REVISED AND UPDATED FOR JAVA SE 7

PRENTICE HALL

CORE JAVA™

Volume I—Fundamentals

NINTH EDITION

CAY S. HORSTMANN · GARY CORNELL

Core Java™

Volume I—Fundamentals

Ninth Edition

Core Java™

Volume I—Fundamentals

Ninth Edition

Cay S. Horstmann
Gary Cornell

PRENTICE HALL

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

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 United States, please contact:

International Sales
international@pearson.com

Visit us on the Web: informit.com/ph

Library of Congress Cataloging-in-Publication Data

Horstmann, Cay S., 1959-
Core Java / Cay S. Horstmann, Gary Cornell.—Ninth edition.
pages cm
Includes index.
ISBN 978-0-13-708189-9 (v. 1 : pbk. : alk. paper) 1. Java
(Computer program language) I. Cornell, Gary. II. Title.
QA76.73.J38H6753 2013
005.13'3—dc23

2012035397

ISBN-13: 978-0-13-708189-9
ISBN-10: 0-13-708189-8

Text printed in the United States on recycled paper at Edwards Brothers Malloy in Ann Arbor, Michigan.
Second printing, July 2013

Contents

This page intentionally left blank

Preface

To the Reader

In late 1995, the Java programming language burst onto the Internet scene and gained instant celebrity status. The promise of Java technology was that it would become the *universal glue* that connects users with information wherever it comes from—web servers, databases, information providers, or any other imaginable source. Indeed, Java is in a unique position to fulfill this promise. It is an extremely solidly engineered language that has gained acceptance by all major vendors, except for Microsoft. Its built-in security and safety features are reassuring both to programmers and to the users of Java programs. Java even has built-in support for advanced programming tasks, such as network programming, database connectivity, and multithreading.

Since 1995, eight major revisions of the Java Development Kit have been released. Over the course of the last 17 years, the Application Programming Interface (API) has grown from about 200 to over 3,000 classes. The API now spans such diverse areas as user interface construction, database management, internationalization, security, and XML processing.

The book you have in your hands is the first volume of the ninth edition of *Core Java*™. Each edition closely followed a release of the Java Development Kit, and each time, we rewrote the book to take advantage of the newest Java features. This edition has been updated to reflect the features of Java Standard Edition (SE) 7.

As with the previous editions of this book, we *still target serious programmers who want to put Java to work on real projects*. We think of you, our reader, as a programmer with a solid background in a programming language other than Java, and we assume that you don't like books filled with toy examples (such as toasters, zoo animals, or "nervous text"). You won't find any of these in our book. Our goal is to enable you to fully understand the Java language and library, not to give you an illusion of understanding.

In this book you will find lots of sample code demonstrating almost every language and library feature that we discuss. We keep the sample programs purposefully simple to focus on the major points, but, for the most part, they aren't fake and they don't cut corners. They should make good starting points for your own code.

We assume you are willing, even eager, to learn about all the advanced features that Java puts at your disposal. For example, we give you a detailed treatment of

- Object-oriented programming
- Reflection and proxies
- Interfaces and inner classes
- The event listener model
- Graphical user interface design with the Swing UI toolkit
- Exception handling
- Generic programming
- The collections framework
- Concurrency

With the explosive growth of the Java class library, a one-volume treatment of all the features of Java that serious programmers need to know is no longer possible. Hence, we decided to break up the book into two volumes. The first volume, which you hold in your hands, concentrates on the fundamental concepts of the Java language, along with the basics of user-interface programming. The second volume, *Core Java™, Volume II—Advanced Features* (forthcoming, ISBN: 978-0-13-708160-8), goes further into the enterprise features and advanced user-interface programming. It includes detailed discussions of

- Files and streams
- Distributed objects
- Databases
- Advanced GUI components
- Native methods
- XML processing
- Network programming
- Advanced graphics
- Internationalization
- JavaBeans
- Annotations

When writing a book, errors and inaccuracies are inevitable. We'd very much like to know about them. But, of course, we'd prefer to learn about each of them only once. We have put up a list of frequently asked questions, bugs fixes, and workarounds on a web page at http://horstmann.com/corejava. Strategically placed at the end of the errata page (to encourage you to read through it first) is a form you can use to report bugs and suggest improvements. Please don't be disappointed if we don't answer every query or don't get back to you immediately. We do read all e-mail and appreciate your input to make future editions of this book clearer and more informative.

A Tour of This Book

Chapter 1 gives an overview of the capabilities of Java that set it apart from other programming languages. We explain what the designers of the language set out to do and to what extent they succeeded. Then, we give a short history of how Java came into being and how it has evolved.

In **Chapter 2**, we tell you how to download and install the JDK and the program examples for this book. Then we guide you through compiling and running three typical Java programs—a console application, a graphical application, and an applet—using the plain JDK, a Java-enabled text editor, and a Java IDE.

Chapter 3 starts the discussion of the Java language. In this chapter, we cover the basics: variables, loops, and simple functions. If you are a C or C++ programmer, this is smooth sailing because the syntax for these language features is essentially the same as in C. If you come from a non-C background such as Visual Basic, you will want to read this chapter carefully.

Object-oriented programming (OOP) is now in the mainstream of programming practice, and Java is an object-oriented programming language. **Chapter 4** introduces encapsulation, the first of two fundamental building blocks of object orientation, and the Java language mechanism to implement it—that is, classes and methods. In addition to the rules of the Java language, we also give advice on sound OOP design. Finally, we cover the marvelous `javadoc` tool that formats your code comments as a set of hyperlinked web pages. If you are familiar with C++, you can browse through this chapter quickly. Programmers coming from a non-object-oriented background should expect to spend some time mastering the OOP concepts before going further with Java.

Classes and encapsulation are only one part of the OOP story, and **Chapter 5** introduces the other—namely, *inheritance*. Inheritance lets you take an existing class and modify it according to your needs. This is a fundamental technique for programming in Java. The inheritance mechanism in Java is quite similar to that in C++. Once again, C++ programmers can focus on the differences between the languages.

Chapter 6 shows you how to use Java's notion of an *interface*. Interfaces let you go beyond the simple inheritance model of Chapter 5. Mastering interfaces allows you to have full access to the power of Java's completely object-oriented approach to programming. We also cover a useful technical feature of Java called *inner classes*. Inner classes help make your code cleaner and more concise.

In **Chapter 7**, we begin application programming in earnest. Every Java programmer should know a bit about GUI programming, and this volume contains the basics. We show how you can make windows, how to paint on them, how to draw with geometric shapes, how to format text in multiple fonts, and how to display images.

Chapter 8 is a detailed discussion of the event model of the AWT, the *abstract window toolkit*. You'll see how to write code that responds to events, such as mouse clicks or key presses. Along the way you'll see how to handle basic GUI elements like buttons and panels.

Chapter 9 discusses the Swing GUI toolkit in great detail. The Swing toolkit allows you to build cross-platform graphical user interfaces. You'll learn all about the various kinds of buttons, text components, borders, sliders, list boxes, menus, and dialog boxes. However, some of the more advanced components are discussed in Volume II.

Chapter 10 shows you how to deploy your programs, either as applications or applets. We describe how to package programs in JAR files, and how to deliver applications over the Internet with the Java Web Start and applet mechanisms. Finally, we explain how Java programs can store and retrieve configuration information once they have been deployed.

Chapter 11 discusses *exception handling*—Java's robust mechanism to deal with the fact that bad things can happen to good programs. Exceptions give you an efficient way of separating the normal processing code from the error handling. Of course, even after hardening your program by handling all exceptional conditions, it still might fail to work as expected. In the second half of this chapter, we give you a large number of useful debugging tips. Finally, we guide you through a sample debugging session.

Chapter 12 gives an overview of generic programming—a major advance of Java SE 5.0. Generic programming makes your programs easier to read and safer. We show you how to use strong typing and remove unsightly and unsafe casts, and how to deal with the complexities that arise from the need to stay compatible with older versions of Java.

The topic of **Chapter 13** is the collections framework of the Java platform. Whenever you want to collect multiple objects and retrieve them later, you should use a collection that is best suited for your circumstances, instead of just tossing the elements into an array. This chapter shows you how to take advantage of the standard collections that are prebuilt for your use.

Chapter 14 finishes the book with a discussion of multithreading, which enables you to program tasks to be done in parallel. (A thread is a flow of control within a program.) We show you how to set up threads and how to deal with thread synchronization. Multithreading has changed a great deal in Java SE 5.0, and we tell you all about the new mechanisms.

The **Appendix** lists the reserved words of the Java language.

Conventions

As is common in many computer books, we use `monospace type` to represent computer code.

NOTE: Notes are tagged with “note” icons that look like this.

TIP: Tips are tagged with “tip” icons that look like this.

CAUTION: When there is danger ahead, we warn you with a “caution” icon.

C++ NOTE: There are many C++ notes that explain the differences between Java and C++. You can skip over them if you don’t have a background in C++ or if you consider your experience with that language a bad dream of which you’d rather not be reminded.

Java comes with a large programming library, or Application Programming Interface (API). When using an API call for the first time, we add a short summary description at the end of the section. These descriptions are a bit more informal but, we hope, also a little more informative than those in the official online API documentation. The names of interfaces are in italics, just like in the official documentation. The number after a class, interface, or method name is the JDK version in which the feature was introduced.

`Application Programming Interface` 1.2

Programs whose source code is on the book's companion web site are listed as listings, for instance

Listing 1.1 `InputTest/InputTest.java`

Sample Code

The web site for this book at http://horstmann.com/corejava contains all sample code from the book, in compressed form. You can expand the file either with one of the familiar unzipping programs or simply with the jar utility that is part of the Java Development Kit. See Chapter 2 for more information about installing the Java Development Kit and the sample code.

Acknowledgments

Writing a book is always a monumental effort, and rewriting it doesn't seem to be much easier, especially with the continuous change in Java technology. Making a book a reality takes many dedicated people, and it is my great pleasure to acknowledge the contributions of the entire Core Java team.

A large number of individuals at Prentice Hall provided valuable assistance but managed to stay behind the scenes. I'd like them all to know how much I appreciate their efforts. As always, my warm thanks go to my editor, Greg Doench, for steering the book through the writing and production process, and for allowing me to be blissfully unaware of the existence of all those folks behind the scenes. I am very grateful to Julie Nahil for production support, and to Dmitry Kirsanov and Alina Kirsanova for copyediting and typesetting the manuscript. My thanks also to my coauthor of earlier editions, Gary Cornell, who has since moved on to other ventures.

Thanks to the many readers of earlier editions who reported embarrassing errors and made lots of thoughtful suggestions for improvement. I am particularly grateful to the excellent reviewing team that went over the manuscript with an amazing eye for detail and saved me from many more embarrassing errors.

Reviewers of this and earlier editions include Chuck Allison (Utah Valley University), Lance Andersen (Oracle), Alec Beaton (IBM), Cliff Berg, Joshua Bloch, David Brown, Corky Cartwright, Frank Cohen (PushToTest), Chris Crane (devXsolution), Dr. Nicholas J. De Lillo (Manhattan College), Rakesh Dhoopar (Oracle), David Geary (Clarity Training), Jim Gish (Oracle), Brian Goetz (Oracle), Angela Gordon, Dan Gordon (Electric Cloud), Rob Gordon, John Gray (University of Hartford), Cameron Gregory (olabs.com), Marty Hall (coreservlets.com, Inc.), Vincent Hardy (Adobe Systems), Dan Harkey (San Jose State University), William Higgins (IBM), Vladimir Ivanovic (PointBase), Jerry Jackson (CA Technologies), Tim Kimmet (Walmart), Chris Laffra, Charlie Lai (Apple), Angelika Langer, Doug Langston, Hang Lau (McGill University), Mark Lawrence, Doug Lea (SUNY Oswego), Gregory Longshore, Bob Lynch (Lynch Associates), Philip Milne (consultant), Mark Morrissey (The Oregon Graduate Institute), Mahesh Neelakanta (Florida Atlantic University), Hao Pham, Paul Philion, Blake Ragsdell, Stuart Reges (University of Arizona), Rich Rosen (Interactive Data Corporation), Peter Sanders (ESSI University, Nice, France), Dr. Paul Sanghera (San Jose State University and

Brooks College), Paul Sevinc (Teamup AG), Devang Shah (Sun Microsystems), Bradley A. Smith, Steven Stelting (Oracle), Christopher Taylor, Luke Taylor (Valtech), George Thiruvathukal, Kim Topley (StreamingEdge), Janet Traub, Paul Tyma (consultant), Peter van der Linden (Motorola Mobile Devices), Burt Walsh, Dan Xu (Oracle), and John Zavgren (Oracle).

Cay Horstmann
San Francisco, California
September 2012

CHAPTER 6

Interfaces and Inner Classes

In this chapter:

You have now seen all the basic tools for object-oriented programming in Java. This chapter shows you several advanced techniques that are commonly used. Despite their less obvious nature, you will need to master them to complete your Java tool chest.

The first technique, called *interfaces*, is a way of describing *what* classes should do, without specifying *how* they should do it. A class can *implement* one or more interfaces. You can then use objects of these implementing classes whenever conformance to the interface is required. After we cover interfaces, we take up cloning an object (or deep copying, as it is sometimes called). A clone of an object is a new object that has the same state as the original. In particular, you can modify the clone without affecting the original.

Next, we move on to the mechanism of *inner classes*. Inner classes are technically somewhat complex—they are defined inside other classes, and their methods can access the fields of the surrounding class. Inner classes are useful when you design collections of cooperating classes. In particular, inner classes enable you to write concise, professional looking code to handle GUI events.

This chapter concludes with a discussion of *proxies*, objects that implement arbitrary interfaces. A proxy is a very specialized construct that is useful for building system-level tools. You can safely skip that section on first reading.

6.1 Interfaces

In the Java programming language, an interface is not a class but a set of *requirements* for the classes that want to conform to the interface.

Typically, the supplier of some service states: "If your class conforms to a particular interface, then I'll perform the service." Let's look at a concrete example. The `sort` method of the `Arrays` class promises to sort an array of objects, but under one condition: The objects must belong to classes that implement the `Comparable` interface.

Here is what the `Comparable` interface looks like:

```
public interface Comparable
{
   int compareTo(Object other);
}
```

This means that any class that implements the `Comparable` interface is required to have a `compareTo` method, and the method must take an `Object` parameter and return an integer.

NOTE: As of Java SE 5.0, the `Comparable` interface has been enhanced to be a generic type.

```
public interface Comparable<T>
{
   int compareTo(T other); // parameter has type T
}
```

For example, a class that implements `Comparable<Employee>` must supply a method

```
int compareTo(Employee other)
```

You can still use the "raw" `Comparable` type without a type parameter, but then you have to manually cast the parameter of the `compareTo` method to the desired type.

All methods of an interface are automatically `public`. For that reason, it is not necessary to supply the keyword `public` when declaring a method in an interface.

Of course, there is an additional requirement that the interface cannot spell out: When calling `x.compareTo(y)`, the `compareTo` method must actually be able to *compare* the two objects and return an indication whether `x` or `y` is larger. The method is supposed to return a negative number if `x` is smaller than `y`, zero if they are equal, and a positive number otherwise.

This particular interface has a single method. Some interfaces have multiple methods. As you will see later, interfaces can also define constants. What is more important, however, is what interfaces *cannot* supply. Interfaces never have instance fields, and the methods are never implemented in the interface. Supplying instance fields and method implementations is the job of the classes that implement the interface. You can think of an interface as being similar to an abstract class with no instance fields. However, there are some differences between these two concepts—we look at them later in some detail.

Now suppose we want to use the `sort` method of the `Arrays` class to sort an array of `Employee` objects. Then the `Employee` class must *implement* the `Comparable` interface.

To make a class implement an interface, you carry out two steps:

1. You declare that your class intends to implement the given interface.
2. You supply definitions for all methods in the interface.

To declare that a class implements an interface, use the `implements` keyword:

```
class Employee implements Comparable
```

Of course, now the `Employee` class needs to supply the `compareTo` method. Let's suppose that we want to compare employees by their salary. Here is an implementation of the `compareTo` method:

```
public int compareTo(Object otherObject)
{
   Employee other = (Employee) otherObject;
   return Double.compare(salary, other.salary);
}
```

Here, we use the static `Double.compare` method that returns a negative if the first argument is less than the second argument, `0` if they are equal, and a positive value otherwise.

CAUTION: In the interface declaration, the `compareTo` method was not declared `public` because all methods in an *interface* are automatically public. However, when implementing the interface, you must declare the method as `public`. Otherwise, the compiler assumes that the method has package visibility—the default for a *class*. The compiler then complains that you're trying to supply a weaker access privilege.

As of Java SE 5.0, we can do a little better. We'll implement the `Comparable<Employee>` interface type instead.

```
class Employee implements Comparable<Employee>
{
   public int compareTo(Employee other)
   {
      return Double.compare(salary, other.salary);
   }
   . . .
}
```

Note that the unsightly cast of the `Object` parameter has gone away.

TIP: The `compareTo` method of the `Comparable` interface returns an integer. If the objects are not equal, it does not matter what negative or positive value you return. This flexibility can be useful when you are comparing integer fields. For example, suppose each employee has a unique integer `id` and you want to sort by the employee ID number. Then you can simply return `id - other.id`. That value will be some negative value if the first ID number is less than the other, `0` if they are the same ID, and some positive value otherwise. However, there is one caveat: The range of the integers must be small enough so that the subtraction does not overflow. If you know that the IDs are not negative or that their absolute value is at most `(Integer.MAX_VALUE - 1) / 2`, you are safe.

Of course, the subtraction trick doesn't work for floating-point numbers. The difference `salary - other.salary` can round to `0` if the salaries are close together but not identical. The call `Double.compare(x, y)` simply returns `-1` if `x < y` or `1` if `x > y`.

Now you saw what a class must do to avail itself of the sorting service—it must implement a `compareTo` method. That's eminently reasonable. There needs to be some way for the `sort` method to compare objects. But why can't the `Employee` class simply provide a `compareTo` method without implementing the `Comparable` interface?

The reason for interfaces is that the Java programming language is *strongly typed*. When making a method call, the compiler needs to be able to check that the method actually exists. Somewhere in the `sort` method will be statements like this:

```
if (a[i].compareTo(a[j]) > 0)
{
   // rearrange a[i] and a[j]
   . . .
}
```

The compiler must know that `a[i]` actually has a `compareTo` method. If `a` is an array of `Comparable` objects, then the existence of the method is assured because every class that implements the `Comparable` interface must supply the method.

NOTE: You would expect that the `sort` method in the `Arrays` class is defined to accept a `Comparable[]` array so that the compiler can complain if anyone ever calls `sort` with an array whose element type doesn't implement the `Comparable` interface. Sadly, that is not the case. Instead, the `sort` method accepts an `Object[]` array and uses a clumsy cast:

```
// Approach used in the standard library--not recommended
if (((Comparable) a[i]).compareTo(a[j]) > 0)
{
   // rearrange a[i] and a[j]
   . . .
}
```

If `a[i]` does not belong to a class that implements the `Comparable` interface, the virtual machine throws an exception.

Listing 6.1 presents the full code for sorting an array of instances of the class `Employee` (Listing 6.2). for sorting an employee array.

Listing 6.1 `interfaces/EmployeeSortTest.java`

```
package interfaces;

import java.util.*;

/**
 * This program demonstrates the use of the Comparable interface.
 * @version 1.30 2004-02-27
 * @author Cay Horstmann
 */
public class EmployeeSortTest
{
   public static void main(String[] args)
   {
      Employee[] staff = new Employee[3];

      staff[0] = new Employee("Harry Hacker", 35000);
      staff[1] = new Employee("Carl Cracker", 75000);
      staff[2] = new Employee("Tony Tester", 38000);

      Arrays.sort(staff);

      // print out information about all Employee objects
      for (Employee e : staff)
         System.out.println("name=" + e.getName() + ",salary=" + e.getSalary());
   }
}
```

Listing 6.2 `interfaces/Employee.java`

```java
package interfaces;

public class Employee implements Comparable<Employee>
{
   private String name;
   private double salary;

   public Employee(String n, double s)
   {
      name = n;
      salary = s;
   }

   public String getName()
   {
      return name;
   }

   public double getSalary()
   {
      return salary;
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   /**
    * Compares employees by salary
    * @param other another Employee object
    * @return a negative value if this employee has a lower salary than
    * otherObject, 0 if the salaries are the same, a positive value otherwise
    */
   public int compareTo(Employee other)
   {
      return Double.compare(salary, other.salary);
   }
}
```

java.lang.Comparable<T> 1.0

- `int compareTo(T other)`
  compares this object with `other` and returns a negative integer if this object is less than `other`, zero if they are equal, and a positive integer otherwise.

java.util.Arrays 1.2

- `static void sort(Object[] a)`
  sorts the elements in the array `a`, using a tuned `mergesort` algorithm. All elements in the array must belong to classes that implement the `Comparable` interface, and they must all be comparable to each other.

java.lang.Integer 7

- `static int compare(int x, int y)`
  returns a negative integer if `x < y`, zero if `x` and `y` are equal, and a positive integer otherwise.

java.lang.Double 7

- `static int compare(double x, double y)`
  returns a negative integer if `x < y`, zero if `x` and `y` are equal, and a positive integer otherwise.

NOTE: According to the language standard: "The implementor must ensure `sgn(x.compareTo(y)) = -sgn(y.compareTo(x))` for all `x` and `y`. (This implies that `x.compareTo(y)` must throw an exception if `y.compareTo(x)` throws an exception.)" Here, `sgn` is the *sign* of a number: `sgn(`n`)` is –1 if n is negative, `0` if n equals `0`, and `1` if n is positive. In plain English, if you flip the parameters of `compareTo`, the sign (but not necessarily the actual value) of the result must also flip.

As with the `equals` method, problems can arise when inheritance comes into play.

Since `Manager` extends `Employee`, it implements `Comparable<Employee>` and not `Comparable<Manager>`. If `Manager` chooses to override `compareTo`, it must be prepared to compare managers to employees. It can't simply cast an employee to a manager:

```
class Manager extends Employee
{
   public int compareTo(Employee other)
   {
      Manager otherManager = (Manager) other; // NO
      . . .
   }
   . . .
}
```

That violates the "antisymmetry" rule. If `x` is an `Employee` and `y` is a `Manager`, then the call `x.compareTo(y)` doesn't throw an exception—it simply compares `x` and `y` as employees. But the reverse, `y.compareTo(x)`, throws a `ClassCastException`.

This is the same situation as with the `equals` method that we discussed in Chapter 5, and the remedy is the same. There are two distinct scenarios.

If subclasses have different notions of comparison, then you should outlaw comparison of objects that belong to different classes. Each `compareTo` method should start out with the test

```
if (getClass() != other.getClass()) throw new ClassCastException();
```

If there is a common algorithm for comparing subclass objects, simply provide a single `compareTo` method in the superclass and declare it as `final`.

For example, suppose that you want managers to be better than regular employees, regardless of the salary. What about other subclasses such as `Executive` and `Secretary`? If you need to establish a pecking order, supply a method such as `rank` in the `Employee` class. Have each subclass override `rank`, and implement a single `compareTo` method that takes the `rank` values into account.

6.1.1 Properties of Interfaces

Interfaces are not classes. In particular, you can never use the `new` operator to instantiate an interface:

```
x = new Comparable(. . .); // ERROR
```

However, even though you can't construct interface objects, you can still declare interface variables.

```
Comparable x; // OK
```

An interface variable must refer to an object of a class that implements the interface:

```
x = new Employee(. . .); // OK provided Employee implements Comparable
```

Next, just as you use `instanceof` to check whether an object is of a specific class, you can use `instanceof` to check whether an object implements an interface:

```
if (anObject instanceof Comparable) { . . . }
```

Just as you can build hierarchies of classes, you can extend interfaces. This allows for multiple chains of interfaces that go from a greater degree of generality to a greater degree of specialization. For example, suppose you had an interface called `Moveable`.

```
public interface Moveable
{
   void move(double x, double y);
}
```

Then, you could imagine an interface called `Powered` that extends it:

```
public interface Powered extends Moveable
{
   double milesPerGallon();
}
```

Although you cannot put instance fields or static methods in an interface, you can supply constants in them. For example:

```
public interface Powered extends Moveable
{
   double milesPerGallon();
   double SPEED_LIMIT = 95; // a public static final constant
}
```

Just as methods in an interface are automatically `public`, fields are always `public static final`.

NOTE: It is legal to tag interface methods as `public`, and fields as `public static final`. Some programmers do that, either out of habit or for greater clarity. However, the Java Language Specification recommends that the redundant keywords not be supplied, and we follow that recommendation.

Some interfaces define just constants and no methods. For example, the standard library contains an interface `SwingConstants` that defines constants `NORTH`, `SOUTH`, `HORIZONTAL`, and so on. Any class that chooses to implement the `SwingConstants` interface automatically inherits these constants. Its methods can simply refer to `NORTH` rather than the more cumbersome `SwingConstants.NORTH`. However, this use of interfaces seems rather degenerate, and we do not recommend it.

While each class can have only one superclass, classes can implement *multiple* interfaces. This gives you the maximum amount of flexibility in defining a class's behavior. For example, the Java programming language has an important interface built into it, called `Cloneable`. (We will discuss this interface in detail in the next section.) If your class implements `Cloneable`, the `clone` method in the `Object` class will make an exact copy of your class's objects. Suppose, therefore, you want cloneability and comparability. Then you simply implement both interfaces.

```
class Employee implements Cloneable, Comparable
```

Use commas to separate the interfaces that describe the characteristics that you want to supply.

6.1.2 Interfaces and Abstract Classes

If you read the section about abstract classes in Chapter 5, you may wonder why the designers of the Java programming language bothered with introducing the concept of interfaces. Why can't `Comparable` simply be an abstract class:

```
abstract class Comparable // why not?
{
   public abstract int compareTo(Object other);
}
```

The `Employee` class would then simply extend this abstract class and supply the `compareTo` method:

```
class Employee extends Comparable // why not?
{
   public int compareTo(Object other) { . . . }
}
```

There is, unfortunately, a major problem with using an abstract base class to express a generic property. A class can only extend a single class. Suppose that the `Employee` class already extends a different class, say, `Person`. Then it can't extend a second class.

```
class Employee extends Person, Comparable // ERROR
```

But each class can implement as many interfaces as it likes:

```
class Employee extends Person implements Comparable // OK
```

Other programming languages, in particular C++, allow a class to have more than one superclass. This feature is called *multiple inheritance*. The designers of Java chose not to support multiple inheritance, because it makes the language either very complex (as in C++) or less efficient (as in Eiffel).

Instead, interfaces afford most of the benefits of multiple inheritance while avoiding the complexities and inefficiencies.

C++ **C++ NOTE:** C++ has multiple inheritance and all the complications that come with it, such as virtual base classes, dominance rules, and transverse pointer casts. Few C++ programmers use multiple inheritance, and some say it should never be used. Other programmers recommend using multiple inheritance only for the “mix-in” style of inheritance. In the mix-in style, a primary base class describes the parent object, and additional base classes (the so-called mix-ins) may supply auxiliary characteristics. That style is similar to a Java class with a single base class and additional interfaces. However, in C++, mix-ins can add default behavior, whereas Java interfaces cannot.

6.2 Object Cloning

When you make a copy of a variable, the original and the copy are references to the same object. (See Figure 6.1.) This means a change to either variable also affects the other.

```
Employee original = new Employee("John Public", 50000);
Employee copy = original;
copy.raiseSalary(10); // oops--also changed original
```

If you would like `copy` to be a new object that begins its life being identical to `original` but whose state can diverge over time, use the `clone` method.

```
Employee copy = original.clone();
copy.raiseSalary(10); // OK--original unchanged
```

But it isn’t quite so simple. The `clone` method is a `protected` method of `Object`, which means that your code cannot simply call it. Only the `Employee` class can clone `Employee` objects. There is a reason for this restriction. Think about the way in which the `Object` class can implement `clone`. It knows nothing about the object at all, so it can make only a field-by-field copy. If all data fields in the object are numbers or other basic types, copying the fields is just fine. But if the object contains references to subobjects, then copying the field gives you another reference to the same subobject, so the original and the cloned objects still share some information.

To visualize that phenomenon, let’s consider the `Employee` class that was introduced in Chapter 4. Figure 6.2 shows what happens when you use the `clone` method of the `Object` class to clone such an `Employee` object. As you can see, the default cloning operation is “shallow”—it doesn’t clone objects that are referenced inside other objects.

Does it matter if the copy is shallow? It depends. If the subobject shared between the original and the shallow clone is *immutable*, then the sharing is safe. This certainly happens if the subobject belongs to an immutable class, such as `String`.

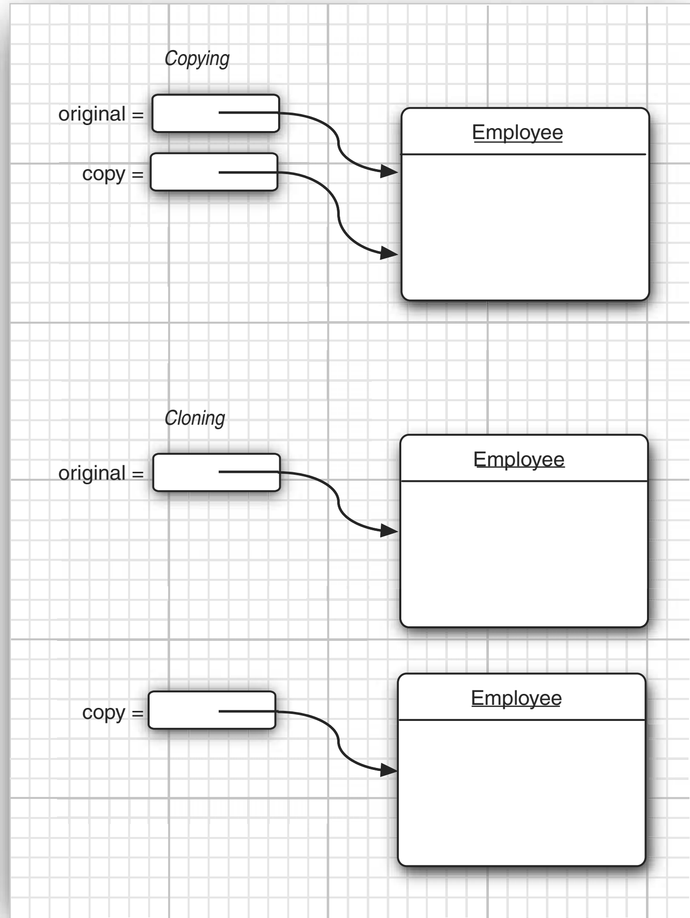

Figure 6.1 Copying and cloning

Alternatively, the subobject may simply remain constant throughout the lifetime of the object, with no mutators touching it and no methods yielding a reference to it.

Quite frequently, however, subobjects are mutable, and you must redefine the `clone` method to make a *deep copy* that clones the subobjects as well. In our example, the `hireDay` field is a `Date`, which is mutable.

For every class, you need to decide whether

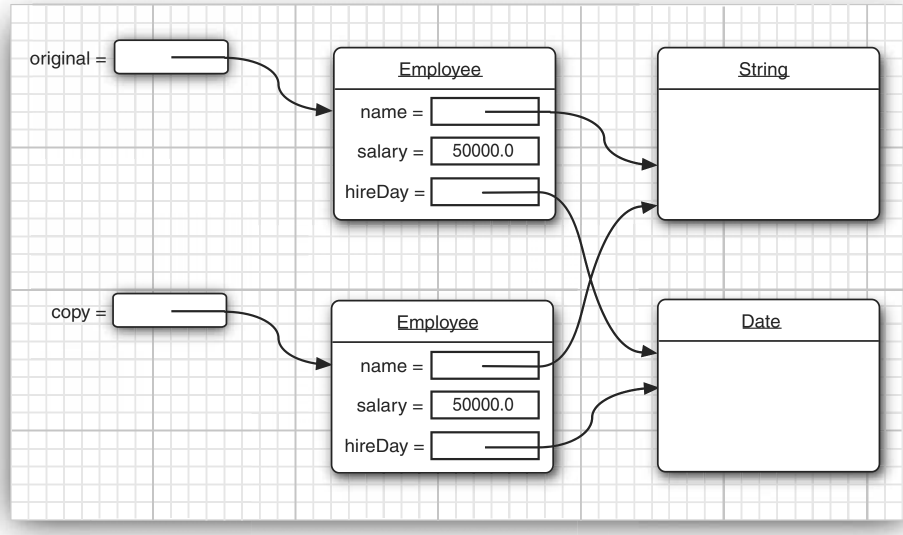

Figure 6.2 A shallow copy

1. The default `clone` method is good enough;
2. The default `clone` method can be patched up by calling `clone` on the mutable subobjects; and
3. `clone` should not be attempted.

The third option is actually the default. To choose either the first or the second option, a class must

1. Implement the `Cloneable` interface; and
2. Redefine the `clone` method with the `public` access modifier.

> **NOTE:** The `clone` method is declared `protected` in the `Object` class, so that your code can't simply call `anObject.clone()`. But aren't protected methods accessible from any subclass, and isn't every class a subclass of `Object`? Fortunately, the rules for protected access are more subtle (see Chapter 5). A subclass can call a protected `clone` method only to clone *its own* objects. You must redefine `clone` to be public to allow objects to be cloned by any method.

In this case, the appearance of the `Cloneable` interface has nothing to do with the normal use of interfaces. In particular, it does *not* specify the `clone` method—that method is inherited from the `Object` class. The interface merely serves as a tag, indicating that the class designer understands the cloning process. Objects are so paranoid about cloning that they generate a checked exception if an object requests cloning but does not implement that interface.

NOTE: The `Cloneable` interface is one of a handful of *tagging interfaces* that Java provides. (Some programmers call them *marker interfaces*.) Recall that the usual purpose of an interface such as `Comparable` is to ensure that a class implements a particular method or set of methods. A tagging interface has no methods; its only purpose is to allow the use of `instanceof` in a type inquiry:

```
if (obj instanceof Cloneable) . . .
```

We recommend that you do not use tagging interfaces in your own programs.

Even if the default (shallow copy) implementation of `clone` is adequate, you still need to implement the `Cloneable` interface, redefine `clone` to be public, and call `super.clone()`. Here is an example:

```
class Employee implements Cloneable
{
   // raise visibility level to public, change return type
   public Employee clone() throws CloneNotSupportedException
   {
      return (Employee) super.clone();
   }
   . . .
}
```

NOTE: Before Java SE 5.0, the `clone` method always had return type `Object`. Nowadays, the covariant return type feature lets you specify the correct return type for your `clone` methods.

The `clone` method that you just saw adds no functionality to the shallow copy provided by `Object.clone`. It merely makes the method public. To make a deep copy, you have to work harder and clone the mutable instance fields.

Here is an example of a `clone` method that creates a deep copy:

```
class Employee implements Cloneable
{
   . . .
   public Employee clone() throws CloneNotSupportedException
   {
      // call Object.clone()
      Employee cloned = (Employee) super.clone();
```

```
         // clone mutable fields
         cloned.hireDay = (Date) hireDay.clone();

         return cloned;
      }
   }
```

The clone method of the Object class threatens to throw a CloneNotSupportedException—it does that whenever clone is invoked on an object whose class does not implement the Cloneable interface. Of course, the Employee and Date classes implement the Cloneable interface, so the exception won't be thrown. However, the compiler does not know that. Therefore, we declared the exception:

```
public Employee clone() throws CloneNotSupportedException
```

Would it be better to catch the exception instead?

```
public Employee clone()
{
   try
   {
      return (Employee) super.clone();
   }
   catch (CloneNotSupportedException e) { return null; }
   // this won't happen, since we are Cloneable
}
```

This is appropriate for final classes. Otherwise, it is a good idea to leave the throws specifier in place. That gives subclasses the option of throwing a CloneNotSupportedException if they can't support cloning.

You have to be careful about cloning of subclasses. For example, once you have defined the clone method for the Employee class, anyone can use it to clone Manager objects. Can the Employee clone method do the job? It depends on the fields of the Manager class. In our case, there is no problem because the bonus field has primitive type. But Manager might have acquired fields that require a deep copy or are not cloneable. There is no guarantee that the implementor of the subclass has fixed clone to do the right thing. For that reason, the clone method is declared as protected in the Object class. But you don't have that luxury if you want users of your classes to invoke clone.

Should you implement clone in your own classes? If your clients need to make deep copies, then you probably should. Some authors feel that you should avoid clone altogether and instead implement another method for the same purpose. We agree that clone is rather awkward, but you'll run into the same issues if you shift the responsibility to another method. At any rate, cloning is less common than you may think. Less than 5 percent of the classes in the standard library implement `clone`.

The program in Listing 6.3 clones an instance of the class `Employee` (Listing 6.4), then invokes two mutators. The `raiseSalary` method changes the value of the `salary` field, whereas the `setHireDay` method changes the state of the `hireDay` field. Neither mutation affects the original object because `clone` has been defined to make a deep copy.

Listing 6.3 clone/CloneTest.java

```
package clone;

/**
 * This program demonstrates cloning.
 * @version 1.10 2002-07-01
 * @author Cay Horstmann
 */
public class CloneTest
{
   public static void main(String[] args)
   {
      try
      {
         Employee original = new Employee("John Q. Public", 50000);
         original.setHireDay(2000, 1, 1);
         Employee copy = original.clone();
         copy.raiseSalary(10);
         copy.setHireDay(2002, 12, 31);
         System.out.println("original=" + original);
         System.out.println("copy=" + copy);
      }
      catch (CloneNotSupportedException e)
      {
         e.printStackTrace();
      }
   }
}
```

Listing 6.4 clone/Employee.java

```
package clone;

import java.util.Date;
import java.util.GregorianCalendar;
```

```java
public class Employee implements Cloneable
{
   private String name;
   private double salary;
   private Date hireDay;

   public Employee(String n, double s)
   {
      name = n;
      salary = s;
      hireDay = new Date();
   }

   public Employee clone() throws CloneNotSupportedException
   {
      // call Object.clone()
      Employee cloned = (Employee) super.clone();

      // clone mutable fields
      cloned.hireDay = (Date) hireDay.clone();

      return cloned;
   }

   /**
    * Set the hire day to a given date.
    * @param year the year of the hire day
    * @param month the month of the hire day
    * @param day the day of the hire day
    */
   public void setHireDay(int year, int month, int day)
   {
      Date newHireDay = new GregorianCalendar(year, month - 1, day).getTime();

      // Example of instance field mutation
      hireDay.setTime(newHireDay.getTime());
   }

   public void raiseSalary(double byPercent)
   {
      double raise = salary * byPercent / 100;
      salary += raise;
   }

   public String toString()
   {
      return "Employee[name=" + name + ",salary=" + salary + ",hireDay=" + hireDay + "]";
   }
}
```

NOTE: All array types have a `clone` method that is public, not protected. You can use it to make a new array that contains copies of all elements. For example:

```
int[] luckyNumbers = { 2, 3, 5, 7, 11, 13 };
int[] cloned = luckyNumbers.clone();
cloned[5] = 12; // doesn't change luckyNumbers[5]
```

NOTE: Chapter 1 of Volume II shows an alternate mechanism for cloning objects, using the object serialization feature of Java. That mechanism is easy to implement and safe, but not very efficient.

6.3 Interfaces and Callbacks

A common pattern in programming is the *callback* pattern. In this pattern, you want to specify the action that should occur whenever a particular event happens. For example, you may want a particular action to occur when a button is clicked or a menu item is selected. However, as you have not yet seen how to implement user interfaces, we will consider a similar but simpler situation.

The `javax.swing` package contains a `Timer` class that is useful if you want to be notified whenever a time interval has elapsed. For example, if a part of your program contains a clock, then you can ask to be notified every second so that you can update the clock face.

When you construct a timer, you set the time interval and you tell it what it should do whenever the time interval has elapsed.

How do you tell the timer what it should do? In many programming languages, you supply the name of a function that the timer should call periodically. However, the classes in the Java standard library take an object-oriented approach. You pass an object of some class. The timer then calls one of the methods on that object. Passing an object is more flexible than passing a function because the object can carry additional information.

Of course, the timer needs to know what method to call. The timer requires that you specify an object of a class that implements the `ActionListener` interface of the `java.awt.event` package. Here is that interface:

```
public interface ActionListener
{
   void actionPerformed(ActionEvent event);
}
```

The timer calls the `actionPerformed` method when the time interval has expired.

C++ NOTE: As you saw in Chapter 5, Java does have the equivalent of function pointers, namely, `Method` objects. However, they are difficult to use, slower, and cannot be checked for type safety at compile time. Whenever you would use a function pointer in C++, you should consider using an interface in Java.

Suppose you want to print a message "At the tone, the time is . . .", followed by a beep, once every 10 seconds. You would define a class that implements the `ActionListener` interface. You would then place whatever statements you want to have executed inside the `actionPerformed` method.

```
class TimePrinter implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      Date now = new Date();
      System.out.println("At the tone, the time is " + now);
      Toolkit.getDefaultToolkit().beep();
   }
}
```

Note the `ActionEvent` parameter of the `actionPerformed` method. This parameter gives information about the event, such as the source object that generated it—see Chapter 8 for more information. However, detailed information about the event is not important in this program, and you can safely ignore the parameter.

Next, you construct an object of this class and pass it to the `Timer` constructor.

```
ActionListener listener = new TimePrinter();
Timer t = new Timer(10000, listener);
```

The first parameter of the `Timer` constructor is the time interval that must elapse between notifications, measured in milliseconds. We want to be notified every 10 seconds. The second parameter is the listener object.

Finally, you start the timer.

```
t.start();
```

Every 10 seconds, a message like

```
At the tone, the time is Thu Apr 13 23:29:08 PDT 2000
```

is displayed, followed by a beep.

Listing 6.5 puts the timer and its action listener to work. After the timer is started, the program puts up a message dialog and waits for the user to click the Ok button to stop. While the program waits for the user, the current time is displayed at 10-second intervals.

Be patient when running the program. The “Quit program?” dialog box appears right away, but the first timer message is displayed after 10 seconds.

Note that the program imports the `javax.swing.Timer` class by name, in addition to importing `javax.swing.*` and `java.util.*`. This breaks the ambiguity between `javax.swing.Timer` and `java.util.Timer`, an unrelated class for scheduling background tasks.

Listing 6.5 `timer/TimerTest.java`

```
package timer;
/**
   @version 1.00 2000-04-13
   @author Cay Horstmann
*/

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.Timer;
// to resolve conflict with java.util.Timer

public class TimerTest
{
   public static void main(String[] args)
   {
      ActionListener listener = new TimePrinter();

      // construct a timer that calls the listener
      // once every 10 seconds
      Timer t = new Timer(10000, listener);
      t.start();

      JOptionPane.showMessageDialog(null, "Quit program?");
      System.exit(0);
   }
}

class TimePrinter implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      Date now = new Date();
      System.out.println("At the tone, the time is " + now);
      Toolkit.getDefaultToolkit().beep();
   }
}
```

javax.swing.JOptionPane 1.2

- `static void showMessageDialog(Component parent, Object message)`
  displays a dialog box with a message prompt and an Ok button. The dialog is centered over the `parent` component. If `parent` is `null`, the dialog is centered on the screen.

javax.swing.Timer 1.2

- `Timer(int interval, ActionListener listener)`
  constructs a timer that notifies `listener` whenever `interval` milliseconds have elapsed.
- `void start()`
  starts the timer. Once started, the timer calls `actionPerformed` on its listeners.
- `void stop()`
  stops the timer. Once stopped, the timer no longer calls `actionPerformed` on its listeners.

java.awt.Toolkit 1.0

- `static Toolkit getDefaultToolkit()`
  gets the default toolkit. A toolkit contains information about the GUI environment.
- `void beep()`
  emits a beep sound.

6.4 Inner Classes

An *inner class* is a class that is defined inside another class. Why would you want to do that? There are three reasons:

- Inner class methods can access the data from the scope in which they are defined—including the data that would otherwise be private.
- Inner classes can be hidden from other classes in the same package.
- *Anonymous* inner classes are handy when you want to define callbacks without writing a lot of code.

We will break up this rather complex topic into several steps.

1. Starting on page 307, you will see a simple inner class that accesses an instance field of its outer class.
2. On page 311, we cover the special syntax rules for inner classes.

3. Starting on page 312, we peek inside inner classes to see how they are translated into regular classes. Squeamish readers may want to skip that section.
4. Starting on page 315, we discuss *local inner classes* that can access local variables of the enclosing scope.
5. Starting on page 318, we introduce *anonymous inner classes* and show how they are commonly used to implement callbacks.
6. Finally, starting on page 322, you will see how *static inner classes* can be used for nested helper classes.

C++ NOTE: C++ has *nested classes*. A nested class is contained inside the scope of the enclosing class. Here is a typical example: A linked list class defines a class to hold the links, and a class to define an iterator position.

```
class LinkedList
{
public:
   class Iterator // a nested class
   {
   public:
      void insert(int x);
      int erase();
      . . .
   };
   . . .
private:
   class Link // a nested class
   {
   public:
      Link* next;
      int data;
   };
   . . .
};
```

The nesting is a relationship between *classes,* not *objects.* A `LinkedList` object does *not* have subobjects of type `Iterator` or `Link`.

There are two benefits: *name control* and *access control*. The name `Iterator` is nested inside the `LinkedList` class, so it is known externally as `LinkedList::Iterator` and cannot conflict with another class called `Iterator`. In Java, this benefit is not as important because Java *packages* give the same kind of name control. Note that the `Link` class is in the *private* part of the `LinkedList` class. It is completely hidden from all other code. For that reason, it is safe to make its data fields public. They can be accessed by the methods of the `LinkedList` class (which has a legitimate need to access them), and they are not visible elsewhere. In Java, this kind of control was not possible until inner classes were introduced.

However, the Java inner classes have an additional feature that makes them richer and more useful than nested classes in C++. An object that comes from an inner class has an implicit reference to the outer class object that instantiated it. Through this pointer, it gains access to the total state of the outer object. You will see the details of the Java mechanism later in this chapter.

In Java, `static` inner classes do not have this added pointer. They are the Java analog to nested classes in C++.

6.4.1 Use of an Inner Class to Access Object State

The syntax for inner classes is rather complex. For that reason, we present a simple but somewhat artificial example to demonstrate the use of inner classes. We refactor the `TimerTest` example and extract a `TalkingClock` class. A talking clock is constructed with two parameters: the interval between announcements and a flag to turn beeps on or off.

```
public class TalkingClock
{
   private int interval;
   private boolean beep;

   public TalkingClock(int interval, boolean beep) { . . . }
   public void start() { . . . }

   public class TimePrinter implements ActionListener
      // an inner class
   {
      . . .
   }
}
```

Note that the `TimePrinter` class is now located inside the `TalkingClock` class. This does *not* mean that every `TalkingClock` has a `TimePrinter` instance field. As you will see, the `TimePrinter` objects are constructed by methods of the `TalkingClock` class.

Here is the `TimePrinter` class in greater detail. Note that the `actionPerformed` method checks the `beep` flag before emitting a beep.

```
public class TimePrinter implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      Date now = new Date();
      System.out.println("At the tone, the time is " + now);
      if (beep) Toolkit.getDefaultToolkit().beep();
   }
}
```

Something surprising is going on. The `TimePrinter` class has no instance field or variable named `beep`. Instead, `beep` refers to the field of the `TalkingClock` object that created this `TimePrinter`. This is quite innovative. Traditionally, a method could refer to the data fields of the object invoking the method. An inner class method gets to access both its own data fields *and* those of the outer object creating it.

For this to work, an object of an inner class always gets an implicit reference to the object that created it. (See Figure 6.3.)

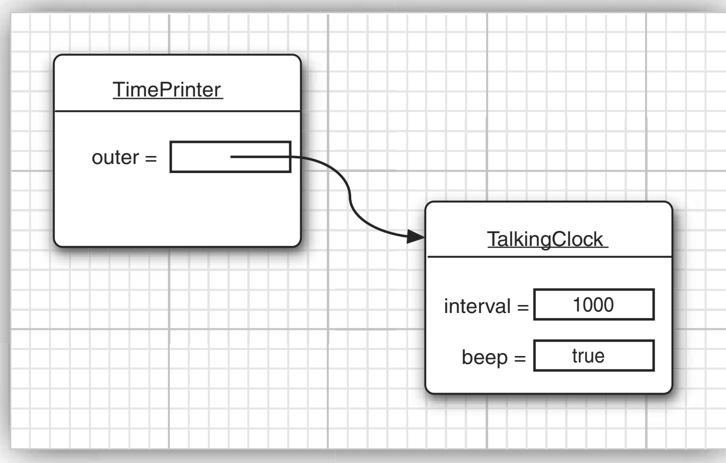

Figure 6.3 An inner class object has a reference to an outer class object

This reference is invisible in the definition of the inner class. However, to illuminate the concept, let us call the reference to the outer object *outer*. Then, the `actionPerformed` method is equivalent to the following:

```
public void actionPerformed(ActionEvent event)
{
   Date now = new Date();
   System.out.println("At the tone, the time is " + now);
   if (outer.beep) Toolkit.getDefaultToolkit().beep();
}
```

The outer class reference is set in the constructor. The compiler modifies all inner class constructors, adding a parameter for the outer class reference. The TimePrinter class defines no constructors; therefore, the compiler synthesizes a no-argument constructor, generating code like this:

```
public TimePrinter(TalkingClock clock) // automatically generated code
{
   outer = clock;
}
```

Again, please note, *outer* is not a Java keyword. We just use it to illustrate the mechanism involved in an inner class.

When a TimePrinter object is constructed in the start method, the compiler passes the this reference to the current talking clock into the constructor:

```
ActionListener listener = new TimePrinter(this); // parameter automatically added
```

Listing 6.6 shows the complete program that tests the inner class. Have another look at the access control. Had the TimePrinter class been a regular class, it would have needed to access the beep flag through a public method of the TalkingClock class. Using an inner class is an improvement. There is no need to provide accessors that are of interest only to one other class.

NOTE: We could have declared the TimePrinter class as private. Then only TalkingClock methods would be able to construct TimePrinter objects. Only inner classes can be private. Regular classes always have either package or public visibility.

Listing 6.6 innerClass/InnerClassTest.java

```
package innerClass;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.Timer;
```

(Continues)

Listing 6.6 *(Continued)*

```
/**
 * This program demonstrates the use of inner classes.
 * @version 1.10 2004-02-27
 * @author Cay Horstmann
 */
public class InnerClassTest
{
   public static void main(String[] args)
   {
      TalkingClock clock = new TalkingClock(1000, true);
      clock.start();

      // keep program running until user selects "Ok"
      JOptionPane.showMessageDialog(null, "Quit program?");
      System.exit(0);
   }
}

/**
 * A clock that prints the time in regular intervals.
 */
class TalkingClock
{
   private int interval;
   private boolean beep;

   /**
    * Constructs a talking clock
    * @param interval the interval between messages (in milliseconds)
    * @param beep true if the clock should beep
    */
   public TalkingClock(int interval, boolean beep)
   {
      this.interval = interval;
      this.beep = beep;
   }

   /**
    * Starts the clock.
    */
   public void start()
   {
      ActionListener listener = new TimePrinter();
      Timer t = new Timer(interval, listener);
      t.start();
   }
```

```
   public class TimePrinter implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         Date now = new Date();
         System.out.println("At the tone, the time is " + now);
         if (beep) Toolkit.getDefaultToolkit().beep();
      }
   }
}
```

6.4.2 Special Syntax Rules for Inner Classes

In the preceding section, we explained the outer class reference of an inner class by calling it `outer`. Actually, the proper syntax for the outer reference is a bit more complex. The expression

OuterClass`.this`

denotes the outer class reference. For example, you can write the `actionPerformed` method of the `TimePrinter` inner class as

```
public void actionPerformed(ActionEvent event)
{
   . . .
   if (TalkingClock.this.beep) Toolkit.getDefaultToolkit().beep();
}
```

Conversely, you can write the inner object constructor more explicitly, using the syntax

outerObject`.new` *InnerClass*(*construction parameters*)

For example:

```
ActionListener listener = this.new TimePrinter();
```

Here, the outer class reference of the newly constructed `TimePrinter` object is set to the `this` reference of the method that creates the inner class object. This is the most common case. As always, the `this.` qualifier is redundant. However, it is also possible to set the outer class reference to another object by explicitly naming it. For example, since `TimePrinter` is a public inner class, you can construct a `TimePrinter` for any talking clock:

```
TalkingClock jabberer = new TalkingClock(1000, true);
TalkingClock.TimePrinter listener = jabberer.new TimePrinter();
```

Note that you refer to an inner class as

OuterClass.*InnerClass*

when it occurs outside the scope of the outer class.

6.4.3 Are Inner Classes Useful? Actually Necessary? Secure?

When inner classes were added to the Java language in Java 1.1, many programmers considered them a major new feature that was out of character with the Java philosophy of being simpler than C++. The inner class syntax is undeniably complex. (It gets more complex as we study anonymous inner classes later in this chapter.) It is not obvious how inner classes interact with other features of the language, such as access control and security.

By adding a feature that was elegant and interesting rather than needed, has Java started down the road to ruin which has afflicted so many other languages?

While we won't try to answer this question completely, it is worth noting that inner classes are a phenomenon of the *compiler*, not the virtual machine. Inner classes are translated into regular class files with `$` (dollar signs) delimiting outer and inner class names, and the virtual machine does not have any special knowledge about them.

For example, the `TimePrinter` class inside the `TalkingClock` class is translated to a class file `TalkingClock$TimePrinter.class`. To see this at work, try the following experiment: run the `ReflectionTest` program of Chapter 5, and give it the class `TalkingClock$TimePrinter` to reflect upon. Alternatively, simply use the `javap` utility:

```
javap -private ClassName
```

NOTE: If you use UNIX, remember to escape the `$` character if you supply the class name on the command line. That is, run the `ReflectionTest` or `javap` program as

```
java reflection.ReflectionTest innerClass.TalkingClock\$TimePrinter
```

or

```
javap -private innerClass.TalkingClock\$TimePrinter
```

You will get the following printout:

```
public class TalkingClock$TimePrinter
{
   public TalkingClock$TimePrinter(TalkingClock);

   public void actionPerformed(java.awt.event.ActionEvent);

   final TalkingClock this$0;
}
```

You can plainly see that the compiler has generated an additional instance field, `this$0`, for the reference to the outer class. (The name `this$0` is synthesized by the compiler—you cannot refer to it in your code.) You can also see the `TalkingClock` parameter for the constructor.

If the compiler can automatically do this transformation, couldn't you simply program the same mechanism by hand? Let's try it. We would make `TimePrinter` a regular class, outside the `TalkingClock` class. When constructing a `TimePrinter` object, we pass it the `this` reference of the object that is creating it.

```
class TalkingClock
{
   . . .
   public void start()
   {
      ActionListener listener = new TimePrinter(this);
      Timer t = new Timer(interval, listener);
      t.start();
   }
}

class TimePrinter implements ActionListener
{
   private TalkingClock outer;
   . . .
   public TimePrinter(TalkingClock clock)
   {
       outer = clock;
   }
}
```

Now let us look at the `actionPerformed` method. It needs to access `outer.beep`.

```
if (outer.beep) . . . // ERROR
```

Here we run into a problem. The inner class can access the private data of the outer class, but our external `TimePrinter` class cannot.

Thus, inner classes are genuinely more powerful than regular classes because they have more access privileges.

You may well wonder how inner classes manage to acquire those added access privileges, if they are translated to regular classes with funny names—the virtual machine knows nothing at all about them. To solve this mystery, let's again use the `ReflectionTest` program to spy on the `TalkingClock` class:

```
class TalkingClock
{
   private int interval;
   private boolean beep;

   public TalkingClock(int, boolean);

   static boolean access$0(TalkingClock);
   public void start();
}
```

Notice the static `access$0` method that the compiler added to the outer class. It returns the `beep` field of the object that is passed as a parameter. (The method name might be slightly different, such as `access$000`, depending on your compiler.)

The inner class methods call that method. The statement

```
if (beep)
```

in the `actionPerformed` method of the `TimePrinter` class effectively makes the following call:

```
if (access$0(outer));
```

Is this a security risk? You bet it is. It is an easy matter for someone else to invoke the `access$0` method to read the private `beep` field. Of course, `access$0` is not a legal name for a Java method. However, hackers who are familiar with the structure of class files can easily produce a class file with virtual machine instructions to call that method, for example, by using a hex editor. Since the secret access methods have package visibility, the attack code would need to be placed inside the same package as the class under attack.

To summarize, if an inner class accesses a private data field, then it is possible to access that data field through other classes added to the package of the outer class, but to do so requires skill and determination. A programmer cannot accidentally obtain access but must intentionally build or modify a class file for that purpose.

NOTE: The synthesized constructors and methods can get quite convoluted. (Skip this note if you are squeamish.) Suppose we turn `TimePrinter` into a private inner class. There are no private classes in the virtual machine, so the compiler produces the next best thing: a package-visible class with a private constructor

```
private TalkingClock$TimePrinter(TalkingClock);
```

Of course, nobody can call that constructor, so there is a second package-visible constructor

```
TalkingClock$TimePrinter(TalkingClock, TalkingClock$1);
```

that calls the first one.

The complier translates the constructor call in the `start` method of the `TalkingClock` class to

```
new TalkingClock$TimePrinter(this, null)
```

6.4.4 Local Inner Classes

If you look carefully at the code of the `TalkingClock` example, you will find that you need the name of the type `TimePrinter` only once: when you create an object of that type in the `start` method.

In a situation like this, you can define the class *locally in a single method*.

```
public void start()
{
   class TimePrinter implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
         Date now = new Date();
         System.out.println("At the tone, the time is " + now);
         if (beep) Toolkit.getDefaultToolkit().beep();
      }
   }

   ActionListener listener = new TimePrinter();
   Timer t = new Timer(interval, listener);
   t.start();
}
```

Local classes are never declared with an access specifier (that is, `public` or `private`). Their scope is always restricted to the block in which they are declared.

Local classes have one great advantage: They are completely hidden from the outside world—not even other code in the `TalkingClock` class can access them. No method except `start` has any knowledge of the `TimePrinter` class.

6.4.5 Accessing `final` Variables from Outer Methods

Local classes have another advantage over other inner classes. Not only can they access the fields of their outer classes; they can even access local variables! However, those local variables must be declared `final`. Here is a typical example.

Let's move the interval and beep parameters from the TalkingClock constructor to the start method.

```
public void start(int interval, final boolean beep)
{
   class TimePrinter implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
          Date now = new Date();
          System.out.println("At the tone, the time is " + now);
          if (beep) Toolkit.getDefaultToolkit().beep();
      }
   }

   ActionListener listener = new TimePrinter();
   Timer t = new Timer(interval, listener);
   t.start();
}
```

Note that the TalkingClock class no longer needs to store a beep instance field. It simply refers to the beep parameter variable of the start method.

Maybe this should not be so surprising. The line

```
if (beep) . . .
```

is, after all, ultimately inside the start method, so why shouldn't it have access to the value of the beep variable?

To see why there is a subtle issue here, let's consider the flow of control more closely.

1. The start method is called.
2. The object variable listener is initialized by a call to the constructor of the inner class TimePrinter.
3. The listener reference is passed to the Timer constructor, the timer is started, and the start method exits. At this point, the beep parameter variable of the start method no longer exists.
4. A second later, the actionPerformed method executes if (beep) . . .

For the code in the actionPerformed method to work, the TimePrinter class must have copied the beep field, as a local variable of the start method, before the beep parameter value went away. That is indeed exactly what happens. In our example, the compiler synthesizes the name TalkingClock$1TimePrinter for the local inner class. If you use the ReflectionTest program again to spy on the TalkingClock$1TimePrinter class, you will get the following output:

```
class TalkingClock$1TimePrinter
{
   TalkingClock$1TimePrinter(TalkingClock, boolean);

   public void actionPerformed(java.awt.event.ActionEvent);

   final boolean val$beep;
   final TalkingClock this$0;
}
```

Note the `boolean` parameter to the constructor and the `val$beep` instance variable. When an object is created, the value `beep` is passed into the constructor and stored in the `val$beep` field. The compiler detects access of local variables, makes matching instance fields for each one of them, and copies the local variables into the constructor so that the instance fields can be initialized.

From the programmer's point of view, local variable access is quite pleasant. It makes your inner classes simpler by reducing the instance fields that you need to program explicitly.

As we already mentioned, the methods of a local class can refer only to local variables that are declared `final`. For that reason, the `beep` parameter was declared `final` in our example. A local variable that is declared `final` cannot be modified after it has been initialized. Thus, it is guaranteed that the local variable and the copy that is made inside the local class will always have the same value.

NOTE: You have seen `final` variables used for constants, such as

```
public static final double SPEED_LIMIT = 55;
```

The `final` keyword can be applied to local variables, instance variables, and static variables. In all cases it means the same thing: You can assign to this variable *once* after it has been created. Afterwards, you cannot change the value—it is final.

However, you don't have to initialize a `final` variable when you define it. For example, the `final` parameter variable `beep` is initialized once after its creation, when the `start` method is called. (If the method is called multiple times, each call has its own newly created `beep` parameter.) The `val$beep` instance variable that you can see in the `TalkingClock$1TimePrinter` inner class is set once, in the inner class constructor. A final variable that isn't initialized when it is defined is often called a *blank final* variable.

The final restriction is somewhat inconvenient. Suppose, for example, that you want to update a counter in the enclosing scope. Here, we want to count how often the compareTo method is called during sorting:

```
int counter = 0;
Date[] dates = new Date[100];
for (int i = 0; i < dates.length; i++)
   dates[i] = new Date()
      {
         public int compareTo(Date other)
         {
            counter++; // ERROR
            return super.compareTo(other);
         }
      };
Arrays.sort(dates);
System.out.println(counter + " comparisons.");
```

You can't declare counter as final because you clearly need to update it. You can't replace it with an Integer because Integer objects are immutable. The remedy is to use an array of length 1:

```
final int[] counter = new int[1];
for (int i = 0; i < dates.length; i++)
   dates[i] = new Date()
      {
         public int compareTo(Date other)
         {
            counter[0]++;
            return super.compareTo(other);
         }
      };
```

(The array variable is still declared as final, but that merely means that you can't have it refer to a different array. You are free to mutate the array elements.)

When inner classes were first invented, the prototype compiler automatically made this transformation for all local variables that were modified in the inner class. However, some programmers were fearful of having the compiler produce heap objects behind their backs, and the final restriction was adopted instead. It is possible that a future version of the Java language will revise this decision.

6.4.6 Anonymous Inner Classes

When using local inner classes, you can often go a step further. If you want to make only a single object of this class, you don't even need to give the class a name. Such a class is called an *anonymous inner class*.

```
public void start(int interval, final boolean beep)
{
   ActionListener listener = new ActionListener()
      {
         public void actionPerformed(ActionEvent event)
         {
             Date now = new Date();
             System.out.println("At the tone, the time is " + now);
             if (beep) Toolkit.getDefaultToolkit().beep();
         }
      };
   Timer t = new Timer(interval, listener);
   t.start();
}
```

This syntax is very cryptic indeed. What it means is this: Create a new object of a class that implements the `ActionListener` interface, where the required method `actionPerformed` is the one defined inside the braces `{ }`.

In general, the syntax is

```
new SuperType(construction parameters)
   {
      inner class methods and data
   }
```

Here, *SuperType* can be an interface, such as `ActionListener`; then, the inner class implements that interface. *SuperType* can also be a class; then, the inner class extends that class.

An anonymous inner class cannot have constructors because the name of a constructor must be the same as the name of a class, and the class has no name. Instead, the construction parameters are given to the *superclass* constructor. In particular, whenever an inner class implements an interface, it cannot have any construction parameters. Nevertheless, you must supply a set of parentheses as in

```
new InterfaceType()
   {
     methods and data
   }
```

You have to look carefully to see the difference between the construction of a new object of a class and the construction of an object of an anonymous inner class extending that class.

```
Person queen = new Person("Mary");
   // a Person object
Person count = new Person("Dracula") { . . .};
   // an object of an inner class extending Person
```

If the closing parenthesis of the construction parameter list is followed by an opening brace, then an anonymous inner class is being defined.

Are anonymous inner classes a great idea or are they a great way of writing obfuscated code? Probably a bit of both. When the code for an inner class is short, just a few lines of simple code, then anonymous inner classes can save typing time—but it is exactly such time-saving features that lead you down the slippery slope to "Obfuscated Java Code Contests."

Listing 6.7 contains the complete source code for the talking clock program with an anonymous inner class. If you compare this program with Listing 6.6, you will find that in this case, the solution with the anonymous inner class is quite a bit shorter, and, hopefully, with a bit of practice, as easy to comprehend.

Listing 6.7 `anonymousInnerClass/AnonymousInnerClassTest.java`

```
package anonymousInnerClass;

import java.awt.*;
import java.awt.event.*;
import java.util.*;
import javax.swing.*;
import javax.swing.Timer;

/**
 * This program demonstrates anonymous inner classes.
 * @version 1.10 2004-02-27
 * @author Cay Horstmann
 */
public class AnonymousInnerClassTest
{
   public static void main(String[] args)
   {
      TalkingClock clock = new TalkingClock();
      clock.start(1000, true);

      // keep program running until user selects "Ok"
      JOptionPane.showMessageDialog(null, "Quit program?");
      System.exit(0);
   }
}

/**
 * A clock that prints the time in regular intervals.
 */
```

```
class TalkingClock
{
   /**
    * Starts the clock.
    * @param interval the interval between messages (in milliseconds)
    * @param beep true if the clock should beep
    */
   public void start(int interval, final boolean beep)
   {
      ActionListener listener = new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               Date now = new Date();
               System.out.println("At the tone, the time is " + now);
               if (beep) Toolkit.getDefaultToolkit().beep();
            }
         };
      Timer t = new Timer(interval, listener);
      t.start();
   }
}
```

NOTE: The following trick, called *double brace initialization*, takes advantage of the inner class syntax. Suppose you want to construct an array list and pass it to a method:

```
ArrayList<String> friends = new ArrayList<>();
friends.add("Harry");
friends.add("Tony");
invite(friends);
```

If you don't need the array list again, it would be nice to make it anonymous. But then how can you add the elements? Here is how:

```
invite(new ArrayList<String>() {{ add("Harry"); add("Tony"); }})
```

Note the double braces. The outer braces make an anonymous subclass of `ArrayList`. The inner braces are an object construction block (see Chapter 4).

CAUTION: It is often convenient to make an anonymous subclass that is almost, but not quite, like its superclass. But you need to be careful with the `equals` method. In Chapter 5, we recommended that your `equals` methods use a test

```
if (getClass() != other.getClass()) return false;
```

An anonymous subclass will fail this test.

TIP: When you produce logging or debugging messages, you often want to include the name of the current class, such as

```
System.err.println("Something awful happened in " + getClass());
```

But that fails in a static method. After all, the call to `getClass` calls `this.getClass()`, and a static method has no `this`. Use the following expression instead:

```
new Object(){}.getClass().getEnclosingClass() // gets class of static method
```

Here, `new Object(){}` makes an anonymous object of an anonymous subclass of `Object`, and `getEnclosingClass` gets its enclosing class—that is, the class containing the static method.

6.4.7 Static Inner Classes

Occasionally, you may want to use an inner class simply to hide one class inside another—but you don't need the inner class to have a reference to the outer class object. You can suppress the generation of that reference by declaring the inner class `static`.

Here is a typical example of where you would want to do this. Consider the task of computing the minimum and maximum value in an array. Of course, you write one method to compute the minimum and another method to compute the maximum. When you call both methods, the array is traversed twice. It would be more efficient to traverse the array only once, computing both the minimum and the maximum simultaneously.

```
double min = Double.MAX_VALUE;
double max = Double.MIN_VALUE;
for (double v : values)
{
   if (min > v) min = v;
   if (max < v) max = v;
}
```

However, the method must return two numbers. We can achieve that by defining a class `Pair` that holds two values:

```
class Pair
{
   private double first;
   private double second;

   public Pair(double f, double s)
   {
      first = f;
      second = s;
   }
```

```
      public double getFirst() { return first; }
      public double getSecond() {  return second; }
   }
```

The minmax method can then return an object of type Pair.

```
class ArrayAlg
{
   public static Pair minmax(double[] values)
   {
      . . .
      return new Pair(min, max);
   }
}
```

The caller of the method uses the getFirst and getSecond methods to retrieve the answers:

```
Pair p = ArrayAlg.minmax(d);
System.out.println("min = " + p.getFirst());
System.out.println("max = " + p.getSecond());
```

Of course, the name Pair is an exceedingly common name, and in a large project, it is quite possible that some other programmer had the same bright idea—but made a Pair class that contains a pair of strings. We can solve this potential name clash by making Pair a public inner class inside ArrayAlg. Then the class will be known to the public as ArrayAlg.Pair:

```
ArrayAlg.Pair p = ArrayAlg.minmax(d);
```

However, unlike the inner classes that we used in previous examples, we do not want to have a reference to any other object inside a Pair object. That reference can be suppressed by declaring the inner class static:

```
class ArrayAlg
{
   public static class Pair
   {
      . . .
   }
   . . .
}
```

Of course, only inner classes can be declared static. A static inner class is exactly like any other inner class, except that an object of a static inner class does not have a reference to the outer class object that generated it. In our example, we must use a static inner class because the inner class object is constructed inside a static method:

```
public static Pair minmax(double[] d)
{
    . . .
    return new Pair(min, max);
}
```

Had the `Pair` class not been declared as `static`, the compiler would have complained that there was no implicit object of type `ArrayAlg` available to initialize the inner class object.

NOTE: Use a static inner class whenever the inner class does not need to access an outer class object. Some programmers use the term *nested class* to describe static inner classes.

NOTE: Inner classes that are declared inside an interface are automatically `static` and `public`.

Listing 6.8 contains the complete source code of the `ArrayAlg` class and the nested `Pair` class.

Listing 6.8 `staticInnerClass/StaticInnerClassTest.java`

```
package staticInnerClass;

/**
 * This program demonstrates the use of static inner classes.
 * @version 1.01 2004-02-27
 * @author Cay Horstmann
 */
public class StaticInnerClassTest
{
   public static void main(String[] args)
   {
      double[] d = new double[20];
      for (int i = 0; i < d.length; i++)
         d[i] = 100 * Math.random();
      ArrayAlg.Pair p = ArrayAlg.minmax(d);
      System.out.println("min = " + p.getFirst());
      System.out.println("max = " + p.getSecond());
   }
}
```

```
class ArrayAlg
{
   /**
    * A pair of floating-point numbers
    */
   public static class Pair
   {
      private double first;
      private double second;

      /**
       * Constructs a pair from two floating-point numbers
       * @param f the first number
       * @param s the second number
       */
      public Pair(double f, double s)
      {
         first = f;
         second = s;
      }

      /**
       * Returns the first number of the pair
       * @return the first number
       */
      public double getFirst()
      {
         return first;
      }

      /**
       * Returns the second number of the pair
       * @return the second number
       */
      public double getSecond()
      {
         return second;
      }
   }

   /**
    * Computes both the minimum and the maximum of an array
    * @param values an array of floating-point numbers
    * @return a pair whose first element is the minimum and whose second element
    * is the maximum
    */
```

(Continues)

Listing 6.8 *(Continued)*

```
   public static Pair minmax(double[] values)
   {
      double min = Double.MAX_VALUE;
      double max = Double.MIN_VALUE;
      for (double v : values)
      {
         if (min > v) min = v;
         if (max < v) max = v;
      }
      return new Pair(min, max);
   }
}
```

6.5 Proxies

In the final section of this chapter, we discuss *proxies*. You can use a proxy to create, at runtime, new classes that implement a given set of interfaces. Proxies are only necessary when you don't yet know at compile time which interfaces you need to implement. This is not a common situation for application programmers, and you should feel free to skip this section if you are not interested in advanced wizardry. However, for certain systems programming applications, the flexibility that proxies offer can be very important.

Suppose you want to construct an object of a class that implements one or more interfaces whose exact nature you may not know at compile time. This is a difficult problem. To construct an actual class, you can simply use the `newInstance` method or use reflection to find a constructor. But you can't instantiate an interface. You need to define a new class in a running program.

To overcome this problem, some programs generate code, place it into a file, invoke the compiler, and then load the resulting class file. Naturally, this is slow, and it also requires deployment of the compiler together with the program. The *proxy* mechanism is a better solution. The proxy class can create brand-new classes at runtime. Such a proxy class implements the interfaces that you specify. In particular, the proxy class has the following methods:

- All methods required by the specified interfaces; and
- All methods defined in the `Object` class (`toString`, `equals`, and so on).

However, you cannot define new code for these methods at runtime. Instead, you must supply an *invocation handler*. An invocation handler is an object of any class that implements the `InvocationHandler` interface. That interface has a single method:

```
Object invoke(Object proxy, Method method, Object[] args)
```

Whenever a method is called on the proxy object, the `invoke` method of the invocation handler gets called, with the `Method` object and parameters of the original call. The invocation handler must then figure out how to handle the call.

To create a proxy object, use the `newProxyInstance` method of the `Proxy` class. The method has three parameters:

- A *class loader*. As part of the Java security model, different class loaders can be used for system classes, classes that are downloaded from the Internet, and so on. We will discuss class loaders in Chapter 9 of Volume II. For now, we specify `null` to use the default class loader.
- An array of `Class` objects, one for each interface to be implemented.
- An invocation handler.

There are two remaining questions. How do we define the handler? And what can we do with the resulting proxy object? The answers depend, of course, on the problem that we want to solve with the proxy mechanism. Proxies can be used for many purposes, such as

- Routing method calls to remote servers
- Associating user interface events with actions in a running program
- Tracing method calls for debugging purposes

In our example program, we use proxies and invocation handlers to trace method calls. We define a `TraceHandler` wrapper class that stores a wrapped object. Its `invoke` method simply prints the name and parameters of the method to be called and then calls the method with the wrapped object as the implicit parameter.

```
class TraceHandler implements InvocationHandler
{
   private Object target;

   public TraceHandler(Object t)
   {
      target = t;
   }
```

```
    public Object invoke(Object proxy, Method m, Object[] args)
       throws Throwable
    {
       // print method name and parameters
       . . .
       // invoke actual method
       return m.invoke(target, args);
    }
 }
```

Here is how you construct a proxy object that causes the tracing behavior whenever one of its methods is called:

```
Object value = . . .;
// construct wrapper
InvocationHandler handler = new TraceHandler(value);
// construct proxy for one or more interfaces
Class[] interfaces = new Class[] { Comparable.class};
Object proxy = Proxy.newProxyInstance(null, interfaces, handler);
```

Now, whenever a method from one of the interfaces is called on `proxy`, the method name and parameters are printed out and the method is then invoked on `value`.

In the program shown in Listing 6.9, we use proxy objects to trace a binary search. We fill an array with proxies to the integers `1 . . . 1000`. Then we invoke the `binarySearch` method of the `Arrays` class to search for a random integer in the array. Finally, we print the matching element.

```
Object[] elements = new Object[1000];
// fill elements with proxies for the integers 1 . . . 1000
for (int i = 0; i < elements.length; i++)
{
   Integer value = i + 1;
   elements[i] = Proxy.newProxyInstance(. . .); // proxy for value;
}

// construct a random integer
Integer key = new Random().nextInt(elements.length) + 1;

// search for the key
int result = Arrays.binarySearch(elements, key);

// print match if found
if (result >= 0) System.out.println(elements[result]);
```

The `Integer` class implements the `Comparable` interface. The proxy objects belong to a class that is defined at runtime. (It has a name such as `$Proxy0`.) That class also implements the `Comparable` interface. However, its `compareTo` method calls the `invoke` method of the proxy object's handler.

NOTE: As you saw earlier in this chapter, the `Integer` class actually implements `Comparable<Integer>`. However, at runtime, all generic types are erased and the proxy is constructed with the class object for the raw `Comparable` class.

The `binarySearch` method makes calls like this:

```
if (elements[i].compareTo(key) < 0) . . .
```

Since we filled the array with proxy objects, the `compareTo` calls call the `invoke` method of the `TraceHandler` class. That method prints the method name and parameters and then invokes `compareTo` on the wrapped `Integer` object.

Finally, at the end of the sample program, we call

```
System.out.println(elements[result]);
```

The `println` method calls `toString` on the proxy object, and that call is also redirected to the invocation handler.

Here is the complete trace of a program run:

```
500.compareTo(288)
250.compareTo(288)
375.compareTo(288)
312.compareTo(288)
281.compareTo(288)
296.compareTo(288)
288.compareTo(288)
288.toString()
```

You can see how the binary search algorithm homes in on the key by cutting the search interval in half in every step. Note that the `toString` method is proxied even though it does not belong to the `Comparable` interface—as you will see in the next section, certain `Object` methods are always proxied.

Listing 6.9 `proxy/ProxyTest.java`

```
package proxy;

import java.lang.reflect.*;
import java.util.*;
/**
 * This program demonstrates the use of proxies.
 * @version 1.00 2000-04-13
 * @author Cay Horstmann
 */
```

(Continues)

Listing 6.9 *(Continued)*

```
public class ProxyTest
{
   public static void main(String[] args)
   {
      Object[] elements = new Object[1000];

      // fill elements with proxies for the integers 1 ... 1000
      for (int i = 0; i < elements.length; i++)
      {
         Integer value = i + 1;
         InvocationHandler handler = new TraceHandler(value);
         Object proxy = Proxy.newProxyInstance(null, new Class[] { Comparable.class } , handler);
         elements[i] = proxy;
      }

      // construct a random integer
      Integer key = new Random().nextInt(elements.length) + 1;

      // search for the key
      int result = Arrays.binarySearch(elements, key);

      // print match if found
      if (result >= 0) System.out.println(elements[result]);
   }
}

/**
 * An invocation handler that prints out the method name and parameters, then
 * invokes the original method
 */
class TraceHandler implements InvocationHandler
{
   private Object target;

   /**
    * Constructs a TraceHandler
    * @param t the implicit parameter of the method call
    */
   public TraceHandler(Object t)
   {
      target = t;
   }

   public Object invoke(Object proxy, Method m, Object[] args) throws Throwable
   {
      // print implicit argument
      System.out.print(target);
```

```
        // print method name
        System.out.print("." + m.getName() + "(");

        // print explicit arguments
        if (args != null)
        {
           for (int i = 0; i < args.length; i++)
           {
              System.out.print(args[i]);
              if (i < args.length - 1) System.out.print(", ");
           }
        }
        System.out.println(")");

        // invoke actual method
        return m.invoke(target, args);
     }
  }
```

6.5.1 Properties of Proxy Classes

Now that you have seen proxy classes in action, let's go over some of their properties. Remember that proxy classes are created on the fly in a running program. However, once they are created, they are regular classes, just like any other classes in the virtual machine.

All proxy classes extend the class `Proxy`. A proxy class has only one instance field—the invocation handler, which is defined in the `Proxy` superclass. Any additional data required to carry out the proxy objects' tasks must be stored in the invocation handler. For example, when we proxied `Comparable` objects in the program shown in Listing 6.9, the `TraceHandler` wrapped the actual objects.

All proxy classes override the `toString`, `equals`, and `hashCode` methods of the `Object` class. Like all proxy methods, these methods simply call `invoke` on the invocation handler. The other methods of the `Object` class (such as `clone` and `getClass`) are not redefined.

The names of proxy classes are not defined. The `Proxy` class in Oracle's virtual machine generates class names that begin with the string `$Proxy`.

There is only one proxy class for a particular class loader and ordered set of interfaces. That is, if you call the `newProxyInstance` method twice with the same class loader and interface array, you get two objects of the same class. You can also obtain that class with the `getProxyClass` method:

```
Class proxyClass = Proxy.getProxyClass(null, interfaces);
```

A proxy class is always `public` and `final`. If all interfaces that the proxy class implements are `public`, the proxy class does not belong to any particular package.

Otherwise, all non-`public` interfaces must belong to the same package, and the proxy class will also belong to that package.

You can test whether a particular `Class` object represents a proxy class by calling the `isProxyClass` method of the `Proxy` class.

`java.lang.reflect.InvocationHandler` 1.3

- `Object invoke(Object proxy, Method method, Object[] args)`
  define this method to contain the action that you want carried out whenever a method was invoked on the proxy object.

`java.lang.reflect.Proxy` 1.3

- `static Class getProxyClass(ClassLoader loader, Class[] interfaces)`
  returns the proxy class that implements the given interfaces.
- `static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler handler)`
  constructs a new instance of the proxy class that implements the given interfaces. All methods call the `invoke` method of the given handler object.
- `static boolean isProxyClass(Class c)`
  returns `true` if `c` is a proxy class.

This ends our final chapter on the fundamentals of the Java programming language. Interfaces and inner classes are concepts that you will encounter frequently. However, as we already mentioned, proxies are an advanced technique that is of interest mainly to tool builders, not application programmers. You are now ready to go on to learn about graphics and user interfaces, starting with Chapter 7.

This page intentionally left blank

Index

Symbols

A

E

F

K

L