PRENTICE HALL

CORE JAVA®

Volume I—Fundamentals

TENTH EDITION

CAY S. HORSTMANN

Core Java®

Volume I—Fundamentals

Tenth Edition

This page intentionally left blank

Core Java®

Volume I—Fundamentals

Tenth Edition

Cay S. Horstmann

PRENTICE HALL

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
Sao Paulo • Sidney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the United States, please contact international@pearsoned.com.

Visit us on the Web: informit.com/ph

Library of Congress Cataloging-in-Publication Data
Names: Horstmann, Cay S., 1959- author.
Title: Core Java / Cay S. Horstmann.
Description: Tenth edition. | New York : Prentice Hall, [2016] | Includes index.
Identifiers: LCCN 2015038763| ISBN 9780134177304 (volume 1 : pbk. : alk. paper) | ISBN 0134177304 (volume 1 : pbk. : alk. paper)
Subjects: LCSH: Java (Computer program language)
Classification: LCC QA76.73.J38 H6753 2016 | DDC 005.13/3—dc23
LC record available at http://lccn.loc.gov/2015038763

ISBN-13: 978-0-13-417730-4
ISBN-10: 0-13-417730-4

Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.
First printing, December 2015

Contents

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 wide acceptance. Its built-in security and safety features are reassuring both to programmers and to the users of Java programs. Java has built-in support for advanced programming tasks, such as network programming, database connectivity, and concurrency.

Since 1995, nine major revisions of the Java Development Kit have been released. Over the course of the last 20 years, the Application Programming Interface (API) has grown from about 200 to over 4,000 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 tenth 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) 8.

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
- Exception handling
- Generic programming
- The collections framework
- The event listener model
- Graphical user interface design with the Swing UI toolkit
- 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*, goes further into the enterprise features and advanced user-interface programming. It includes detailed discussions of

- The Stream API
- File processing and regular expressions
- Databases
- XML processing
- Annotations
- Internationalization
- Network programming
- Advanced GUI components
- Advanced graphics
- Native methods

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, bug 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. After we cover interfaces, we move on to *lambda expressions*, a concise way for expressing a block of code that can be executed at a later point in time. We then cover a useful technical feature of Java called *inner classes*.

Chapter 7 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 final part of this chapter, we give you a number of useful debugging tips.

Chapter 8 gives an overview of generic programming. 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 9** 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 10 starts the coverage of GUI programming. 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 11 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 such as buttons and panels.

Chapter 12 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 13 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. We also explain how Java programs can store and retrieve configuration information once they have been deployed.

Chapter 14 finishes the book with a discussion of concurrency, which enables you to program tasks to be done in parallel. This is an important and exciting application of Java technology in an era where most processors have multiple cores that you want to keep busy.

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, as shown in the following example:

`Application Programming Interface` 1.2

Programs whose source code is on the book's companion web site are presented 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 on 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 who went over the manuscript with an amazing eye for detail and saved me from many embarrassing errors.

Reviewers of this and earlier editions include Chuck Allison (Utah Valley University), Lance Andersen (Oracle), Paul Anderson (Anderson Software Group), Alec Beaton (IBM), Cliff Berg, Andrew Binstock (Oracle), 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), Yoshiki Shibata, 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, Christian Ullenboom, Burt Walsh, Dan Xu (Oracle), and John Zavgren (Oracle).

Cay Horstmann
Biel/Bienne, Switzerland
November 2015

CHAPTER 6

Interfaces, Lambda Expressions, 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 move on to *lambda expressions*, a concise way for expressing a block of code that can be executed at a later point in time. Using lambda expressions, you can express code that uses callbacks or variable behavior in an elegant and concise fashion.

We then discuss 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.

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 following sections, you will learn what Java interfaces are and how to use them. You will also find out how interfaces have been made more powerful in Java SE 8.

6.1.1 The Interface Concept

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. Then the `compareTo` method has a parameter of type `Object`, and you have to manually cast that parameter of the `compareTo` method to the desired type. We will do just that for a little while so that you don't have to worry about two new concepts at the same time.

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. Before Java SE 8, methods were never implemented in interfaces. (As you will see in Section 6.1.4, "Static Methods," on p. 298 and Section 6.1.5, "Default Methods," on p. 298, it is now possible to supply simple methods in interfaces. Of course, those methods cannot refer to instance fields—interfaces don't have any.)

Supplying instance fields and methods that operate on them 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 more restrictive access privilege.

We can do a little better by supplying a type parameter for the generic `Comparable` interface:

```
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. Otherwise, call the static `Integer.compare` method.

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`.

NOTE: The documentation of the `Comparable` interface suggests that the `compareTo` method should be compatible with the `equals` method. That is, `x.compareTo(y)` should be zero exactly when `x.equals(y)`. Most classes in the Java API that implement `Comparable` follow this advice. A notable exception is `BigDecimal`. Consider `x = new BigDecimal("1.0")` and `y = new BigDecimal("1.00")`. Then `x.equals(y)` is `false` because the numbers differ in precision. But `x.compareTo(y)` is zero. Ideally, it shouldn't be, but there was no obvious way of deciding which one should come first.

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`

```
package interfaces;

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

   public Employee(String name, double salary)
   {
      this.name = name;
      this.salary = salary;
   }

   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`. 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 1.0

- `static int compare(int x, int y)` 7

  returns a negative integer if `x < y`, zero if `x` and `y` are equal, and a positive integer otherwise.

java.lang.Double 1.0

- `static int compare(double x, double y)` 1.4

  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 you want managers to be better than regular employees, regardless of 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.

6.1.2 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 Section 6.2.3, "Object Cloning," on p. 306.) If your class implements `Cloneable`, the `clone` method in the `Object` class will make an exact copy of your class's objects. If you want both cloneability and comparability, simply implement both interfaces. Use commas to separate the interfaces that you want to implement:

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

6.1.3 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 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 superclass and additional interfaces.

6.1.4 Static Methods

As of Java SE 8, you are allowed to add static methods to interfaces. There was never a technical reason why this should be outlawed. It simply seemed to be against the spirit of interfaces as abstract specifications.

Up to now, it has been common to place static methods in companion classes. In the standard library, you find pairs of interfaces and utility classes such as `Collection`/`Collections` or `Path`/`Paths`.

Have a look at the `Paths` class. It only has a couple of factory methods. You can construct a path to a file or directory from a sequence of strings, such as `Paths.get("jdk1.8.0", "jre", "bin")`. In Java SE 8, one could have added this method to the `Path` interface:

```
public interface Path
{
   public static Path get(String first, String... more) {
      return FileSystems.getDefault().getPath(first, more);
   }
   . . .
}
```

Then the `Paths` class is no longer necessary.

It is unlikely that the Java library will be refactored in this way, but when you implement your own interfaces, there is no longer a reason to provide a separate companion class for utility methods.

6.1.5 Default Methods

You can supply a *default* implementation for any interface method. You must tag such a method with the `default` modifier.

```
public interface Comparable<T>
{
    default int compareTo(T other) { return 0; }
        // By default, all elements are the same
}
```

Of course, that is not very useful since every realistic implementation of `Comparable` would override this method. But there are other situations where default methods can be useful. For example, as you will see in Chapter 11, if you want to be notified when a mouse click happens, you are supposed to implement an interface that has five methods:

```
public interface MouseListener
{
   void mouseClicked(MouseEvent event);
   void mousePressed(MouseEvent event);
   void mouseReleased(MouseEvent event);
   void mouseEntered(MouseEvent event);
   void mouseExited(MouseEvent event);
}
```

Most of the time, you only care about one or two of these event types. As of Java SE 8, you can declare all of the methods as default methods that do nothing.

```
public interface MouseListener
{
   default void mouseClicked(MouseEvent event) {}
   default void mousePressed(MouseEvent event) {}
   default void mouseReleased(MouseEvent event) {}
   default void mouseEntered(MouseEvent event) {}
   default void mouseExited(MouseEvent event) {}
}
```

Then programmers who implement this interface only need to override the listeners for the events they actually care about.

A default method can call other methods. For example, a `Collection` interface can define a convenience method

```
public interface Collection
{
   int size(); // An abstract method
   default boolean isEmpty()
   {
      return size() == 0;
   }
   . . .
}
```

Then a programmer implementing `Collection` doesn't have to worry about implementing an `isEmpty` method.

NOTE: In the Java API, you will find a number of interfaces with companion classes that implement some or all of its methods, such as `Collection/AbstractCollection` or `MouseListener/MouseAdapter`. With Java SE 8, this technique is obsolete. Just implement the methods in the interface.

An important use for default methods is *interface evolution*. Consider for example the `Collection` interface that has been a part of Java for many years. Suppose that a long time ago, you provided a class

```
public class Bag implements Collection
```

Later, in Java SE 8, a `stream` method was added to the interface.

Suppose the `stream` method was not a default method. Then the `Bag` class no longer compiles since it doesn't implement the new method. Adding a nondefault method to an interface is not *source compatible*.

But suppose you don't recompile the class and simply use an old JAR file containing it. The class will still load, even with the missing method. Programs can still construct `Bag` instances, and nothing bad will happen. (Adding a method to an interface is *binary compatible*.) However, if a program calls the `stream` method on a `Bag` instance, an `AbstractMethodError` occurs.

Making the method a `default` method solves both problems. The `Bag` class will again compile. And if the class is loaded without being recompiled and the `stream` method is invoked on a `Bag` instance, the `Collection.stream` method is called.

6.1.6 Resolving Default Method Conflicts

What happens if the exact same method is defined as a default method in one interface and then again as a method of a superclass or another interface? Languages such as Scala and C++ have complex rules for resolving such ambiguities. Fortunately, the rules in Java are much simpler. Here they are:

1. Superclasses win. If a superclass provides a concrete method, default methods with the same name and parameter types are simply ignored.
2. Interfaces clash. If a superinterface provides a default method, and another interface supplies a method with the same name and parameter types (default or not), then you must resolve the conflict by overriding that method.

Let's look at the second rule. Consider another interface with a `getName` method:

```
interface Named
{
   default String getName() { return getClass().getName() + "_" + hashCode(); }
}
```

What happens if you form a class that implements both of them?

```
class Student implements Person, Named
{
   . . .
}
```

The class inherits two inconsistent `getName` methods provided by the `Person` and `Named` interfaces. Instead of choosing one over the other, the Java compiler reports an error and leaves it up to the programmer to resolve the ambiguity. Simply provide a `getName` method in the `Student` class. In that method, you can choose one of the two conflicting methods, like this:

```
class Student implements Person, Named
{
   public String getName() { return Person.super.getName(); }
   . . .
}
```

Now assume that the `Named` interface does not provide a default implementation for `getName`:

```
interface Named
{
   String getName();
}
```

Can the `Student` class inherit the default method from the `Person` interface? This might be reasonable, but the Java designers decided in favor of uniformity. It doesn't matter how two interfaces conflict. If at least one interface provides an implementation, the compiler reports an error, and the programmer must resolve the ambiguity.

NOTE: Of course, if neither interface provides a default for a shared method, then we are in the situation before Java SE 8, and there is no conflict. An implementing class has two choices: implement the method, or leave it unimplemented. In the latter case, the class is itself abstract.

We just discussed name clashes between two interfaces. Now consider a class that extends a superclass and implements an interface, inheriting the same method from both. For example, suppose that `Person` is a class and `Student` is defined as

```
class Student extends Person implements Named { . . . }
```

In that case, only the superclass method matters, and any default method from the interface is simply ignored. In our example, `Student` inherits the `getName` method from `Person`, and it doesn't make any difference whether the `Named` interface provides a default for `getName` or not. This is the "class wins" rule.

The "class wins" rule ensures compatibility with Java SE 7. If you add default methods to an interface, it has no effect on code that worked before there were default methods.

CAUTION: You can never make a default method that redefines one of the methods in the `Object` class. For example, you can't define a default method for `toString` or `equals`, even though that might be attractive for interfaces such as `List`. As a consequence of the "classes win" rule, such a method could never win against `Object.toString` or `Objects.equals`.

6.2 Examples of Interfaces

In the next three sections, we give additional examples of interfaces so you can see how they are used in practice.

6.2.1 Interfaces and Callbacks

A common pattern in programming is the *callback* pattern. In this pattern, you 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, 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.

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)
   {
      System.out.println("At the tone, the time is " + new Date());
      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 11 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 Wed Apr 13 23:29:08 PDT 2016
```

is displayed, followed by a beep.

Listing 6.3 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 6.3 `timer/TimerTest.java`

```
package timer;

/**
   @version 1.01 2015-05-12
   @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)
   {
      System.out.println("At the tone, the time is " + new Date());
      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.

6.2.2 The `Comparator` Interface

In Section 6.1.1, "The Interface Concept," on p. 288, you have seen how you can sort an array of objects, provided they are instances of classes that implement the `Comparable` interface. For example, you can sort an array of strings since the `String` class implements `Comparable<String>`, and the `String.compareTo` method compares strings in dictionary order.

Now suppose we want to sort strings by increasing length, not in dictionary order. We can't have the `String` class implement the `compareTo` method in two ways—and at any rate, the `String` class isn't ours to modify.

To deal with this situation, there is a second version of the `Arrays.sort` method whose parameters are an array and a *comparator*—an instance of a class that implements the `Comparator` interface.

```
public interface Comparator<T>
{
   int compare(T first, T second);
}
```

To compare strings by length, define a class that implements `Comparator<String>`:

```
class LengthComparator implements Comparator<String>
{
   public int compare(String first, String second) {
      return first.length() - second.length();
   }
}
```

To actually do the comparison, you need to make an instance:

```
Comparator<String> comp = new LengthComparator();
if (comp.compare(words[i], words[j]) > 0) . . .
```

Contrast this call with `words[i].compareTo(words[j])`. The `compare` method is called on the comparator object, not the string itself.

NOTE: Even though the `LengthComparator` object has no state, you still need to make an instance of it. You need the instance to call the `compare` method—it is not a static method.

To sort an array, pass a `LengthComparator` object to the `Arrays.sort` method:

```
String[] friends = { "Peter", "Paul", "Mary" };
Arrays.sort(friends, new LengthComparator());
```

Now the array is either `["Paul", "Mary", "Peter"]` or `["Mary", "Paul", "Peter"]`.

You will see in Section 6.3, “Lambda Expressions,” on p. 314 how to use a `Comparator` much more easily with a lambda expression.

6.2.3 Object Cloning

In this section, we discuss the `Cloneable` interface that indicates that a class has provided a safe `clone` method. Since cloning is not all that common, and the details are quite technical, you may just want to glance at this material until you need it.

To understand what cloning means, recall what happens when you make a copy of a variable holding an object reference. 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.

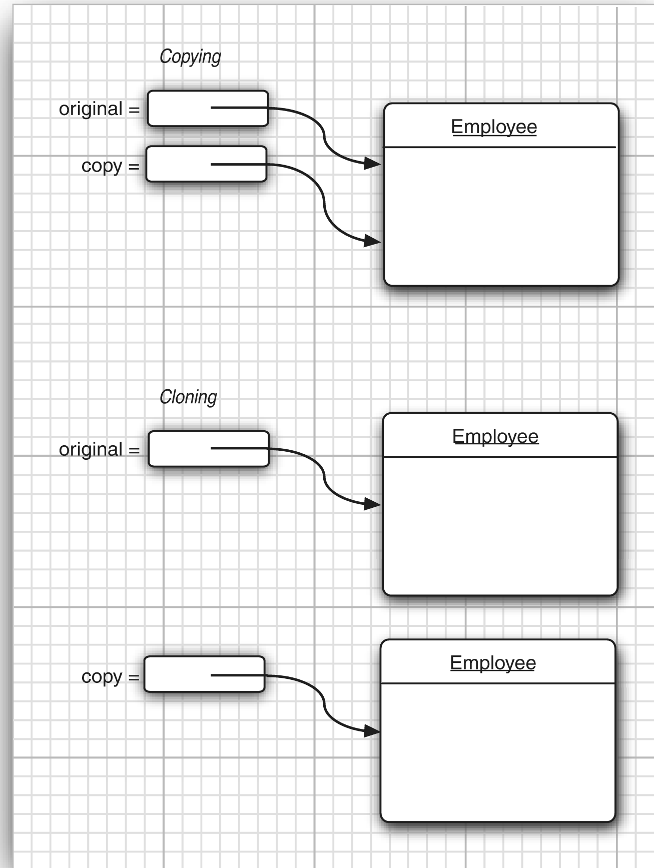

Figure 6.1 Copying and cloning

```
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, 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. (The figure shows a shared `Date` object. For reasons that will become clear shortly, this example uses a version of the `Employee` class in which the hire day is represented as a `Date`.)

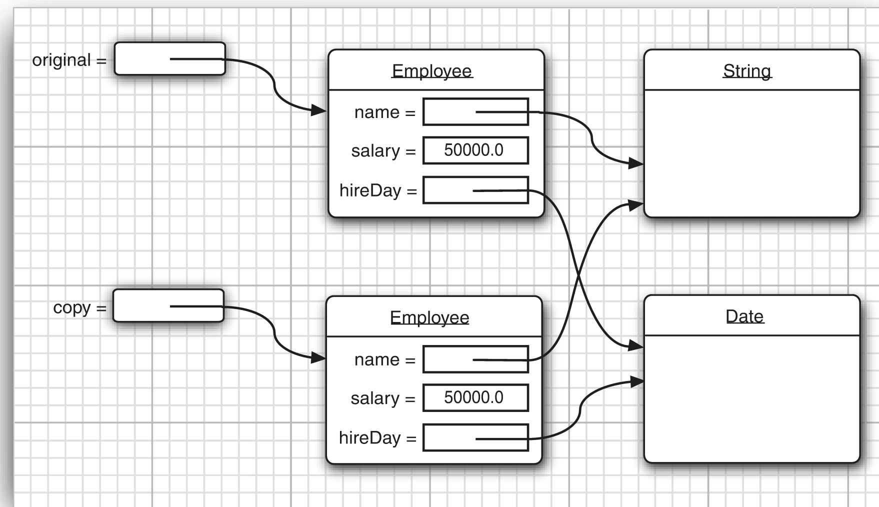

Figure 6.2 A shallow copy

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`. 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, so it too must be cloned. (For that reason, this example uses a field of type `Date`, not `LocalDate`, to demonstrate the cloning process. Had `hireDay` been an instance of the immutable `LocalDate` class, no further action would have been required.)

For every class, you need to decide whether

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: Up to Java SE 1.4, the `clone` method always had return type `Object`. Nowadays, you can specify the correct return type for your `clone` methods. This is an example of covariant return types (see Chapter 5).

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
   {
      Employee cloned =  (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 6.4 clones an instance of the class `Employee` (Listing 6.5), 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.

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 2 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.

Listing 6.4 `clone/CloneTest.java`

```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 6.5 `clone/Employee.java`

```java
package clone;

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

public class Employee implements Cloneable
{
```

```
   private String name;
   private double salary;
   private Date hireDay;

   public Employee(String name, double salary)
   {
      this.name = name;
      this.salary = salary;
      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 + "]";
   }
}
```

6.3 Lambda Expressions

Now you are ready to learn about lambda expressions, the most exciting change to the Java language in many years. You will see how to use lambda expressions for defining blocks of code with a concise syntax, and how to write code that consumes lambda expressions.

6.3.1 Why Lambdas?

A lambda expression is a block of code that you can pass around so it can be executed later, once or multiple times. Before getting into the syntax (or even the curious name), let's step back and observe where we have used such code blocks in Java.

In Section 6.2.1, "Interfaces and Callbacks," on p. 302, you saw how to do work in timed intervals. Put the work into the `actionPerformed` method of an `ActionListener`:

```
class Worker implements ActionListener
{
   public void actionPerformed(ActionEvent event)
   {
      // do some work
   }
}
```

Then, when you want to repeatedly execute this code, you construct an instance of the `Worker` class. You then submit the instance to a `Timer` object.

The key point is that the `actionPerformed` method contains code that you want to execute later.

Or consider sorting with a custom comparator. If you want to sort strings by length instead of the default dictionary order, you can pass a `Comparator` object to the `sort` method:

```
class LengthComparator implements Comparator<String>
{
   public int compare(String first, String second)
   {
      return first.length() - second.length();
   }
}
. . .
Arrays.sort(strings, new LengthComparator());
```

The `compare` method isn't called right away. Instead, the `sort` method keeps calling the `compare` method, rearranging the elements if they are out of order, until the array is sorted. You give the `sort` method a snippet of code needed to compare elements, and that code is integrated into the rest of the sorting logic, which you'd probably not care to reimplement.

Both examples have something in common. A block of code was passed to someone—a timer, or a `sort` method. That code block was called at some later time.

Up to now, giving someone a block of code hasn't been easy in Java. You couldn't just pass code blocks around. Java is an object-oriented language, so you had to construct an object belonging to a class that has a method with the desired code.

In other languages, it is possible to work with blocks of code directly. The Java designers have resisted adding this feature for a long time. After all, a great strength of Java is its simplicity and consistency. A language can become an unmaintainable mess if it includes every feature that yields marginally more concise code. However, in those other languages it isn't just easier to spawn a thread or to register a button click handler; large swaths of their APIs are simpler, more consistent, and more powerful. In Java, one could have written similar APIs that take objects of classes implementing a particular function, but such APIs would be unpleasant to use.

For some time now, the question was not whether to augment Java for functional programming, but how to do it. It took several years of experimentation before a design emerged that is a good fit for Java. In the next section, you will see how you can work with blocks of code in Java SE 8.

6.3.2 The Syntax of Lambda Expressions

Consider again the sorting example from the preceding section. We pass code that checks whether one string is shorter than another. We compute

```
first.length() - second.length()
```

What are `first` and `second`? They are both strings. Java is a strongly typed language, and we must specify that as well:

```
(String first, String second)
   -> first.length() - second.length()
```

You have just seen your first *lambda expression*. Such an expression is simply a block of code, together with the specification of any variables that must be passed to the code.

Why the name? Many years ago, before there were any computers, the logician Alonzo Church wanted to formalize what it means for a mathematical function to be effectively computable. (Curiously, there are functions that are known to exist, but nobody knows how to compute their values.) He used the Greek letter lambda (λ) to mark parameters. Had he known about the Java API, he would have written

```
λfirst.λsecond.first.length() - second.length()
```

NOTE: Why the letter λ? Did Church run out of other letters of the alphabet? Actually, the venerable *Principia Mathematica* used the ^ accent to denote free variables, which inspired Church to use an uppercase lambda Λ for parameters. But in the end, he switched to the lowercase version. Ever since, an expression with parameter variables has been called a lambda expression.

You have just seen one form of lambda expressions in Java: parameters, the -> arrow, and an expression. If the code carries out a computation that doesn't fit in a single expression, write it exactly like you would have written a method: enclosed in {} and with explicit `return` statements. For example,

```
(String first, String second) ->
   {
      if (first.length() < second.length()) return -1;
      else if (first.length() > second.length()) return 1;
      else return 0;
   }
```

If a lambda expression has no parameters, you still supply empty parentheses, just as with a parameterless method:

```
() -> { for (int i = 100; i >= 0; i--) System.out.println(i); }
```

If the parameter types of a lambda expression can be inferred, you can omit them. For example,

```
Comparator<String> comp
   = (first, second) // Same as (String first, String second)
      -> first.length() - second.length();
```

Here, the compiler can deduce that `first` and `second` must be strings because the lambda expression is assigned to a string comparator. (We will have a closer look at this assignment in the next section.)

If a method has a single parameter with inferred type, you can even omit the parentheses:

```
ActionListener listener = event ->
   System.out.println("The time is " + new Date()");
      // Instead of (event) -> . . . or (ActionEvent event) -> . . .
```

You never specify the result type of a lambda expression. It is always inferred from context. For example, the expression

```
(String first, String second) -> first.length() - second.length()
```

can be used in a context where a result of type `int` is expected.

NOTE: It is illegal for a lambda expression to return a value in some branches but not in others. For example, `(int x) -> { if (x >= 0) return 1; }` is invalid.

The program in Listing 6.6 shows how to use lambda expressions for a comparator and an action listener.

Listing 6.6 `lambda/LambdaTest.java`

```java
package lambda;

import java.util.*;

import javax.swing.*;
import javax.swing.Timer;

/**
 * This program demonstrates the use of lambda expressions.
 * @version 1.0 2015-05-12
 * @author Cay Horstmann
 */
public class LambdaTest
{
   public static void main(String[] args)
   {
      String[] planets = new String[] { "Mercury", "Venus", "Earth", "Mars",
            "Jupiter", "Saturn", "Uranus", "Neptune" };
      System.out.println(Arrays.toString(planets));
      System.out.println("Sorted in dictionary order:");
      Arrays.sort(planets);
      System.out.println(Arrays.toString(planets));
      System.out.println("Sorted by length:");
      Arrays.sort(planets, (first, second) -> first.length() - second.length());
      System.out.println(Arrays.toString(planets));

      Timer t = new Timer(1000, event ->
         System.out.println("The time is " + new Date()));
      t.start();

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

6.3.3 Functional Interfaces

As we discussed, there are many existing interfaces in Java that encapsulate blocks of code, such as `ActionListener` or `Comparator`. Lambdas are compatible with these interfaces.

You can supply a lambda expression whenever an object of an interface with a single abstract method is expected. Such an interface is called a *functional interface*.

> **NOTE:** You may wonder why a functional interface must have a single *abstract* method. Aren't all methods in an interface abstract? Actually, it has always been possible for an interface to redeclare methods from the `Object` class such as `toString` or `clone`, and these declarations do not make the methods abstract. (Some interfaces in the Java API redeclare `Object` methods in order to attach javadoc comments. Check out the `Comparator` API for an example.) More importantly, as you saw in Section 6.1.5, "Default Methods," on p. 298, in Java SE 8, interfaces can declare nonabstract methods.

To demonstrate the conversion to a functional interface, consider the `Arrays.sort` method. Its second parameter requires an instance of `Comparator`, an interface with a single method. Simply supply a lambda:

```
Arrays.sort(words,
   (first, second) -> first.length() - second.length());
```

Behind the scenes, the `Arrays.sort` method receives an object of some class that implements `Comparator<String>`. Invoking the `compare` method on that object executes the body of the lambda expression. The management of these objects and classes is completely implementation dependent, and it can be much more efficient than using traditional inner classes. It is best to think of a lambda expression as a function, not an object, and to accept that it can be passed to a functional interface.

This conversion to interfaces is what makes lambda expressions so compelling. The syntax is short and simple. Here is another example:

```
Timer t = new Timer(1000, event ->
   {
      System.out.println("At the tone, the time is " + new Date());
      Toolkit.getDefaultToolkit().beep();
   });
```

That's a lot easier to read than the alternative with a class that implements the `ActionListener` interface.

In fact, conversion to a functional interface is the *only* thing that you can do with a lambda expression in Java. In other programming languages that support function literals, you can declare function types such as `(String, String) -> int`, declare variables of those types, and use the variables to save function expressions. However, the Java designers decided to stick with the familiar concept of interfaces instead of adding function types to the language.

> **NOTE:** You can't even assign a lambda expression to a variable of type `Object`—`Object` is not a functional interface.

The Java API defines a number of very generic functional interfaces in the `java.util.function` package. One of the interfaces, `BiFunction<T, U, R>`, describes functions with parameter types `T` and `U` and return type `R`. You can save our string comparison lambda in a variable of that type:

```
BiFunction<String, String, Integer> comp
   = (first, second) -> first.length() - second.length();
```

However, that does not help you with sorting. There is no `Arrays.sort` method that wants a `BiFunction`. If you have used a functional programming language before, you may find this curious. But for Java programmers, it's pretty natural. An interface such as `Comparator` has a specific purpose, not just a method with given parameter and return types. Java SE 8 retains this flavor. When you want to do something with lambda expressions, you still want to keep the purpose of the expression in mind, and have a specific functional interface for it.

A particularly useful interface in the `java.util.function` package is `Predicate`:

```
public interface Predicate<T>
{
   boolean test(T t);
   // Additional default and static methods
}
```

The `ArrayList` class has a `removeIf` method whose parameter is a `Predicate`. It is specifically designed to pass a lambda expression. For example, the following statement removes all null values from an array list:

```
list.removeIf(e -> e == null);
```

6.3.4 Method References

Sometimes, there is already a method that carries out exactly the action that you'd like to pass on to some other code. For example, suppose you simply want to print the event object whenever a timer event occurs. Of course, you could call

```
Timer t = new Timer(1000, event -> System.out.println(event));
```

It would be nicer if you could just pass the `println` method to the `Timer` constructor. Here is how you do that:

```
Timer t = new Timer(1000, System.out::println);
```

The expression `System.out::println` is a *method reference* that is equivalent to the lambda expression `x -> System.out.println(x)`.

As another example, suppose you want to sort strings regardless of letter case. You can pass this method expression:

```
Arrays.sort(strings, String::compareToIgnoreCase)
```

As you can see from these examples, the `::` operator separates the method name from the name of an object or class. There are three principal cases:

- *object*`::`*instanceMethod*
- *Class*`::`*staticMethod*
- *Class*`::`*instanceMethod*

In the first two cases, the method reference is equivalent to a lambda expression that supplies the parameters of the method. As already mentioned, `System.out::println` is equivalent to `x -> System.out.println(x)`. Similarly, `Math::pow` is equivalent to `(x, y) -> Math.pow(x, y)`.

In the third case, the first parameter becomes the target of the method. For example, `String::compareToIgnoreCase` is the same as `(x, y) -> x.compareToIgnoreCase(y)`.

NOTE: When there are multiple overloaded methods with the same name, the compiler will try to find from the context which one you mean. For example, there are two versions of the `Math.max` method, one for integers and one for `double` values. Which one gets picked depends on the method parameters of the functional interface to which `Math::max` is converted. Just like lambda expressions, method references don't live in isolation. They are always turned into instances of functional interfaces.

You can capture the `this` parameter in a method reference. For example, `this::equals` is the same as `x -> this.equals(x)`. It is also valid to use `super`. The method expression

`super::`*instanceMethod*

uses `this` as the target and invokes the superclass version of the given method. Here is an artificial example that shows the mechanics:

```
class Greeter
{
   public void greet()
   {
      System.out.println("Hello, world!");
   }
}

class TimedGreeter extends Greeter
{
   public void greet()
   {
      Timer t = new Timer(1000, super::greet);
      t.start();
   }
}
```

When the `TimedGreeter.greet` method starts, a `Timer` is constructed that executes the `super::greet` method on every timer tick. That method calls the `greet` method of the superclass.

6.3.5 Constructor References

Constructor references are just like method references, except that the name of the method is `new`. For example, `Person::new` is a reference to a `Person` constructor. Which constructor? It depends on the context. Suppose you have a list of strings. Then you can turn it into an array of `Person` objects, by calling the constructor on each of the strings, with the following invocation:

```
ArrayList<String> names = . . .;
Stream<Person> stream = names.stream().map(Person::new);
List<Person> people = stream.collect(Collectors.toList());
```

We will discuss the details of the `stream`, `map`, and `collect` methods in Chapter 1 of Volume II. For now, what's important is that the `map` method calls the `Person(String)` constructor for each list element. If there are multiple `Person` constructors, the compiler picks the one with a `String` parameter because it infers from the context that the constructor is called with a string.

You can form constructor references with array types. For example, `int[]::new` is a constructor reference with one parameter: the length of the array. It is equivalent to the lambda expression `x -> new int[x]`.

Array constructor references are useful to overcome a limitation of Java. It is not possible to construct an array of a generic type `T`. The expression `new T[n]` is an error since it would be erased to `new Object[n]`. That is a problem for library authors. For example, suppose we want to have an array of `Person` objects. The `Stream` interface has a `toArray` method that returns an `Object` array:

```
Object[] people = stream.toArray();
```

But that is unsatisfactory. The user wants an array of references to `Person`, not references to `Object`. The stream library solves that problem with constructor references. Pass `Person[]::new` to the `toArray` method:

```
Person[] people = stream.toArray(Person[]::new);
```

The `toArray` method invokes this constructor to obtain an array of the correct type. Then it fills and returns the array.

6.3.6 Variable Scope

Often, you want to be able to access variables from an enclosing method or class in a lambda expression. Consider this example:

```
public static void repeatMessage(String text, int delay)
{
   ActionListener listener = event ->
      {
         System.out.println(text);
         Toolkit.getDefaultToolkit().beep();
      };
   new Timer(delay, listener).start();
}
```

Consider a call

```
repeatMessage("Hello", 1000); // Prints Hello every 1,000 milliseconds
```

Now look at the variable `text` inside the lambda expression. Note that this variable is *not* defined in the lambda expression. Instead, it is a parameter variable of the `repeatMessage` method.

If you think about it, something nonobvious is going on here. The code of the lambda expression may run long after the call to `repeatMessage` has returned and the parameter variables are gone. How does the `text` variable stay around?

To understand what is happening, we need to refine our understanding of a lambda expression. A lambda expression has three ingredients:

1. A block of code
2. Parameters
3. Values for the *free* variables, that is, the variables that are not parameters and not defined inside the code

In our example, the lambda expression has one free variable, `text`. The data structure representing the lambda expression must store the values for the free variables, in our case, the string "Hello". We say that such values have been *captured* by the lambda expression. (It's an implementation detail how that is done. For example, one can translate a lambda expression into an object with a single method, so that the values of the free variables are copied into instance variables of that object.)

NOTE: The technical term for a block of code together with the values of the free variables is a *closure*. If someone gloats that their language has closures, rest assured that Java has them as well. In Java, lambda expressions are closures.

As you have seen, a lambda expression can capture the value of a variable in the enclosing scope. In Java, to ensure that the captured value is well-defined, there is an important restriction. In a lambda expression, you can only reference variables whose value doesn't change. For example, the following is illegal:

```
public static void countDown(int start, int delay)
{
   ActionListener listener = event ->
      {
         start--; // Error: Can't mutate captured variable
         System.out.println(start);
      };
   new Timer(delay, listener).start();
}
```

There is a reason for this restriction. Mutating variables in a lambda expression is not safe when multiple actions are executed concurrently. This won't happen for the kinds of actions that we have seen so far, but in general, it is a serious problem. See Chapter 14 for more information on this important issue.

It is also illegal to refer to variable in a lambda expression that is mutated outside. For example, the following is illegal:

```
public static void repeat(String text, int count)
{
   for (int i = 1; i <= count; i++)
   {
      ActionListener listener = event ->
         {
            System.out.println(i + ": " + text);
               // Error: Cannot refer to changing i
         };
      new Timer(1000, listener).start();
   }
}
```

The rule is that any captured variable in a lambda expression must be *effectively final*. An effectively final variable is a variable that is never assigned a new value after it has been initialized. In our case, `text` always refers to the same `String` object, and it is OK to capture it. However, the value of `i` is mutated, and therefore `i` cannot be captured.

The body of a lambda expression has *the same scope as a nested block*. The same rules for name conflicts and shadowing apply. It is illegal to declare a parameter or a local variable in the lambda that has the same name as a local variable.

```
Path first = Paths.get("/usr/bin");
Comparator<String> comp =
   (first, second) -> first.length() - second.length();
   // Error: Variable first already defined
```

Inside a method, you can't have two local variables with the same name, and therefore, you can't introduce such variables in a lambda expression either.

When you use the `this` keyword in a lambda expression, you refer to the `this` parameter of the method that creates the lambda. For example, consider

```
public class Application()
{
   public void init()
   {
      ActionListener listener = event ->
         {
            System.out.println(this.toString());
            . . .
         }
      . . .
   }
}
```

The expression `this.toString()` calls the `toString` method of the `Application` object, *not* the `ActionListener` instance. There is nothing special about the use of `this` in a lambda expression. The scope of the lambda expression is nested inside the `init` method, and `this` has the same meaning anywhere in that method.

6.3.7 Processing Lambda Expressions

Up to now, you have seen how to produce lambda expressions and pass them to a method that expects a functional interface. Now let us see how to write methods that can consume lambda expressions.

The point of using lambdas is *deferred execution*. After all, if you wanted to execute some code right now, you'd do that, without wrapping it inside a lambda. There are many reasons for executing code later, such as:

- Running the code in a separate thread
- Running the code multiple times
- Running the code at the right point in an algorithm (for example, the comparison operation in sorting)
- Running the code when something happens (a button was clicked, data has arrived, and so on)
- Running the code only when necessary

Let's look at a simple example. Suppose you want to repeat an action `n` times. The action and the count are passed to a `repeat` method:

```
repeat(10, () -> System.out.println("Hello, World!"));
```

To accept the lambda, we need to pick (or, in rare cases, provide) a functional interface. Table 6.1 lists the most important functional interfaces that are provided in the Java API. In this case, we can use the `Runnable` interface:

```
public static void repeat(int n, Runnable action)
{
   for (int i = 0; i < n; i++) action.run();
}
```

Note that the body of the lambda expression is executed when `action.run()` is called.

Now let's make this example a bit more sophisticated. We want to tell the action in which iteration it occurs. For that, we need to pick a functional interface that has a method with an `int` parameter and a `void` return. The standard interface for processing `int` values is

```
public interface IntConsumer
{
   void accept(int value);
}
```

Here is the improved version of the `repeat` method:

```
public static void repeat(int n, IntConsumer action)
{
   for (int i = 0; i < n; i++) action.accept(i);
}
```

And here is how you call it:

```
repeat(10, i -> System.out.println("Countdown: " + (9 - i)));
```

Table 6.1 Common Functional Interfaces

Functional Interface	Parameter Types	Return Type	Abstract Method Name	Description	Other Methods
`Runnable`	none	`void`	`run`	Runs an action without arguments or return value	
`Supplier<T>`	none	`T`	`get`	Supplies a value of type `T`	
`Consumer<T>`	`T`	`void`	`accept`	Consumes a value of type `T`	`andThen`
`BiConsumer<T, U>`	`T, U`	`void`	`accept`	Consumes values of types `T` and `U`	`andThen`
`Function<T, R>`	`T`	`R`	`apply`	A function with argument of type `T`	`compose`, `andThen`, `identity`
`BiFunction<T, U, R>`	`T, U`	`R`	`apply`	A function with arguments of types `T` and `U`	`andThen`
`UnaryOperator<T>`	`T`	`T`	`apply`	A unary operator on the type `T`	`compose`, `andThen`, `identity`
`BinaryOperator<T>`	`T, T`	`T`	`apply`	A binary operator on the type `T`	`andThen`, `maxBy`, `minBy`
`Predicate<T>`	`T`	`boolean`	`test`	A boolean-valued function	`and`, `or`, `negate`, `isEqual`
`BiPredicate<T, U>`	`T, U`	`boolean`	`test`	A boolean-valued function with two arguments	`and`, `or`, `negate`

Table 6.2 lists the 34 available specializations for primitive types int, long, and double. It is a good idea to use these specializations to reduce autoboxing. For that reason, I used an IntConsumer instead of a Consumer<Integer> in the example of the preceding section.

Table 6.2 Functional Interfaces for Primitive Types
p, *q* is int, long, double; *P*, *Q* is Int, Long, Double

Functional Interface	Parameter Types	Return Type	Abstract Method Name
BooleanSupplier	none	boolean	getAsBoolean
*P*Supplier	none	*p*	getAs*P*
*P*Consumer	*p*	void	accept
Obj*P*Consumer<T>	T, *p*	void	accept
*P*Function<T>	*p*	T	apply
*P*To*Q*Function	*p*	*q*	applyAs*Q*
To*P*Function<T>	T	*p*	applyAs*P*
To*P*BiFunction<T, U>	T, U	*p*	applyAs*P*
*P*UnaryOperator	*p*	*p*	applyAs*P*
*P*BinaryOperator	*p*, *p*	*p*	applyAs*P*
*P*Predicate	*p*	boolean	test

TIP: It is a good idea to use an interface from Tables 6.1 or 6.2 whenever you can. For example, suppose you write a method to process files that match a certain criterion. There is a legacy interface java.io.FileFilter, but it is better to use the standard Predicate<File>. The only reason not to do so would be if you already have many useful methods producing FileFilter instances.

NOTE: Most of the standard functional interfaces have nonabstract methods for producing or combining functions. For example, Predicate.isEqual(a) is the same as a::equals, but it also works if a is null. There are default methods and, or, negate for combining predicates. For example, Predicate.isEqual(a).or(Predicate.isEqual(b)) is the same as x -> a.equals(x) || b.equals(x).

NOTE: If you design your own interface with a single abstract method, you can tag it with the `@FunctionalInterface` annotation. This has two advantages. The compiler gives an error message if you accidentally add another nonabstract method. And the javadoc page includes a statement that your interface is a functional interface.

It is not required to use the annotation. Any interface with a single abstract method is, by definition, a functional interface. But using the `@FunctionalInterface` annotation is a good idea.

6.3.8 More about Comparators

The `Comparator` interface has a number of convenient static methods for creating comparators. These methods are intended to be used with lambda expressions or method references.

The static `comparing` method takes a "key extractor" function that maps a type `T` to a comparable type (such as `String`). The function is applied to the objects to be compared, and the comparison is then made on the returned keys. For example, suppose you have an array of `Person` objects. Here is how you can sort them by name:

```
Arrays.sort(people, Comparator.comparing(Person::getName));
```

This is certainly much easier than implementing a `Comparator` by hand. Moreover, the code is clearer since it is obvious that we want to compare people by name.

You can chain comparators with the `thenComparing` method for breaking ties. For example,

```
Arrays.sort(people,
   Comparator.comparing(Person::getLastName)
   .thenComparing(Person::getFirstName));
```

If two people have the same last name, then the second comparator is used.

There are a few variations of these methods. You can specify a comparator to be used for the keys that the `comparing` and `thenComparing` methods extract. For example, here we sort people by the length of their names:

```
Arrays.sort(people, Comparator.comparing(Person::getName,
   (s, t) -> Integer.compare(s.length(), t.length())));
```

Moreover, both the `comparing` and `thenComparing` methods have variants that avoid boxing of `int`, `long`, or `double` values. An easier way of producing the preceding operation would be

```
Arrays.sort(people, Comparator.comparingInt(p -> p.getName().length()));
```

If your key function can return `null`, you will like the `nullsFirst` and `nullsLast` adapters. These static methods take an existing comparator and modify it so that it doesn't throw an exception when encountering `null` values but ranks them as smaller or larger than regular values. For example, suppose `getMiddleName` returns a `null` when a person has no middle name. Then you can use `Comparator.comparing(Person::getMiddleName(), Comparator.nullsFirst(...))`.

The `nullsFirst` method needs a comparator—in this case, one that compares two strings. The `naturalOrder` method makes a comparator for any class implementing `Comparable`. A `Comparator.<String>naturalOrder()` is what we need. Here is the complete call for sorting by potentially null middle names. I use a static import of `java.util.Comparator.*`, to make the expression more legible. Note that the type for `naturalOrder` is inferred.

```
Arrays.sort(people, comparing(Person::getMiddleName, nullsFirst(naturalOrder())));
```

The static `reverseOrder` method gives the reverse of the natural order. To reverse any comparator, use the `reversed` instance method. For example, `naturalOrder().reversed()` is the same as `reverseOrder()`.

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 331, you will see a simple inner class that accesses an instance field of its outer class.
2. On page 334, we cover the special syntax rules for inner classes.
3. Starting on page 335, 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 339, we discuss *local inner classes* that can access local variables of the enclosing scope.
5. Starting on page 342, we introduce *anonymous inner classes* and show how they were commonly used to implement callbacks before Java had lambda expressions.

6. Finally, starting on page 346, 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) but 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)
   {
      System.out.println("At the tone, the time is " + new Date());
      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).

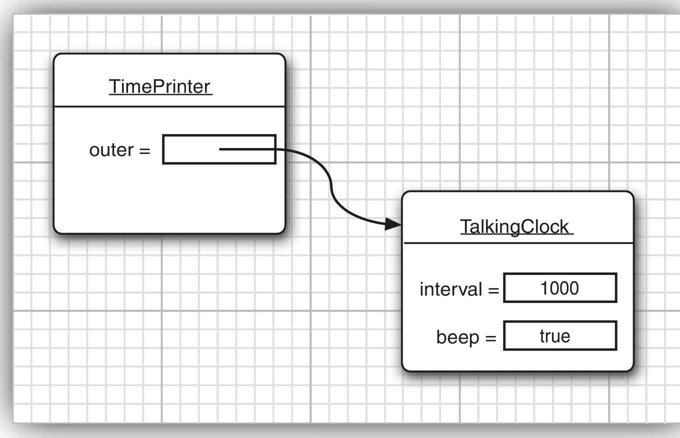

Figure 6.3 An inner class object has a reference to an outer class object

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)
{
   System.out.println("At the tone, the time is " + new Date());
   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 that *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 6.7 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 6.7 `innerClass/InnerClassTest.java`

```java
package innerClass;

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

/**
 * This program demonstrates the use of inner classes.
 * @version 1.11 2015-05-12
 * @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.
 */
```

(Continues)

Listing 6.7 *(Continued)*

```
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)
      {
         System.out.println("At the tone, the time is " + new Date());
         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.

NOTE: Any static fields declared in an inner class must be `final`. There is a simple reason. One expects a unique instance of a static field, but there is a separate instance of the inner class for each outer object. If the field was not `final`, it might not be unique.

An inner class cannot have `static` methods. The Java Language Specification gives no reason for this limitation. It would have been possible to allow static methods that only access static fields and methods from the enclosing class. Apparently, the language designers decided that the complexities outweighed the benefits.

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 when 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 (TalkingClock.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 `TalkingClock$1` class is synthesized solely to distinguish this constructor from others.

The compiler 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)
      {
         System.out.println("At the tone, the time is " + new Date());
         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 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 *effectively final*. That means, they may never change once they have been assigned.

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, boolean beep)
{
   class TimePrinter implements ActionListener
   {
      public void actionPerformed(ActionEvent event)
      {
```

```
            System.out.println("At the tone, the time is " + new Date());
                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, 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 made inside the local class will always have the same value.

NOTE: Before Java SE 8, it was necessary to declare any local variables that are accessed from local classes as `final`. For example, this is how the `start` method would have been declared so that the inner class can access the `beep` parameter:

```
public void start(int interval, final boolean beep)
```

The "effectively `final`" restriction is sometimes 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. A remedy is to use an array of length 1:

```
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);
         }
      };
```

When inner classes were first invented, a prototype version of the compiler automatically made this transformation for all local variables that were modified in the inner class. However, this was later abandoned. After all, there is a danger. When the code in the inner class is executed at the same time in multiple threads, the concurrent updates can lead to race conditions—see Chapter 14.

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, boolean beep)
{
   ActionListener listener = new ActionListener()
   {
      public void actionPerformed(ActionEvent event)
      {
         System.out.println("At the tone, the time is " + new Date());
         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.

Listing 6.8 contains the complete source code for the talking clock program with an anonymous inner class. If you compare this program with Listing 6.7, you will see that in this case, the solution with the anonymous inner class is quite a bit shorter and, hopefully, with some practice, as easy to comprehend.

For many years, Java programmers routinely used anonymous inner classes for event listeners and other callbacks. Nowadays, you are better off using a lambda expression. For example, the `start` method from the beginning of this section can be written much more concisely with a lambda expression like this:

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

Listing 6.8 `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 1.11 2015-05-12
 * @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, boolean beep)
   {
      ActionListener listener = new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               System.out.println("At the tone, the time is " + new Date());
               if (beep) Toolkit.getDefaultToolkit().beep();
            }
         };
      Timer t = new Timer(interval, listener);
      t.start();
   }
}
```

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.POSITIVE_INFINITY;
double max = Double.NEGATIVE_INFINITY;
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: Unlike regular inner classes, static inner classes can have static fields and methods.

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

Listing 6.9 contains the complete source code of the `ArrayAlg` class and the nested `Pair` class.

Listing 6.9 `staticInnerClass/StaticInnerClassTest.java`

```
package staticInnerClass;

/**
 * This program demonstrates the use of static inner classes.
 * @version 1.02 2015-05-12
 * @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;

```

```java
      /**
       * 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
    */
   public static Pair minmax(double[] values)
   {
      double min = Double.POSITIVE_INFINITY;
      double max = Double.NEGATIVE_INFINITY;
      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.

6.5.1 When to Use Proxies

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.

6.5.2 Creating Proxy Objects

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 6.10, 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 6.10 proxy/ProxyTest.java

```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
 */
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;
```

(Continues)

Listing 6.10 *(Continued)*

```

      // 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);
   }
}
```

6.5.3 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 6.10, 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<?> cl)`

  returns `true` if `cl` is a proxy class.

This ends our final chapter on the fundamentals of the Java programming language. Interfaces, lambda expressions, and inner classes are concepts that you will encounter frequently. However, as we already mentioned, cloning and proxies are advanced techniques that are of interest mainly to library designers and tool builders, not application programmers. You are now ready to learn how to deal with exceptional situations in your programs in Chapter 7.

This page intentionally left blank

Index

C

D

H

I

Q

R