static` methods declared in those abstract superclasses.

Consider another application of polymorphism. A drawing program needs to display many shapes, including types of new shapes that you'll add to the system after writing the drawing program. The drawing program might need to display shapes, such as `Circle`s, `Triangle`s, `Rectangle`s or others, that derive from abstract class `Shape`. The drawing program uses `Shape` variables to manage the objects that are displayed. To draw any object in this inheritance hierarchy, the drawing program uses a superclass `Shape` variable containing a reference to the subclass object to invoke the object's `draw` method. This method is declared `abstract` in superclass `Shape`, so each concrete subclass *must* implement method `draw` in a manner specific to that shape—each object in the `Shape` inheritance hierarchy *knows how to draw itself*. The drawing program does not have to worry about the type of each object or whether the program has ever encountered objects of that type.

Layered Software Systems

Polymorphism is particularly effective for implementing so-called layered software systems. In operating systems, for example, each type of physical device could operate quite differently from the others. Even so, commands to read or write data from and to devices may have a certain uniformity. For each device, the operating system uses a piece of software called a *device driver* to control all communication between the system and the device. The write message sent to a device-driver object needs to be interpreted specifically in the context of that driver and how it manipulates devices of a specific type. However, the write call itself really is no different from the write to any other device in the system—place some number of bytes from memory onto that device. An object-oriented operating system might use an abstract superclass to provide an "interface" appropriate for all device drivers. Then, through inheritance from that abstract superclass, subclasses are formed that all behave similarly. The device-driver methods are declared as abstract methods in the abstract superclass. The implementations of these abstract methods are provided in the concrete subclasses that correspond to the specific types of device drivers. New devices are always being developed, often long after the operating system has been released. When you buy a new device, it comes with a device driver provided by the device vendor. The device is immediately operational after you connect it to your computer and install the driver. This is another elegant example of how polymorphism makes systems *extensible*.

10.5 Case Study: Payroll System Using Polymorphism

This section reexamines the hierarchy that we explored throughout Section 9.4. Now we use an abstract method and polymorphism to perform payroll calculations based on an enhanced employee inheritance hierarchy that meets the following requirements:

A company pays its employees on a weekly basis. The employees are of four types: Salaried employees are paid a fixed weekly salary regardless of the number of hours worked, hourly employees are paid by the hour and receive overtime pay (i.e., 1.5 times their hourly salary rate) for all hours worked in excess of 40 hours, commission employees are paid a percentage of their sales and base-salaried commission employees receive a base salary plus a percentage of their sales. For the current pay period, the company has decided to reward salaried-commission employees by adding 10% to their base salaries. The company wants to write an application that performs its payroll calculations polymorphically.

We use `abstract` class `Employee` to represent the general concept of an employee. The classes that extend `Employee` are `SalariedEmployee`, `CommissionEmployee` and `HourlyEmployee`. Class `BasePlusCommissionEmployee`—which extends `CommissionEmployee`—represents the last employee type. The UML class diagram in Fig. 10.2 shows the inheritance hierarchy for our polymorphic employee-payroll application. Abstract class name `Employee` is italicized—a convention of the UML.

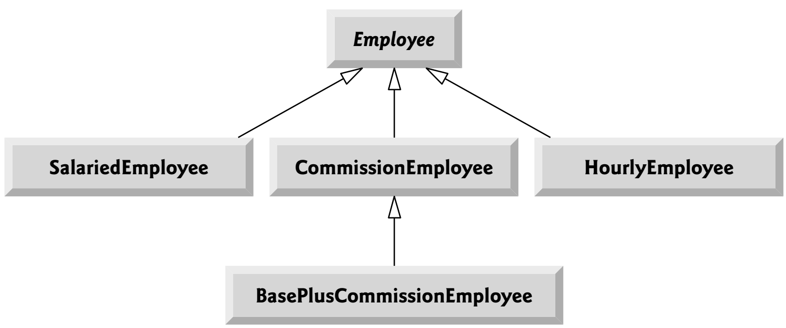


Fig. 10.2 | `Employee` hierarchy UML class diagram.

Abstract superclass `Employee` declares the "interface" to the hierarchy—that is, the set of methods that a program can invoke on all `Employee` objects. We use the term "interface" here in a general sense to refer to the various ways programs can communicate with objects of any `Employee` subclass. Be careful not to confuse the general notion of an "interface" with the formal notion of a Java interface, the subject of Section 10.7. Each employee, regardless of the way his or her earnings are calculated, has a first name, a last name and a social security number, so `private` instance variables `firstName`, `lastName` and `socialSecurityNumber` appear in abstract superclass `Employee`.

The following sections implement the `Employee` class hierarchy of Fig. 10.2. The first section implements abstract superclass `Employee`. The next four sections each implement one of the concrete classes. The last section implements a test program that builds objects of all these classes and processes those objects polymorphically.

10.5.1 Abstract Superclass `Employee`

Class `Employee` (Fig. 10.4) provides methods `earnings` and `toString`, in addition to the *get* and *set* methods that manipulate `Employee`'s instance variables. An `earnings` method certainly applies generically to all employees. But each earnings calculation depends on the employee's class. So we declare `earnings` as `abstract` in superclass `Employee` because a default implementation does not make sense for that method—there isn't enough information to determine what amount `earnings` should return. Each subclass overrides `earnings` with an appropriate implementation. To calculate an employee's earnings, the program assigns to a superclass `Employee` variable a reference to the employee's object, then invokes the `earnings` method on that variable. We maintain an array of `Employee` variables, each holding a reference to an `Employee` object. (Of course, there cannot be `Employee` objects, because `Employee` is an abstract class. Because of inheritance, however, all objects of all subclasses of `Employee` may nevertheless be thought of as `Employee` objects.) The program will iterate through the array and call method `earnings` for each `Employee` object. Java processes these method calls polymorphically. Declaring `earnings` as an `abstract` method in `Employee` enables the calls to `earnings` through `Employee` variables to compile and forces every direct concrete subclass of `Employee` to override `earnings`.

Method `toString` in class `Employee` returns a `String` containing the first name, last name and social security number of the employee. As we'll see, each subclass of `Employee` overrides method `toString` to create a `String` representation of an object of that class that contains the employee's type (e.g., `"salaried employee:"`) followed by the rest of the employee's information.

The diagram in Fig. 10.3 shows each of the five classes in the hierarchy down the left side and methods `earnings` and `toString` across the top. For each class, the diagram

	`earnings`	`toString`
`Employee`	`abstract`	*firstName lastName* `social security number:` *SSN*
`Salaried-Employee`	`weeklySalary`	`salaried employee:` *firstName lastName* `social security number:` *SSN* `weekly salary:` *weeklySalary*

`Hourly-Employee`	`if (hours <= 40)`
	`  wage * hours`
	`else if (hours > 40)`
	`{`
	`  40 * wage +`
	`  ( hours - 40 ) *`
	`  wage * 1.5`
	`}`
		`hourly employee:` *firstName lastName*
		`social security number:` *SSN*
		`hourly wage:` *wage*`; hours worked:` *hours*

`Commission-Employee`	`commissionRate *` `grossSales`	`commission employee:` *firstName lastName* `social security number:` *SSN* `gross sales:` *grossSales*`;` `commission rate:` *commissionRate*

`BasePlus-Commission-Employee`	`(commissionRate *`
	`grossSales) +`
	`baseSalary`
		`base salaried commission employee:`
		firstName lastName
		`social security number:` *SSN*
		`gross sales:` *grossSales*`;`
		`commission rate:` *commissionRate*`;`
		`base salary:` *baseSalary*

Fig. 10.3 | Polymorphic interface for the `Employee` hierarchy classes.

shows the desired results of each method. We do not list superclass `Employee`'s *get* and *set* methods because they're not overridden in any of the subclasses—each of these methods is inherited and used "as is" by each subclass.

Let's consider class `Employee`'s declaration (Fig. 10.4). The class includes a constructor that takes the first name, last name and social security number as arguments (lines 11–16); *get* methods that return the first name, last name and social security number (lines 25–28, 37–40 and 49–52, respectively); *set* methods that set the first name, last name and social security number (lines 19–22, 31–34 and 43–46, respectively); method `toString` (lines 55–60), which returns the `String` representation of an `Employee`; and `abstract` method `earnings` (line 63), which will be implemented by each of the concrete subclasses. The `Employee` constructor does not validate its parameters in this example; normally, such validation should be provided.

Why did we decide to declare `earnings` as an `abstract` method? It simply does not make sense to provide an implementation of this method in class `Employee`. We cannot calculate the earnings for a *general* `Employee`—we first must know the *specific* type of `Employee` to determine the appropriate earnings calculation. By declaring this method `abstract`, we indicate that each concrete subclass *must* provide an appropriate `earnings` implementation and that a program will be able to use superclass `Employee` variables to invoke method `earnings` polymorphically for any type of `Employee`.

```
1  // Fig. 10.4: Employee.java
2  // Employee abstract superclass.
3
4  public abstract class Employee
5  {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9
10    // three-argument constructor
11    public Employee( String first, String last, String ssn )
12    {
13       firstName = first;
14       lastName = last;
15       socialSecurityNumber = ssn;
16    } // end three-argument Employee constructor
17
18    // set first name
19    public void setFirstName( String first )
20    {
21       firstName = first; // should validate
22    } // end method setFirstName
23
24    // return first name
25    public String getFirstName()
26    {
27       return firstName;
28    } // end method getFirstName
29
```

Fig. 10.4 | `Employee` abstract superclass. (Part 1 of 2.)

```
    // set last name
    public void setLastName( String last )
    {
       lastName = last; // should validate
    } // end method setLastName

    // return last name
    public String getLastName()
    {
       return lastName;
    } // end method getLastName

    // set social security number
    public void setSocialSecurityNumber( String ssn )
    {
       socialSecurityNumber = ssn; // should validate
    } // end method setSocialSecurityNumber

    // return social security number
    public String getSocialSecurityNumber()
    {
       return socialSecurityNumber;
    } // end method getSocialSecurityNumber

    // return String representation of Employee object
    @Override
    public String toString()
    {
       return String.format( "%s %s\nsocial security number: %s",
          getFirstName(), getLastName(), getSocialSecurityNumber() );
    } // end method toString

    // abstract method overridden by concrete subclasses
    public abstract double earnings(); // no implementation here
 } // end abstract class Employee
```

Fig. 10.4 | `Employee` abstract superclass. (Part 2 of 2.)

10.5.2 Concrete Subclass SalariedEmployee

Class `SalariedEmployee` (Fig. 10.5) extends class `Employee` (line 4) and overrides abstract method `earnings` (lines 33–37), which makes `SalariedEmployee` a concrete class. The class includes a constructor (lines 9–14) that takes a first name, a last name, a social security number and a weekly salary as arguments; a *set* method to assign a new nonnegative value to instance variable `weeklySalary` (lines 17–24); a *get* method to return `weeklySalary`'s value (lines 27–30); a method `earnings` (lines 33–37) to calculate a `SalariedEmployee`'s earnings; and a method `toString` (lines 40–45), which returns a `String` including the employee's type, namely, `"salaried employee: "` followed by employee-specific information produced by superclass `Employee`'s `toString` method and `SalariedEmployee`'s `getWeeklySalary` method. Class `SalariedEmployee`'s constructor passes the first name, last name and social security number to the `Employee` constructor (line 12) to initialize the `private` instance variables not inherited from the superclass. Method `earn-`

`ings` overrides `Employee`'s abstract method `earnings` to provide a concrete implementation that returns the `SalariedEmployee`'s weekly salary. If we do not implement `earnings`, class `SalariedEmployee` must be declared `abstract`—otherwise, class `SalariedEmployee` will not compile. Of course, we want `SalariedEmployee` to be a concrete class in this example.

```java
// Fig. 10.5: SalariedEmployee.java
// SalariedEmployee concrete class extends abstract class Employee.

public class SalariedEmployee extends Employee
{
   private double weeklySalary;

   // four-argument constructor
   public SalariedEmployee( String first, String last, String ssn,
      double salary )
   {
      super( first, last, ssn ); // pass to Employee constructor
      setWeeklySalary( salary ); // validate and store salary
   } // end four-argument SalariedEmployee constructor

   // set salary
   public void setWeeklySalary( double salary )
   {
      if ( salary >= 0.0 )
         baseSalary = salary;
      else
         throw new IllegalArgumentException(
            "Weekly salary must be >= 0.0" );
   } // end method setWeeklySalary

   // return salary
   public double getWeeklySalary()
   {
      return weeklySalary;
   } // end method getWeeklySalary

   // calculate earnings; override abstract method earnings in Employee
   @Override
   public double earnings()
   {
      return getWeeklySalary();
   } // end method earnings

   // return String representation of SalariedEmployee object
   @Override
   public String toString()
   {
      return String.format( "salaried employee: %s\n%s: $%,.2f",
         super.toString(), "weekly salary", getWeeklySalary() );
   } // end method toString
} // end class SalariedEmployee
```

Fig. 10.5 | `SalariedEmployee` concrete class extends `abstract` class `Employee`.

Method `toString` (lines 40–45) overrides `Employee` method `toString`. If class `SalariedEmployee` did not override `toString`, `SalariedEmployee` would have inherited the `Employee` version of `toString`. In that case, `SalariedEmployee`'s `toString` method would simply return the employee's full name and social security number, which does not adequately represent a `SalariedEmployee`. To produce a complete `String` representation of a `SalariedEmployee`, the subclass's `toString` method returns `"salaried employee: "` followed by the superclass `Employee`-specific information (i.e., first name, last name and social security number) obtained by invoking the superclass's `toString` method (line 44)—this is a nice example of code reuse. The `String` representation of a `SalariedEmployee` also contains the employee's weekly salary obtained by invoking the class's `getWeeklySalary` method.

10.5.3 Concrete Subclass HourlyEmployee

Class `HourlyEmployee` (Fig. 10.6) also extends `Employee` (line 4). The class includes a constructor (lines 10–16) that takes as arguments a first name, a last name, a social security number, an hourly wage and the number of hours worked. Lines 19–26 and 35–42 declare *set* methods that assign new values to instance variables `wage` and `hours`, respectively. Method `setWage` (lines 19–26) ensures that `wage` is nonnegative, and method `setHours` (lines 35–42) ensures that `hours` is between 0 and 168 (the total number of hours in a week) inclusive. Class `HourlyEmployee` also includes *get* methods (lines 29–32 and 45–48) to return the values of `wage` and `hours`, respectively; a method `earnings` (lines 51–58) to calculate an `HourlyEmployee`'s earnings; and a method `toString` (lines 61–67), which returns a `String` containing the employee's type (`"hourly employee: "`) and the employee-specific information. The `HourlyEmployee` constructor, like the `SalariedEmployee` constructor, passes the first name, last name and social security number to the superclass `Employee` constructor (line 13) to initialize the `private` instance variables. In addition, method `toString` calls superclass method `toString` (line 65) to obtain the `Employee`-specific information (i.e., first name, last name and social security number)—this is another nice example of code reuse.

```
 1  // Fig. 10.6: HourlyEmployee.java
 2  // HourlyEmployee class extends Employee.
 3
 4  public class HourlyEmployee extends Employee
 5  {
 6     private double wage; // wage per hour
 7     private double hours; // hours worked for week
 8
 9     // five-argument constructor
10     public HourlyEmployee( String first, String last, String ssn,
11        double hourlyWage, double hoursWorked )
12     {
13        super( first, last, ssn );
14        setWage( hourlyWage ); // validate hourly wage
15        setHours( hoursWorked ); // validate hours worked
16     } // end five-argument HourlyEmployee constructor
17
```

Fig. 10.6 | `HourlyEmployee` class extends `Employee`. (Part 1 of 2.)

```
   // set wage
   public void setWage( double hourlyWage )
   {
      if ( hourlyWage >= 0.0 )
         wage = hourlyWage;
      else
         throw new IllegalArgumentException(
            "Hourly wage must be >= 0.0" );
   } // end method setWage

   // return wage
   public double getWage()
   {
      return wage;
   } // end method getWage

   // set hours worked
   public void setHours( double hoursWorked )
   {
      if ( ( hoursWorked >= 0.0 ) && ( hoursWorked <= 168.0 ) )
         hours = hoursWorked;
      else
         throw new IllegalArgumentException(
            "Hours worked must be >= 0.0 and <= 168.0" );
   } // end method setHours

   // return hours worked
   public double getHours()
   {
      return hours;
   } // end method getHours

   // calculate earnings; override abstract method earnings in Employee
   @Override
   public double earnings()
   {
      if ( getHours() <= 40 ) // no overtime
         return getWage() * getHours();
      else
         return 40 * getWage() + ( getHours() - 40 ) * getWage() * 1.5;
   } // end method earnings

   // return String representation of HourlyEmployee object
   @Override
   public String toString()
   {
      return String.format( "hourly employee: %s\n%s: $%,.2f; %s: %,.2f",
         super.toString(), "hourly wage", getWage(),
         "hours worked", getHours() );
   } // end method toString
} // end class HourlyEmployee
```

Fig. 10.6 | `HourlyEmployee` class extends `Employee`. (Part 2 of 2.)

10.5.4 Concrete Subclass CommissionEmployee

Class `CommissionEmployee` (Fig. 10.7) extends class `Employee` (line 4). The class includes a constructor (lines 10–16) that takes a first name, a last name, a social security number, a sales amount and a commission rate; *set* methods (lines 19–26 and 35–42) to assign new values to instance variables `commissionRate` and `grossSales`, respectively; *get* methods (lines 29–32 and 45–48) that retrieve the values of these instance variables; method `earnings` (lines 51–55) to calculate a `CommissionEmployee`'s earnings; and method `toString` (lines 58–65), which returns the employee's type, namely, `"commission employee: "` and employee-specific information. The constructor also passes the first name, last name and social security number to `Employee`'s constructor (line 13) to initialize `Employee`'s `private` instance variables. Method `toString` calls superclass method `toString` (line 62) to obtain the `Employee`-specific information (i.e., first name, last name and social security number).

```
1  // Fig. 10.7: CommissionEmployee.java
2  // CommissionEmployee class extends Employee.
3
4  public class CommissionEmployee extends Employee
5  {
6     private double grossSales; // gross weekly sales
7     private double commissionRate; // commission percentage
8
9     // five-argument constructor
10    public CommissionEmployee( String first, String last, String ssn,
11       double sales, double rate )
12    {
13       super( first, last, ssn );
14       setGrossSales( sales );
15       setCommissionRate( rate );
16    } // end five-argument CommissionEmployee constructor
17
18    // set commission rate
19    public void setCommissionRate( double rate )
20    {
21       if ( rate > 0.0 && rate < 1.0 )
22          commissionRate = rate;
23       else
24          throw new IllegalArgumentException(
25             "Commission rate must be > 0.0 and < 1.0" );
26    } // end method setCommissionRate
27
28    // return commission rate
29    public double getCommissionRate()
30    {
31       return commissionRate;
32    } // end method getCommissionRate
33
34    // set gross sales amount
35    public void setGrossSales( double sales )
36    {
```

Fig. 10.7 | `CommissionEmployee` class extends `Employee`. (Part 1 of 2.)

```java
      if ( sales >= 0.0 )
         grossSales = sales;
      else
         throw new IllegalArgumentException(
            "Gross sales must be >= 0.0" );
   } // end method setGrossSales

   // return gross sales amount
   public double getGrossSales()
   {
      return grossSales;
   } // end method getGrossSales

   // calculate earnings; override abstract method earnings in Employee
   @Override
   public double earnings()
   {
      return getCommissionRate() * getGrossSales();
   } // end method earnings

   // return String representation of CommissionEmployee object
   @Override
   public String toString()
   {
      return String.format( "%s: %s\n%s: $%,.2f; %s: %.2f",
         "commission employee", super.toString(),
         "gross sales", getGrossSales(),
         "commission rate", getCommissionRate() );
   } // end method toString
} // end class CommissionEmployee
```

Fig. 10.7 | `CommissionEmployee` class extends `Employee`. (Part 2 of 2.)

10.5.5 Indirect Concrete Subclass BasePlusCommissionEmployee

Class `BasePlusCommissionEmployee` (Fig. 10.8) extends class `CommissionEmployee` (line 4) and therefore is an *indirect* subclass of class `Employee`. Class `BasePlusCommissionEmployee` has a constructor (lines 9–14) that takes as arguments a first name, a last name, a social security number, a sales amount, a commission rate and a base salary. It then passes all of these except the base salary to the `CommissionEmployee` constructor (line 12) to initialize the inherited members. `BasePlusCommissionEmployee` also contains a *set* method (lines 17–24) to assign a new value to instance variable `baseSalary` and a *get* method (lines 27–30) to return `baseSalary`'s value. Method `earnings` (lines 33–37) calculates a `BasePlusCommissionEmployee`'s earnings. Line 36 in method `earnings` calls superclass `CommissionEmployee`'s `earnings` method to calculate the commission-based portion of the employee's earnings—this is another nice example of code reuse. `BasePlusCommissionEmployee`'s `toString` method (lines 40–46) creates a `String` representation of a `BasePlusCommissionEmployee` that contains `"base-salaried"`, followed by the `String` obtained by invoking superclass `CommissionEmployee`'s `toString` method (another example of code reuse), then the base salary. The result is a `String` beginning with `"base-salaried commission employee"` followed by the rest of the `BasePlusCommissionEmployee`'s information. Recall that `CommissionEm-`

ployee's toString obtains the employee's first name, last name and social security number by invoking the toString method of its superclass (i.e., Employee)—yet another example of code reuse. BasePlusCommissionEmployee's toString initiates a chain of method calls that span all three levels of the Employee hierarchy.

```java
// Fig. 10.8: BasePlusCommissionEmployee.java
// BasePlusCommissionEmployee class extends CommissionEmployee.

public class BasePlusCommissionEmployee extends CommissionEmployee
{
   private double baseSalary; // base salary per week

   // six-argument constructor
   public BasePlusCommissionEmployee( String first, String last,
      String ssn, double sales, double rate, double salary )
   {
      super( first, last, ssn, sales, rate );
      setBaseSalary( salary ); // validate and store base salary
   } // end six-argument BasePlusCommissionEmployee constructor

   // set base salary
   public void setBaseSalary( double salary )
   {
      if ( salary >= 0.0 )
         baseSalary = salary;
      else
         throw new IllegalArgumentException(
            "Base salary must be >= 0.0" );
   } // end method setBaseSalary

   // return base salary
   public double getBaseSalary()
   {
      return baseSalary;
   } // end method getBaseSalary

   // calculate earnings; override method earnings in CommissionEmployee
   @Override
   public double earnings()
   {
      return getBaseSalary() + super.earnings();
   } // end method earnings

   // return String representation of BasePlusCommissionEmployee object
   @Override
   public String toString()
   {
      return String.format( "%s %s; %s: $%,.2f",
         "base-salaried", super.toString(),
         "base salary", getBaseSalary() );
   } // end method toString
} // end class BasePlusCommissionEmployee
```

Fig. 10.8 | BasePlusCommissionEmployee class extends CommissionEmployee.

10.5.6 Polymorphic Processing, Operator `instanceof` and Downcasting

To test our `Employee` hierarchy, the application in Fig. 10.9 creates an object of each of the four concrete classes `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` and `BasePlusCommissionEmployee`. The program manipulates these objects nonpolymorphically, via variables of each object's own type, then polymorphically, using an array of `Employee` variables. While processing the objects polymorphically, the program increases the base salary of each `BasePlusCommissionEmployee` by 10%—this requires *determining the object's type at execution time*. Finally, the program polymorphically determines and outputs the type of each object in the `Employee` array. Lines 9–18 create objects of each of the four concrete `Employee` subclasses. Lines 22–30 output the `String` representation and earnings of each of these objects *nonpolymorphically*. Each object's `toString` method is called *implicitly* by `printf` when the object is output as a `String` with the `%s` format specifier.

```
1  // Fig. 10.9: PayrollSystemTest.java
2  // Employee hierarchy test program.
3
4  public class PayrollSystemTest
5  {
6     public static void main( String[] args )
7     {
8        // create subclass objects
9        SalariedEmployee salariedEmployee =
10          new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
11       HourlyEmployee hourlyEmployee =
12          new HourlyEmployee( "Karen", "Price", "222-22-2222", 16.75, 40 );
13       CommissionEmployee commissionEmployee =
14          new CommissionEmployee(
15          "Sue", "Jones", "333-33-3333", 10000, .06 );
16       BasePlusCommissionEmployee basePlusCommissionEmployee =
17          new BasePlusCommissionEmployee(
18          "Bob", "Lewis", "444-44-4444", 5000, .04, 300 );
19
20       System.out.println( "Employees processed individually:\n" );
21
22       System.out.printf( "%s\n%s: $%,.2f\n\n",
23          salariedEmployee, "earned", salariedEmployee.earnings() );
24       System.out.printf( "%s\n%s: $%,.2f\n\n",
25          hourlyEmployee, "earned", hourlyEmployee.earnings() );
26       System.out.printf( "%s\n%s: $%,.2f\n\n",
27          commissionEmployee, "earned", commissionEmployee.earnings() );
28       System.out.printf( "%s\n%s: $%,.2f\n\n",
29          basePlusCommissionEmployee,
30          "earned", basePlusCommissionEmployee.earnings() );
31
32       // create four-element Employee array
33       Employee[] employees = new Employee[ 4 ];
34
35       // initialize array with Employees
36       employees[ 0 ] = salariedEmployee;
37       employees[ 1 ] = hourlyEmployee;
```

Fig. 10.9 | `Employee` hierarchy test program. (Part 1 of 3.)

```java
      employees[ 2 ] = commissionEmployee;
      employees[ 3 ] = basePlusCommissionEmployee;

      System.out.println( "Employees processed polymorphically:\n" );

      // generically process each element in array employees
      for ( Employee currentEmployee : employees )
      {
         System.out.println( currentEmployee ); // invokes toString

         // determine whether element is a BasePlusCommissionEmployee
         if ( currentEmployee instanceof BasePlusCommissionEmployee )
         {
            // downcast Employee reference to
            // BasePlusCommissionEmployee reference
            BasePlusCommissionEmployee employee =
               ( BasePlusCommissionEmployee ) currentEmployee;

            employee.setBaseSalary( 1.10 * employee.getBaseSalary() );

            System.out.printf(
               "new base salary with 10%% increase is: $%,.2f\n",
               employee.getBaseSalary() );
         } // end if

         System.out.printf(
            "earned $%,.2f\n\n", currentEmployee.earnings() );
      } // end for

      // get type name of each object in employees array
      for ( int j = 0; j < employees.length; j++ )
         System.out.printf( "Employee %d is a %s\n", j,
            employees[ j ].getClass().getName() );
   } // end main
} // end class PayrollSystemTest
```

```
Employees processed individually:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned: $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned: $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned: $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
```

Fig. 10.9 | Employee hierarchy test program. (Part 2 of 3.)

```
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
earned: $500.00

Employees processed polymorphically:

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
earned $800.00

hourly employee: Karen Price
social security number: 222-22-2222
hourly wage: $16.75; hours worked: 40.00
earned $670.00

commission employee: Sue Jones
social security number: 333-33-3333
gross sales: $10,000.00; commission rate: 0.06
earned $600.00

base-salaried commission employee: Bob Lewis
social security number: 444-44-4444
gross sales: $5,000.00; commission rate: 0.04; base salary: $300.00
new base salary with 10% increase is: $330.00
earned $530.00

Employee 0 is a SalariedEmployee
Employee 1 is a HourlyEmployee
Employee 2 is a CommissionEmployee
Employee 3 is a BasePlusCommissionEmployee
```

Fig. 10.9 | `Employee` hierarchy test program. (Part 3 of 3.)

Creating the Array of `Employee`*s*

Line 33 declares `employees` and assigns it an array of four `Employee` variables. Line 36 assigns the reference to a `SalariedEmployee` object to `employees[0]`. Line 37 assigns the reference to an `HourlyEmployee` object to `employees[1]`. Line 38 assigns the reference to a `CommissionEmployee` object to `employees[2]`. Line 39 assigns the reference to a `BasePlusCommissionEmployee` object to `employee[3]`. These assignments are allowed, because a `SalariedEmployee` *is an* `Employee`, an `HourlyEmployee` *is an* `Employee`, a `CommissionEmployee` *is an* `Employee` and a `BasePlusCommissionEmployee` *is an* `Employee`. Therefore, we can assign the references of `SalariedEmployee`, `HourlyEmployee`, `CommissionEmployee` and `BasePlusCommissionEmployee` objects to superclass `Employee` variables, *even though* `Employee` *is an abstract class*.

Polymorphically Processing `Employee`*s*

Lines 44–65 iterate through array `employees` and invoke methods `toString` and `earnings` with `Employee` variable `currentEmployee`, which is assigned the reference to a different `Employee` in the array on each iteration. The output illustrates that the appropriate methods for each class are indeed invoked. All calls to method `toString` and `earnings` are resolved at execution time, based on the type of the object to which `currentEmployee` refers. This process is known as **dynamic binding** or **late binding**. For example, line 46 *implicitly* invokes method `toString` of the object to which `currentEmployee` refers. As a result of dynamic binding, Java decides which class's `toString` method to call *at execution time rather than at compile time*. Only the methods of class `Employee` can be called via an `Em-`

`ployee` variable (and `Employee`, of course, includes the methods of class `Object`). A superclass reference can be used to invoke only methods of the superclass—the subclass method implementations are invoked polymorphically.

Performing Type-Specific Operations on **`BasePlusCommissionEmployee`***s*

We perform special processing on `BasePlusCommissionEmployee` objects—as we encounter these objects at execution time, we increase their base salary by 10%. When processing objects polymorphically, we typically do not need to worry about the "specifics," but to adjust the base salary, we *do* have to determine the specific type of `Employee` object at execution time. Line 49 uses the **`instanceof`** operator to determine whether a particular `Employee` object's type is `BasePlusCommissionEmployee`. The condition in line 49 is true if the object referenced by `currentEmployee` *is a* `BasePlusCommissionEmployee`. This would also be true for any object of a `BasePlusCommissionEmployee` subclass because of the *is-a* relationship a subclass has with its superclass. Lines 53–54 downcast `currentEmployee` from type `Employee` to type `BasePlusCommissionEmployee`—this cast is allowed only if the object has an *is-a* relationship with `BasePlusCommissionEmployee`. The condition at line 49 ensures that this is the case. This cast is required if we're to invoke subclass `BasePlusCommissionEmployee` methods `getBaseSalary` and `setBaseSalary` on the current `Employee` object—as you'll see momentarily, *attempting to invoke a subclass-only method directly on a superclass reference is a compilation error.*

> **Common Programming Error 10.3**
>
> *Assigning a superclass variable to a subclass variable (without an explicit cast) is a compilation error.*

> **Software Engineering Observation 10.4**
>
> *If a subclass object's reference has been assigned to a variable of one of its direct or indirect superclasses at execution time, it's acceptable to downcast the reference stored in that superclass variable back to a subclass-type reference. Before performing such a cast, use the* `instanceof` *operator to ensure that the object is indeed an object of an appropriate subclass.*

> **Common Programming Error 10.4**
>
> *When downcasting a reference, a* `ClassCastException` *occurs if the referenced object at execution time does not have an* is-a *relationship with the type specified in the cast operator.*

If the `instanceof` expression in line 49 is `true`, lines 53–60 perform the special processing required for the `BasePlusCommissionEmployee` object. Using `BasePlusCommissionEmployee` variable `employee`, line 56 invokes subclass-only methods `getBaseSalary` and `setBaseSalary` to retrieve and update the employee's base salary with the 10% raise.

Calling **`earnings`** *Polymorphically*

Lines 63–64 invoke method `earnings` on `currentEmployee`, which polymorphically calls the appropriate subclass object's `earnings` method. Obtaining the earnings of the `SalariedEmployee`, `HourlyEmployee` and `CommissionEmployee` polymorphically in lines 63–64 produces the same results as obtaining these employees' earnings individually in lines 22–27. The earnings amount obtained for the `BasePlusCommissionEmployee` in lines 63–64 is higher than that obtained in lines 28–30, due to the 10% increase in its base salary.

Using Reflection to Get Each **`Employee`***'s Class Name*

Lines 68–70 display each employee's type as a `String`, using basic features of Java's so-called reflection capabilities. Every object knows its own class and can access this information through the **`getClass`** method, which all classes inherit from class `Object`. Method `getClass` returns an object of type **`Class`** (from package `java.lang`), which contains information about the object's type, including its class name. Line 70 invokes `getClass` on the current object to get its runtime class. The result of the `getClass` call is used to invoke **`getName`** to get the object's class name.

Avoiding Compilation Errors with Downcasting

In the previous example, we avoided several compilation errors by downcasting an `Employee` variable to a `BasePlusCommissionEmployee` variable in lines 53–54. If you remove the cast operator `(BasePlusCommissionEmployee)` from line 54 and attempt to assign `Employee` variable `currentEmployee` directly to `BasePlusCommissionEmployee` variable `employee`, you'll receive an "`incompatible types`" compilation error. This error indicates that the attempt to assign the reference of superclass object `currentEmployee` to subclass variable `employee` is not allowed. The compiler prevents this assignment because a `CommissionEmployee` is not a `BasePlusCommissionEmployee`—*the* is-a *relationship applies only between the subclass and its superclasses, not vice versa.*

Similarly, if lines 56 and 60 used superclass variable `currentEmployee` to invoke subclass-only methods `getBaseSalary` and `setBaseSalary`, we'd receive "`cannot find symbol`" compilation errors at these lines. Attempting to invoke subclass-only methods via a superclass variable is not allowed—even though lines 56 and 60 execute only if `instanceof` in line 49 returns `true` to indicate that `currentEmployee` holds a reference to a `BasePlusCommissionEmployee` object. Using a superclass `Employee` variable, we can invoke only methods found in class `Employee`—`earnings`, `toString` and `Employee`'s *get* and *set* methods.

Software Engineering Observation 10.5

Although the actual method that's called depends on the runtime type of the object to which a variable refers, a variable can be used to invoke only those methods that are members of that variable's type, which the compiler verifies.

10.5.7 Summary of the Allowed Assignments Between Superclass and Subclass Variables

Now that you've seen a complete application that processes diverse subclass objects polymorphically, we summarize what you can and cannot do with superclass and subclass objects and variables. Although a subclass object also *is a* superclass object, the two objects are nevertheless different. As discussed previously, subclass objects can be treated as objects of their superclass. But because the subclass can have additional subclass-only members, assigning a superclass reference to a subclass variable is not allowed without an explicit cast—such an assignment would leave the subclass members undefined for the superclass object.

We've discussed four ways to assign superclass and subclass references to variables of superclass and subclass types:

1. Assigning a superclass reference to a superclass variable is straightforward.
2. Assigning a subclass reference to a subclass variable is straightforward.

3. Assigning a subclass reference to a superclass variable is safe, because the subclass object *is an* object of its superclass. However, the superclass variable can be used to refer *only* to superclass members. If this code refers to subclass-only members through the superclass variable, the compiler reports errors.
4. Attempting to assign a superclass reference to a subclass variable is a compilation error. To avoid this error, the superclass reference must be cast to a subclass type explicitly. At *execution time*, if the object to which the reference refers is *not* a subclass object, an exception will occur. (For more on exception handling, see Chapter 11.) You should use the `instanceof` operator to ensure that such a cast is performed only if the object is a subclass object.

10.6 `final` Methods and Classes

We saw in Sections 6.3 and 6.9 that variables can be declared `final` to indicate that they cannot be modified after they're initialized—such variables represent constant values. It's also possible to declare methods, method parameters and classes with the `final` modifier.

Final Methods Cannot Be Overridden

A **`final` method** in a superclass *cannot* be overridden in a subclass—this guarantees that the `final` method implementation will be used by all direct and indirect subclasses in the hierarchy. Methods that are declared `private` are implicitly `final`, because it's not possible to override them in a subclass. Methods that are declared `static` are also implicitly `final`. A `final` method's declaration can never change, so all subclasses use the same method implementation, and calls to `final` methods are resolved at compile time—this is known as **static binding**.

Final Classes Cannot Be Superclasses

A **`final` class** that's declared `final` cannot be a superclass (i.e., a class cannot extend a `final` class). All methods in a `final` class are implicitly `final`. Class `String` is an example of a `final` class. If you were allowed to create a subclass of `String`, objects of that subclass could be used wherever `String`s are expected. Since class `String` cannot be extended, programs that use `String`s can rely on the functionality of `String` objects as specified in the Java API. Making the class `final` also prevents programmers from creating subclasses that might bypass security restrictions. For more insights on the use of keyword `final`, visit

```
download.oracle.com/javase/tutorial/java/IandI/final.html
```

and

```
www.ibm.com/developerworks/java/library/j-jtp1029.html
```

Common Programming Error 10.5

Attempting to declare a subclass of a `final` class is a compilation error.

Software Engineering Observation 10.6

In the Java API, the vast majority of classes are not *declared `final`. This enables inheritance and polymorphism. However, in some cases, it's important to declare classes `final`—typically for security reasons.*

10.7 Case Study: Creating and Using Interfaces

Our next example (Figs. 10.11–10.15) reexamines the payroll system of Section 10.5. Suppose that the company involved wishes to perform several accounting operations in a single accounts payable application—in addition to calculating the earnings that must be paid to each employee, the company must also calculate the payment due on each of several invoices (i.e., bills for goods purchased). Though applied to unrelated things (i.e., employees and invoices), both operations have to do with obtaining some kind of payment amount. For an employee, the payment refers to the employee's earnings. For an invoice, the payment refers to the total cost of the goods listed on the invoice. Can we calculate such *different* things as the payments due for employees and invoices in *a single* application polymorphically? Does Java offer a capability requiring that *unrelated* classes implement a set of *common* methods (e.g., a method that calculates a payment amount)? Java **interfaces** offer exactly this capability.

Standardizing Interactions

Interfaces define and standardize the ways in which things such as people and systems can interact with one another. For example, the controls on a radio serve as an interface between radio users and a radio's internal components. The controls allow users to perform only a limited set of operations (e.g., change the station, adjust the volume, choose between AM and FM), and different radios may implement the controls in different ways (e.g., using push buttons, dials, voice commands). The interface specifies *what* operations a radio must permit users to perform but does not specify *how* the operations are performed.

Software Objects Communicate Via Interfaces

Software objects also communicate via interfaces. A Java interface describes a set of methods that can be called on an object to tell it, for example, to perform some task or return some piece of information. The next example introduces an interface named `Payable` to describe the functionality of any object that must be capable of being paid and thus must offer a method to determine the proper payment amount due. An **interface declaration** begins with the keyword **`interface`** and contains only constants and `abstract` methods. Unlike classes, all interface members must be `public`, and *interfaces may not specify any implementation details*, such as concrete method declarations and instance variables. All methods declared in an interface are implicitly `public abstract` methods, and all fields are implicitly `public`, `static` and `final`. [*Note:* As of Java SE 5, it became a better programming practice to declare sets of constants as enumerations with keyword `enum`. See Section 6.9 for an introduction to `enum` and Section 8.9 for additional `enum` details.]

> **Good Programming Practice 10.1**
>
> *According to Chapter 9 of the* Java Language Specification*, it's proper style to declare an interface's methods without keywords* `public` *and* `abstract`*, because they're redundant in interface method declarations. Similarly, constants should be declared without keywords* `public`*,* `static` *and* `final`*, because they, too, are redundant.*

Using an Interface

To use an interface, a concrete class must specify that it **`implements`** the interface and must declare each method in the interface with the signature specified in the interface declaration. To specify that a class implements an interface add the `implements` keyword and the name of the interface to the end of your class declaration's first line. A class that does not implement *all* the methods of the interface is an *abstract* class and must be declared `abstract`. Implementing an interface is like signing a *contract* with the compiler that states, "I will declare all the methods specified by the interface or I will declare my class `abstract`."

Common Programming Error 10.6

Failing to implement any method of an interface in a concrete class that `implements` *the interface results in a compilation error indicating that the class must be declared* `abstract`*.*

Relating Disparate Types

An interface is often used when disparate (i.e., unrelated) classes need to share common methods and constants. This allows objects of unrelated classes to be processed polymorphically—objects of classes that implement the same interface can respond to the same method calls. You can create an interface that describes the desired functionality, then implement this interface in any classes that require that functionality. For example, in the accounts payable application developed in this section, we implement interface `Payable` in any class that must be able to calculate a payment amount (e.g., `Employee`, `Invoice`).

Interfaces vs. Abstract Classes

An interface is often used in place of an `abstract` *class when there's no default implementation to inherit*—that is, no fields and no default method implementations. Like `public abstract` classes, interfaces are typically `public` types. Like a `public` class, a `public` interface must be declared in a file with the same name as the interface and the `.java` file-name extension.

Tagging Interfaces

We'll see in Chapter 17, Files, Streams and Object Serialization, the notion of "tagging interfaces"—empty interfaces that have *no* methods or constant values. They're used to add *is-a* relationships to classes. For example, in Chapter 17 we'll discuss a mechanism called object serialization, which can convert objects to byte representations and can convert those byte representations back to objects. To enable this mechanism to work with your objects, you simply have to mark them as `Serializable` by adding `implements Serializable` to the end of your class declaration's first line. Then, all the objects of your class have the *is-a* relationship with `Serializable`.

10.7.1 Developing a `Payable` Hierarchy

To build an application that can determine payments for employees and invoices alike, we first create interface `Payable`, which contains method `getPaymentAmount` that returns a `double` amount that must be paid for an object of any class that implements the interface. Method `getPaymentAmount` is a general-purpose version of method `earnings` of the `Employee` hierarchy—method `earnings` calculates a payment amount specifically for an `Employee`, while `getPaymentAmount` can be applied to a broad range of unrelated objects. After declaring interface `Payable`, we introduce class `Invoice`, which `implements` interface `Payable`. We then modify class `Employee` such that it also implements interface `Payable`.

Finally, we update `Employee` subclass `SalariedEmployee` to "fit" into the `Payable` hierarchy by renaming `SalariedEmployee` method `earnings` as `getPaymentAmount`.

> **Good Programming Practice 10.2**
> *When declaring a method in an interface, choose a method name that describes the method's purpose in a* general *manner, because the method may be implemented by many unrelated classes.*

Classes `Invoice` and `Employee` both represent things for which the company must be able to calculate a payment amount. Both classes implement the `Payable` interface, so a program can invoke method `getPaymentAmount` on `Invoice` objects and `Employee` objects alike. As we'll soon see, this enables the polymorphic processing of `Invoices` and `Employees` required for the company's accounts payable application.

The UML class diagram in Fig. 10.10 shows the hierarchy used in our accounts payable application. The hierarchy begins with interface `Payable`. The UML distinguishes an interface from other classes by placing the word "interface" in guillemets (« and ») above the interface name. The UML expresses the relationship between a class and an interface through a relationship known as **realization**. A class is said to "realize," or implement, the methods of an interface. A class diagram models a realization as a dashed arrow with a hollow arrowhead pointing from the implementing class to the interface. The diagram in Fig. 10.10 indicates that classes `Invoice` and `Employee` each realize (i.e., implement) interface `Payable`. As in the class diagram of Fig. 10.2, class `Employee` appears in italics, indicating that it's an abstract class. Concrete class `SalariedEmployee` extends `Employee` and *inherits its superclass's realization relationship* with interface `Payable`.

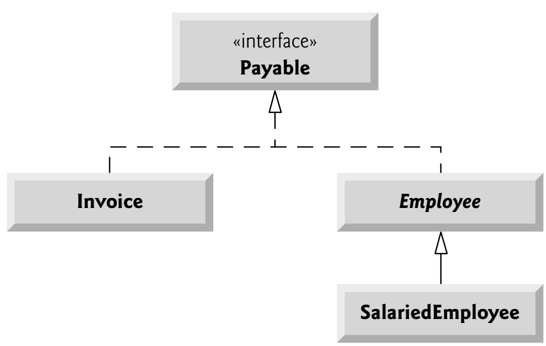


Fig. 10.10 | `Payable` interface hierarchy UML class diagram.

10.7.2 Interface `Payable`

The declaration of interface `Payable` begins in Fig. 10.11 at line 4. Interface `Payable` contains `public abstract` method `getPaymentAmount` (line 6). The method is not explicitly declared `public` or `abstract`. Interface methods are always `public` and `abstract`, so they do not need to be declared as such. Interface `Payable` has only one method—interfaces can have any number of methods. In addition, method `getPaymentAmount` has no parameters, but interface methods *can* have parameters. Interfaces may also contain fields that are implicitly `final` and `static`.

```
// Fig. 10.11: Payable.java
// Payable interface declaration.

public interface Payable
{
   double getPaymentAmount(); // calculate payment; no implementation
} // end interface Payable
```

Fig. 10.11 | `Payable` interface declaration.

10.7.3 Class Invoice

We now create class `Invoice` (Fig. 10.12) to represent a simple invoice that contains billing information for only one kind of part. The class declares `private` instance variables `partNumber`, `partDescription`, `quantity` and `pricePerItem` (in lines 6–9) that indicate the part number, a description of the part, the quantity of the part ordered and the price per item. Class `Invoice` also contains a constructor (lines 12–19), *get* and *set* methods (lines 22–74) that manipulate the class's instance variables and a `toString` method (lines 77–83) that returns a `String` representation of an `Invoice` object. Methods `setQuantity` (lines 46–52) and `setPricePerItem` (lines 61–68) ensure that `quantity` and `pricePerItem` obtain only nonnegative values.

Line 4 indicates that class `Invoice` implements interface `Payable`. Like all classes, class `Invoice` also implicitly extends `Object`. Java does not allow subclasses to inherit from more than one superclass, but it allows a class to inherit from one superclass and implement as many interfaces as it needs. To implement more than one interface, use a comma-separated list of interface names after keyword `implements` in the class declaration, as in:

```
public class ClassName extends SuperclassName implements FirstInterface,
   SecondInterface, …
```

Software Engineering Observation 10.7

All objects of a class that implement multiple interfaces have the is-a *relationship with each implemented interface type.*

```
1  // Fig. 10.12: Invoice.java
2  // Invoice class that implements Payable.
3
4  public class Invoice implements Payable
5  {
6     private String partNumber;
7     private String partDescription;
8     private int quantity;
9     private double pricePerItem;
10
11    // four-argument constructor
12    public Invoice( String part, String description, int count,
13       double price )
14    {
15       partNumber = part;
```

Fig. 10.12 | `Invoice` class that implements `Payable`. (Part 1 of 3.)

```java
      partDescription = description;
      setQuantity( count ); // validate and store quantity
      setPricePerItem( price ); // validate and store price per item
   } // end four-argument Invoice constructor

   // set part number
   public void setPartNumber( String part )
   {
      partNumber = part; // should validate
   } // end method setPartNumber

   // get part number
   public String getPartNumber()
   {
      return partNumber;
   } // end method getPartNumber

   // set description
   public void setPartDescription( String description )
   {
      partDescription = description; // should validate
   } // end method setPartDescription

   // get description
   public String getPartDescription()
   {
      return partDescription;
   } // end method getPartDescription

   // set quantity
   public void setQuantity( int count )
   {
      if ( count >= 0 )
         quantity = count;
      else
         throw new IllegalArgumentException( "Quantity must be >= 0" );
   } // end method setQuantity

   // get quantity
   public int getQuantity()
   {
      return quantity;
   } // end method getQuantity

   // set price per item
   public void setPricePerItem( double price )
   {
      if ( price >= 0.0 )
         pricePerItem = price;
      else
         throw new IllegalArgumentException(
            "Price per item must be >= 0" );
   } // end method setPricePerItem
```

Fig. 10.12 | Invoice class that implements Payable. (Part 2 of 3.)

```
69
70      // get price per item
71      public double getPricePerItem()
72      {
73         return pricePerItem;
74      } // end method getPricePerItem
75
76      // return String representation of Invoice object
77      @Override
78      public String toString()
79      {
80         return String.format( "%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
81            "invoice", "part number", getPartNumber(), getPartDescription(),
82            "quantity", getQuantity(), "price per item", getPricePerItem() );
83      } // end method toString
84
85      // method required to carry out contract with interface Payable
86      @Override
87      public double getPaymentAmount()
88      {
89         return getQuantity() * getPricePerItem(); // calculate total cost
90      } // end method getPaymentAmount
91   } // end class Invoice
```

Fig. 10.12 | `Invoice` class that implements `Payable`. (Part 3 of 3.)

Class `Invoice` implements the one method in interface `Payable`—method `getPaymentAmount` is declared in lines 86–90. The method calculates the total payment required to pay the invoice. The method multiplies the values of `quantity` and `pricePerItem` (obtained through the appropriate *get* methods) and returns the result (line 89). This method satisfies the implementation requirement for this method in interface `Payable`—we've fulfilled the interface contract with the compiler.

10.7.4 Modifying Class `Employee` to Implement Interface `Payable`

We now modify class `Employee` such that it implements interface `Payable`. Figure 10.13 contains the modified class, which is identical to that of Fig. 10.4 with two exceptions. First, line 4 of Fig. 10.13 indicates that class `Employee` now `implements` interface `Payable`. So we must rename `earnings` to `getPaymentAmount` throughout the `Employee` hierarchy. As with method `earnings` in the version of class `Employee` in Fig. 10.4, however, it does not make sense to implement method `getPaymentAmount` in class `Employee` because we cannot calculate the earnings payment owed to a general `Employee`—we must first know the specific type of `Employee`. In Fig. 10.4, we declared method `earnings` as `abstract` for this reason, so class `Employee` had to be declared `abstract`. This forced each `Employee` concrete subclass to override `earnings` with an implementation.

In Fig. 10.13, we handle this situation differently. Recall that when a class implements an interface, it makes a *contract* with the compiler stating either that the class will implement *each* of the methods in the interface or that the class will be declared `abstract`. If the latter option is chosen, we do not need to declare the interface methods as `abstract` in the `abstract` class—they're already implicitly declared as such in the interface. Any concrete subclass of the `abstract` class must implement the interface methods to fulfill the superclass's contract with the compiler. If the subclass does not do so, it too must be declared `abstract`. As indicated by the comments in lines 62–63, class `Employee` of Fig. 10.13 does *not* implement method `getPaymentAmount`, so the class is declared `abstract`. Each direct `Employee` subclass *inherits the superclass's contract* to implement method `getPaymentAmount` and thus must implement this method to become a concrete class for which objects can be instantiated. A class that extends one of `Employee`'s concrete subclasses will inherit an implementation of `getPaymentAmount` and thus will also be a concrete class.

```java
1  // Fig. 10.13: Employee.java
2  // Employee abstract superclass that implements Payable.
3
4  public abstract class Employee implements Payable
5  {
6     private String firstName;
7     private String lastName;
8     private String socialSecurityNumber;
9
10    // three-argument constructor
11    public Employee( String first, String last, String ssn )
12    {
13       firstName = first;
14       lastName = last;
15       socialSecurityNumber = ssn;
16    } // end three-argument Employee constructor
17
18    // set first name
19    public void setFirstName( String first )
20    {
21       firstName = first; // should validate
22    } // end method setFirstName
23
24    // return first name
25    public String getFirstName()
26    {
27       return firstName;
28    } // end method getFirstName
29
30    // set last name
31    public void setLastName( String last )
32    {
33       lastName = last; // should validate
34    } // end method setLastName
35
36    // return last name
37    public String getLastName()
38    {
39       return lastName;
40    } // end method getLastName
41
```

Fig. 10.13 | `Employee` class that implements `Payable`. (Part 1 of 2.)

```java
   // set social security number
   public void setSocialSecurityNumber( String ssn )
   {
      socialSecurityNumber = ssn; // should validate
   } // end method setSocialSecurityNumber

   // return social security number
   public String getSocialSecurityNumber()
   {
      return socialSecurityNumber;
   } // end method getSocialSecurityNumber

   // return String representation of Employee object
   @Override
   public String toString()
   {
      return String.format( "%s %s\nsocial security number: %s",
         getFirstName(), getLastName(), getSocialSecurityNumber() );
   } // end method toString

   // Note: We do not implement Payable method getPaymentAmount here so
   // this class must be declared abstract to avoid a compilation error.
} // end abstract class Employee
```

Fig. 10.13 | Employee class that implements Payable. (Part 2 of 2.)

10.7.5 Modifying Class SalariedEmployee for Use in the Payable Hierarchy

Figure 10.14 contains a modified SalariedEmployee class that extends Employee and fulfills superclass Employee's contract to implement Payable method getPaymentAmount. This version of SalariedEmployee is identical to that of Fig. 10.5, but it replaces method earnings with method getPaymentAmount (lines 34–38). Recall that the Payable version of the method has a more *general* name to be applicable to possibly *disparate* classes. The remaining Employee subclasses (e.g., HourlyEmployee, CommissionEmployee and BasePlusCommissionEmployee) also must be modified to contain method getPaymentAmount in place of earnings to reflect the fact that Employee now implements Payable. We leave these modifications as an exercise.

```java
1  // Fig. 10.14: SalariedEmployee.java
2  // SalariedEmployee class extends Employee, which implements Payable.
3
4  public class SalariedEmployee extends Employee
5  {
6     private double weeklySalary;
7
```

Fig. 10.14 | SalariedEmployee class that implements interface Payable method getPaymentAmount. (Part 1 of 2.)

```java
   // four-argument constructor
   public SalariedEmployee( String first, String last, String ssn,
      double salary )
   {
      super( first, last, ssn ); // pass to Employee constructor
      setWeeklySalary( salary ); // validate and store salary
   } // end four-argument SalariedEmployee constructor

   // set salary
   public void setWeeklySalary( double salary )
   {
      if ( salary >= 0.0 )
         baseSalary = salary;
      else
         throw new IllegalArgumentException(
            "Weekly salary must be >= 0.0" );
   } // end method setWeeklySalary

   // return salary
   public double getWeeklySalary()
   {
      return weeklySalary;
   } // end method getWeeklySalary

   // calculate earnings; implement interface Payable method that was
   // abstract in superclass Employee
   @Override
   public double getPaymentAmount()
   {
      return getWeeklySalary();
   } // end method getPaymentAmount

   // return String representation of SalariedEmployee object
   @Override
   public String toString()
   {
      return String.format( "salaried employee: %s\n%s: $%,.2f",
         super.toString(), "weekly salary", getWeeklySalary() );
   } // end method toString
} // end class SalariedEmployee
```

Fig. 10.14 | `SalariedEmployee` class that implements interface `Payable` method `getPaymentAmount`. (Part 2 of 2.)

When a class implements an interface, the same *is-a* relationship provided by inheritance applies. Class `Employee` implements `Payable`, so we can say that an `Employee` *is a* `Payable`. In fact, objects of any classes that extend `Employee` are also `Payable` objects. `SalariedEmployee` objects, for instance, are `Payable` objects. Objects of any subclasses of the class that `implements` the interface can also be thought of as objects of the interface type. Thus, just as we can assign the reference of a `SalariedEmployee` object to a superclass `Employee` variable, we can assign the reference of a `SalariedEmployee` object to an interface `Payable` variable. `Invoice implements Payable`, so an `Invoice` object also *is a* `Payable` object, and we can assign the reference of an `Invoice` object to a `Payable` variable.

Software Engineering Observation 10.8

When a method parameter is declared with a superclass or interface type, the method processes the object received as an argument polymorphically.

Software Engineering Observation 10.9

Using a superclass reference, we can polymorphically invoke any method declared in the superclass and its superclasses (e.g., class `Object`). Using an interface reference, we can polymorphically invoke any method declared in the interface, its superinterfaces (one interface can extend another) and in class `Object`—a variable of an interface type must refer to an object to call methods, and all objects have the methods of class `Object`.

10.7.6 Using Interface `Payable` to Process `Invoices` and `Employees` Polymorphically

`PayableInterfaceTest` (Fig. 10.15) illustrates that interface `Payable` can be used to process a set of `Invoices` and `Employees` polymorphically in a single application. Line 9 declares `payableObjects` and assigns it an array of four `Payable` variables. Lines 12–13 assign the references of `Invoice` objects to the first two elements of `payableObjects`. Lines 14–17 then assign the references of `SalariedEmployee` objects to the remaining two elements of `payableObjects`. These assignments are allowed because an `Invoice` *is a* `Payable`, a `SalariedEmployee` *is an* `Employee` and an `Employee` *is a* `Payable`. Lines 23–29 use the enhanced `for` statement to polymorphically process each `Payable` object in `payableObjects`, printing the object as a `String`, along with the payment amount due. Line 27 invokes method `toString` via a `Payable` interface reference, even though `toString` is not declared in interface `Payable`—*all references (including those of interface types) refer to objects that extend `Object` and therefore have a `toString` method.* (Method `toString` also can be invoked *implicitly* here.) Line 28 invokes `Payable` method `getPaymentAmount` to obtain the payment amount for each object in `payableObjects`, regardless of the actual type of the object. The output reveals that the method calls in lines 27–28 invoke the appropriate class's implementation of methods `toString` and `getPaymentAmount`. For instance, when `currentPayable` refers to an `Invoice` during the first iteration of the `for` loop, class `Invoice`'s `toString` and `getPaymentAmount` execute.

```
1  // Fig. 10.15: PayableInterfaceTest.java
2  // Tests interface Payable.
3
4  public class PayableInterfaceTest
5  {
6     public static void main( String[] args )
7     {
8        // create four-element Payable array
9        Payable[] payableObjects = new Payable[ 4 ];
```

Fig. 10.15 | `Payable` interface test program processing `Invoices` and `Employees` polymorphically. (Part 1 of 2.)

```java

      // populate array with objects that implement Payable
      payableObjects[ 0 ] = new Invoice( "01234", "seat", 2, 375.00 );
      payableObjects[ 1 ] = new Invoice( "56789", "tire", 4, 79.95 );
      payableObjects[ 2 ] =
         new SalariedEmployee( "John", "Smith", "111-11-1111", 800.00 );
      payableObjects[ 3 ] =
         new SalariedEmployee( "Lisa", "Barnes", "888-88-8888", 1200.00 );

      System.out.println(
         "Invoices and Employees processed polymorphically:\n" );

      // generically process each element in array payableObjects
      for ( Payable currentPayable : payableObjects )
      {
         // output currentPayable and its appropriate payment amount
         System.out.printf( "%s \n%s: $%,.2f\n\n",
            currentPayable.toString(),
            "payment due", currentPayable.getPaymentAmount() );
      } // end for
   } // end main
} // end class PayableInterfaceTest
```

```
Invoices and Employees processed polymorphically:

invoice:
part number: 01234 (seat)
quantity: 2
price per item: $375.00
payment due: $750.00

invoice:
part number: 56789 (tire)
quantity: 4
price per item: $79.95
payment due: $319.80

salaried employee: John Smith
social security number: 111-11-1111
weekly salary: $800.00
payment due: $800.00

salaried employee: Lisa Barnes
social security number: 888-88-8888
weekly salary: $1,200.00
payment due: $1,200.00
```

Fig. 10.15 | `Payable` interface test program processing `Invoices` and `Employees` polymorphically. (Part 2 of 2.)

10.7.7 Common Interfaces of the Java API

In this section, we overview several common interfaces found in the Java API. The power and flexibility of interfaces is used frequently throughout the Java API. These interfaces are implemented and used in the same manner as the interfaces you create (e.g., interface

`Payable` in Section 10.7.2). The Java API's interfaces enable you to use your own classes within the frameworks provided by Java, such as comparing objects of your own types and creating tasks that can execute concurrently with other tasks in the same program. Figure 10.16 overviews a few of the more popular interfaces of the Java API that we use in *Java for Programmers, 2/e*.

Interface	Description
`Comparable`	Java contains several comparison operators (e.g., `<`, `<=`, `>`, `>=`, `==`, `!=`) that allow you to compare primitive values. However, these operators *cannot* be used to compare objects. Interface `Comparable` is used to allow objects of a class that `implements` the interface to be compared to one another. Interface `Comparable` is commonly used for ordering objects in a collection such as an array. We use `Comparable` in Chapter 18, Generic Collections, and Chapter 19, Generic Classes and Methods.
`Serializable`	An interface used to identify classes whose objects can be written to (i.e., serialized) or read from (i.e., deserialized) some type of storage (e.g., file on disk, database field) or transmitted across a network. We use `Serializable` in Chapter 17, Files, Streams and Object Serialization, and Chapter 24, Networking.
`Runnable`	Implemented by any class for which objects of that class should be able to execute in parallel using a technique called multithreading (discussed in Chapter 23, Multithreading). The interface contains one method, `run`, which describes the behavior of an object when executed.
GUI event-listener interfaces	You work with graphical user interfaces (GUIs) every day. In your web browser, you might type the address of a website to visit, or you might click a button to return to a previous site. The browser responds to your interaction and performs the desired task. Your interaction is known as an event, and the code that the browser uses to respond to an event is known as an event handler. In Chapter 14, GUI Components: Part 1, and Chapter 22, GUI Components: Part 2, you'll learn how to build GUIs and event handlers that respond to user interactions. Event handlers are declared in classes that implement an appropriate event-listener interface. Each event-listener interface specifies one or more methods that must be implemented to respond to user interactions.
`SwingConstants`	Contains a set of constants used in GUI programming to position GUI elements on the screen. We explore GUI programming in Chapters 14 and 22.

Fig. 10.16 | Common interfaces of the Java API.

10.8 Wrap-Up

This chapter introduced polymorphism—the ability to process objects that share the same superclass in a class hierarchy as if they're all objects of the superclass. The chapter discussed how polymorphism makes systems extensible and maintainable, then demonstrated how to use overridden methods to effect polymorphic behavior. We introduced abstract classes, which allow you to provide an appropriate superclass from which other classes can inherit. You learned that an abstract class can declare abstract methods that each subclass must implement to become a concrete class and that a program can use variables of an abstract class to invoke the subclasses' implementations of abstract methods polymorphically. You also learned how to determine an object's type at execution time. We discussed the concepts of `final` methods and classes. Finally, the chapter discussed declaring and implementing an interface as another way to achieve polymorphic behavior.

You should now be familiar with classes, objects, encapsulation, inheritance, interfaces and polymorphism—the most essential aspects of object-oriented programming.

In the next chapter, you'll learn about exceptions, useful for handling errors during a program's execution. Exception handling provides for more robust programs.

Index

Symbols

Numerics

A

B

C

D

E

F

H

M

N

O

S

V

W

X

Y

Z