

# core JAVA™

## VOLUME I-FUNDAMENTALS

---

EIGHTH EDITION



CAY S. HORSTMANN

GARY CORNELL



PRENTICE  
HALL

Sun Microsystems Press

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

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

Sun Microsystems, Inc., has intellectual property rights relating to implementations of the technology described in this publication. In particular, and without limitation, these intellectual property rights may include one or more *US* patents, foreign patents, or pending applications. Sun, Sun Microsystems, the Sun logo, J2ME, Solaris, Java, Javadoc, NetBeans, and all Sun and Java based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC., MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCTS) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANYTIME

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact U.S. Corporate and Government Sales, (800) 382-3419, [corpsales@pearsoned.com](mailto:corpsales@pearsoned.com). For sales outside the United States please contact: International Sales, [tatemational@pearsoned.com](mailto:tatemational@pearsoned.com).

Visit us on the Web: [www.prenhallprofessional.com](http://www.prenhallprofessional.com)

*Library of Congress Cataloging-in-Publication Data*

Horstmann, Cay S., 1959-

Core Java. Volume I, Fundamentals / Cay S. Horstmann, Gary Cornell. —  
8th ed.

p. cm.

Includes index.

ISBN 978-0-13-235476-9 (pbk.: alk. paper) I. Java (Computer program language) I. Cornell, Gary. II. Title. III. Title: Fundamentals. IV.

Title: Core-Java fundamentals.

QA76.73.I3SH6753 2008  
005.133-dc22

2007028843

Copyright© 2008 Sun Microsystems, Inc.

4150 Network Circle, Santa Clara, California 95054 U.S.A.

All rights reserved.. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likevi.se. For information regarding permissions, write to: Pearson Education, Inc., Rights and Contracts Department, One Like Street, Upper Saddle River, NJ 07458.

ISBN-13: 978-0-13-235476-9

ISBN-10: 0-13-235476-4

Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.

First printing, September 2007

# Table of Contents

Preface xix

Acknowledgments xxv

Chapter 1: An Introduction to Java 1

**Java As a Programming Platform 2**

**The Java “White Paper” Buzzwords 2**

**Java Applets and the Internet 7**

**A Short History of Java 9**

**Common Misconceptions about Java 11**

Chapter 2: The Java Programming Environment 15

**Installing the Java Development Kit 16**

**Choosing a Development Environment 21**

**Using the Command-Line Tools 22**

**Using an Integrated Development Environment 25**

**Running a Graphical Application 28**

**Building and Running Applets 31**

Chapter 3: Fundamental Programming Structures in Java 35

**A Simple Java Program 36**

**Comments 39**

**Data Types 40**

**Variables 44**

**Operators 46**

**Strings 53**

**Input and Output 63**

**Control Flow 71**

**Big Numbers 88**

**Arrays 90**

Chapter 4: Objects and Classes 105

**Introduction to Object-Oriented Programming 106**

**Using Predefined Classes 111**

**Defining Your Own Classes 122**

**Static Fields and Methods 132**

**Method Parameters 138**

**Object Construction 144**

**Packages 15**

**The Class Path 160**

**Documentation Comments 162**

**Class Design Hints 167**

Chapter 5: Inheritance 171

**Classes, Superclasses, and Subclasses 172**

**Object: The Cosmic Superclass 192**

**Generic Array Lists 204**

**Object Wrappers and Autoboxing 211**  
**Methods with a Variable Number of Parameters 214**  
**Enumeration Classes 215**  
**Reflection 217**  
**Design Hints for Inheritance 238**

Chapter 6: Interfaces and Inner Classes 241

**Interfaces 242**  
**Object Cloning 249**  
**Interfaces and Callbacks 255**  
**Inner Classes 258**  
**Proxies 275**

Chapter 7: Graphics Programming 281

**Introducing Swing 282**  
**Creating a Frame 285**  
**Positioning a Frame 288**  
**Displaying Information in a Component 294**  
**Working with 2D Shapes 299**  
**Using Color 307**  
**Using Special Fonts for Text 310**  
**Displaying Images 318**

Chapter 8: Event Handling 323

**Basics of Event Handling 324**  
**Actions 342**  
**Mouse Events 349**  
**The AWT Event Hierarchy 357**

Chapter 9: User Interface Components with Swing 361

**Swing and the Model-View-Controller Design Pattern 362**  
**Introduction to Layout Management 368**  
**Text Input 377**  
**Choice Components 385**  
**Menus 406**  
**Sophisticated Layout Management 424**  
**Dialog Boxes 452**

Chapter 10: Deploying Applications and Applets 493

**JAR Files 494**  
**Java Web Start 501**  
**Applets 516**  
**Storage of Application Preferences 539**

Chapter 11: Exceptions, Logging, Assertions, and Debugging 551

**Dealing with Errors 552**  
**Catching Exceptions 559**  
**Tips for Using Exceptions 568**  
**Using Assertions 571**  
**Logging 575**  
**Debugging Tips 591**  
**Using a Debugger 607**



Chapter 12: Generic Programming 613

**Why Generic Programming? 614**

**Definition of a Simple Generic Class 616**

**Generic Methods 618**

**Bounds for Type Variables 619**

**Generic Code and the Virtual Machine 621**

**Restrictions and Limitations 626**

**Inheritance Rules for Generic Types 630**

**Wildcard Types 632**

**Reflection and Generics 640**

Chapter 13: Collections 649

**Collection Interfaces 650**

**Concrete Collections 658**

**The Collections Framework 689**

**Algorithms 700**

**Legacy Collections 707**

Chapter 14: Multithreading 715

**What Are Threads? 716**

**Interrupting Threads 728**

**Thread States 730**

**Thread Properties 733**

**Synchronization 736**

**Blocking Queues 764**

**Thread-Safe Collections 771**

**Callables and Futures 774**

**Executors 778**

**Synchronizers 785**

**Threads and Swing 794**

Appendix 809

# 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, whether that information comes from web servers, databases, information providers, or any other imaginable source. Indeed, Java is in a unique position to fulfill this promise. It is an extremely solidly engineered language that has gained acceptance by all major vendors, except for Microsoft. Its built-in security and safety features are reassuring both to programmers and to the users of Java programs. Java even has built-in support that makes advanced programming tasks, such as network programming, database connectivity, and multithreading, straightforward.

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

The book you have in your hands is the first volume of the eighth edition of *Core Java*<sup>™</sup>. With the publishing of each edition, the book followed the release of the Java Development Kit as quickly as possible, 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) 6.

As with the previous editions of this book, *we still target serious programmers who want to put jam 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 this *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 that demonstrates almost every language and library feature that we discuss. We keep the sample programs purposefully simple to focus on the major points, but, for the most part, they aren't fake and they don't cut corners. They should make good starting points for your own code.

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

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

With the explosive growth of the Java class library, a one-volume treatment of all the features of Java that serious programmers need to know is no longer possible. Hence, we decided to break up the book into two volumes. The first volume, which you hold in your hands, concentrates on the fundamental concepts of the Java language, along with the basics of user-interface programming. The second volume, *Core Java, Volume II—Advanced Features* (forthcoming, ISBN: 978-0-13-235479-0), goes further into the enterprise features and advanced user-interface programming. It includes detailed discussions of:

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

In this edition, we reshuffled the contents of the two volumes. In particular, multi-threading is now covered in Volume I because it has become so important, with Moore's law coming to an end.

When writing a book, errors and inaccuracies are inevitable. We'd very much like to know about them. But, of course, we'd prefer to learn about each of them only once. We have put up a list of frequently asked questions, bugs fixes, and workarounds in 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 if we 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 completely object oriented. **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++, then you can browse through this chapter quickly. Programmers coming from a non-object-oriented background should expect to spend some time mastering OOP concepts before going further with Java.

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

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

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

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

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

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

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

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

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

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

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

## Conventions

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



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

---



TIP: Tips are tagged with the "tip" icon 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 difference 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.

### **API** Application Programming Interface

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 tagged with an API icon 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 on-line API documentation. We now tag each API note with the version number in which the feature was introduced, to help the readers who don't use the "bleeding edge" version of Java.

Programs whose source code is on the Web are listed as examples, for instance

#### **Listing 1-1** `WelcomeApplet.java`

### Sample Code

The web site for this book at <http://horstmann.com/core3java> contains all sample code from the book, in compressed form. You can expand the file either with one of the familiar unzipping programs or simply with the `jar` utility that is part of the Java Development Kit. See Chapter 2 for more information about installing the Java Development Kit and the sample code.

# Acknowledgments

Writing a book is always a monumental effort, and rewriting doesn't seem to be much easier, especially with 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 and Sun Microsystems Press provided valuable assistance, but they 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 of Prentice Hall, 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 grateful to Vanessa Moore for the excellent production support. My thanks also to my coauthor of earlier editions, Gary Cornell, who has since moved on to other ventures.

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

Reviewers of this and earlier editions include Chuck Allison (Contributing Editor, *C/C++ Users journal*), Alec Beaton (PointBase, Inc.), Cliff Berg (iSavvix Corporation), Joshua Bloch (Sun Microsystems), David Brown, Corky Cartwright, Frank Cohen (PushToTest), Chris Crane (devXsolution), Dr. Nicholas J. De Lillo (Manhattan College), Rakesh Dhoopar (Oracle), David Geary (Sabrewaro), Brian Goetz (Principal Consultant, Quiotix Corp.), Angela Gordon (Sun Microsystems), Dan Gordon (Sun Microsystems), Rob Gordon, John Gray (University of Hartford), Cameron Gregory (olabs.com), Marty Hall (The Johns Hopkins University Applied Physics Lab), Vincent Hardy (Sun Microsystems), Dan Harkey (San Jose State University), William Higgins (IBM), Vladimir Ivanovic (PointBase), Jerry Jackson (ChannelPoint Software), Tim Kimmel (Preview Systems), Chris Laffra, Charlie Lai (Sun

Microsystems), Angelika Langer, Doug Langston, Hang Liu (McGill University), Mark Lawrence, Doug Lea (SUNY Oswego), Gregory Longshore, Bob Lynch (Lynch Associates), Philip Milne (consultant), Mark Morrisoy (The Oregon Graduate Institute), Mahesh Neelakanta (Florida Atlantic University), Hao Pham, Paul Phillion, Blake Ragsdell, Stuart Reges (University of Arizona), Rich Rosen (Interactive Data Corporation), Peter Sanders (ESSI University, Nice, France), Dr. Paul Sanghera (San Jose State University and Brooks College), Paul Sevinc (Teamup AG), Devang Shah (Sun Microsystems), Bradley A. Smith, Steven Stelting (Sun Microsystems), Christopher Taylor, Luke Taylor (Valtech), George Thiruvathukal, Kim Topley (author of *Core JFC*), Janet Traub, Paul Tyma (consultant), Peter van der Linden (Sun Microsystems), and Burt Walsh.

*Cay Horstmann*  
*San Francisco, 2007*



---

*Chapter*

1

AN INTRODUCTION  
TO JAVA

- ▼ JAVA AS A PROGRAMMING PLATFORM
- ▼ THE JAVA "WHITE PAPER" BUZZWORDS
- ▼ JAVA APPLETS AND THE INTERNET
- ▼ A SHORT HISTORY OF JAVA
- ▼ COMMON MISCONCEPTIONS ABOUT JAVA

The first release of Java in 1996 generated an incredible amount of excitement, not just in the computer press, but in mainstream media such as *The New York Times*, *The Washington Post*, and *Business Week*. Java has the distinction of being the first and only programming language that had a ten-minute story on National Public Radio. A \$100,000,000 venture capital fund was set up solely for products produced by use of a *specific* computer language. It is rather amusing to revisit those heady times, and we give you a brief history of Java in this chapter.

### Java As a Programming Platform

In the first edition of this book, we had this to write about Java:

“As a computer language, Java’s hype is overdone: Java is certainly a *good* programming language. There is no doubt that it is one of the better languages available to serious programmers. We think it could *potentially* have been a great programming language, but it is probably too late for that. Once a language is out in the field, the ugly reality of compatibility with existing code sets in.”

Our editor got a lot of flack for this paragraph from someone very high up at Sun Microsystems who shall remain unnamed. But, in hindsight, our prognosis seems accurate. Java has a lot of nice language features—we examine them in detail later in this chapter. It has its share of warts, and newer additions to the language are not as elegant as the original ones because of the ugly reality of compatibility.

But, as we already said in the first edition, Java was never just a language. There are lots of programming languages out there, and few of them make much of a splash. Java is a whole *platform*, with a huge library, containing lots of reusable code, and an execution environment that provides services such as security, portability across operating systems, and automatic garbage collection.

As a programmer, you will want a language with a pleasant syntax and comprehensible semantics (i.e., not C++). Java fits the bill, as do dozens of other fine languages. Some languages give you portability, garbage collection, and the like, but they don’t have much of a library, forcing you to roll your own if you want fancy graphics or networking or database access. Well, Java has everything—a good language, a high-quality execution environment, and a vast library. That combination is what makes Java an irresistible proposition to so many programmers.

### The Java “White Paper” Buzzwords

The authors of Java have written an influential White Paper that explains their design goals and accomplishments. They also published a shorter summary that is organized along the following 11 buzzwords:

Simple	Portable
Object Oriented	Interpreted
Network-Savvy	High Performance
Robust	Multithreaded
Secure	Dynamic
Architecture Neutral	

In this section, we will

- Summarize, with excerpts from the White Paper, what the Java designers say about each buzzword; and
- Tell you what we think of each buzzword, based on our experiences with the current version of Java.



NOTE: As we write this, the White Paper can be found at <http://java.sun.com/docs/white/langenv/>. The summary with the 11 buzzwords is at <http://java.sun.com/docs/overviews/java/java-overview-1.html>.

### Simple

*We wanted to build a system that could be programmed easily without a lot of esoteric training and which leveraged today's standard practice. So even though we found that C++ was unsuitable, we designed Java as closely to C++ as possible in order to make the system more comprehensible. Java omits many rarely used, poorly understood, confusing features of C++ that, in our experience, bring more grief than benefit.*

The syntax for Java is, indeed, a cleaned-up version of the syntax for C++. There is no need for header files, pointer arithmetic (or even a pointer syntax), structures, unions, operator overloading, virtual base classes, and so on. (See the C++ notes interspersed throughout the text for more on the differences between Java and C++.) The designers did not, however, attempt to fix all of the clumsy features of C++. For example, the syntax of the `switch` statement is unchanged in Java. If you know C++, you will find the transition to the Java syntax easy.

If you are used to a visual programming environment (such as Visual Basic), you will not find Java simple. There is much strange syntax (though it does not take long to get the hang of it). More important, you must do a lot more programming in Java. The beauty of Visual Basic is that its visual design environment almost automatically provides a lot of the infrastructure for an application. The equivalent functionality must be programmed manually, usually with a fair bit of code, in Java. There are, however, third-party development environments that provide “drag-and-drop”-style program development.

*Another aspect of being simple is being small. One of the goals of Java is to enable the construction of software that can run stand-alone in small machines. The size of the basic interpreter and class support is about 40K bytes; adding the basic standard libraries and thread support (essentially a self-contained microkernel) adds an additional 175K.*

This was a great achievement at the time. Of course, the library has since grown to huge proportions. There is now a separate Java Micro Edition with a smaller library, suitable for embedded devices.

### Object Oriented

*Simply stated, object-oriented design is a technique for programming that focuses on the data (= objects) and on the interfaces to that object. To make an analogy with carpentry, an “object-oriented” carpenter would be mostly concerned with the chair*

*he was building, and secondarily with the tools used to make it; a “non-object-oriented” carpenter would think primarily of his tools. The object-oriented facilities of Java are essentially those of C++.*

Object orientation has proven its worth in the last 30 years, and it is inconceivable that a modern programming language would not use it. Indeed, the object-oriented features of Java are comparable to those of C++. The major difference between Java and C++ lies in multiple inheritance, which Java has replaced with the simpler concept of interfaces, and in the Java metaclass model (which we discuss in Chapter 5).



NOTE: If you have no experience with object-oriented programming languages, you will want to carefully read Chapters 4 through 6. These chapters explain what object-oriented programming is and why it is more useful for programming sophisticated projects than are traditional, procedure-oriented languages like C or Basic.

### **Network-Savvy**

*Java has an extensive library of routines for coping with TCP/IP protocols like HTTP and FTP. Java applications can open and access objects across the Net via URLs with the same ease as when accessing a local file system.*

We have found the networking capabilities of Java to be both strong and easy to use. Anyone who has tried to do Internet programming using another language will revel in how simple Java makes onerous tasks like opening a socket connection. (We cover networking in Volume II of this book.) The remote method invocation mechanism enables communication between distributed objects (also covered in Volume II).

### **Robust**

*Java is intended for writing programs that must be reliable in a variety of ways. Java puts a lot of emphasis on early checking for possible problems, later dynamic (runtime) checking, and eliminating situations that are error-prone. . . . The single biggest difference between Java and C/C++ is that Java has a pointer model that eliminates the possibility of overwriting memory and corrupting data.*

This feature is also very useful. The Java compiler detects many problems that, in other languages, would show up only at runtime. As for the second point, anyone who has spent hours chasing memory corruption caused by a pointer bug will be very happy with this feature of Java.

If you are coming from a language like Visual Basic that doesn't explicitly use pointers, you are probably wondering why this is so important. C programmers are not so lucky. They need pointers to access strings, arrays, objects, and even files. In Visual Basic, you do not use pointers for any of these entities, nor do you need to worry about memory allocation for them. On the other hand, many data structures are difficult to implement in a pointerless language. Java gives you the best of both worlds. You do not need pointers for everyday constructs like strings and arrays. You have the power of pointers if you need it, for example, for linked lists. And you always have complete safety, because you can never access a bad pointer, make memory allocation errors, or have to protect against memory leaking away.

**Secure**

*Java is intended to be used in networked/distributed environments. Toward that end, a lot of emphasis has been placed on security. Java enables the construction of virus-free, tamper-free systems.*

In the first edition of *Core Java* we said: “Well, one should ‘never say never again,’” and we turned out to be right. Not long after the first version of the Java Development Kit was shipped, a group of security experts at Princeton University found subtle bugs in the security features of Java 1.0. Sun Microsystems has encouraged research into Java security, making publicly available the specification and implementation of the virtual machine and the security libraries. They have fixed all known security bugs quickly. In any case, Java makes it extremely difficult to outwit its security mechanisms. The bugs found so far have been very technical and few in number.

From the beginning, Java was designed to make certain kinds of attacks impossible, among them:

- Overrunning the runtime stack—a common attack of worms and viruses
- Corrupting memory outside its own process space
- Reading or writing files without permission

A number of security features have been added to Java over time. Since version 1.1, Java has the notion of digitally signed classes (see Volume II). With a signed class, you can be sure who wrote it. Any time you trust the author of the class, the class can be allowed more privileges on your machine.



NOTE: A competing code delivery mechanism from Microsoft based on its ActiveX technology relies on digital signatures alone for security. Clearly this is not sufficient—as any user of Microsoft’s own products can confirm, programs from well-known vendors do crash and create damage. Java has a far stronger security model than that of ActiveX because it controls the application as it runs and stops it from wreaking havoc.

**Architecture Neutral**

*The compiler generates an architecture-neutral object file format—the compiled code is executable on many processors, given the presence of the Java runtime system. The Java compiler does this by generating bytecode instructions which have nothing to do with a particular computer architecture. Rather, they are designed to be both easy to interpret on any machine and easily translated into native machine code on the fly.*

This is not a new idea. More than 30 years ago, both Niklaus Wirth’s original implementation of Pascal and the UCSD Pascal system used the same technique.

Of course, interpreting bytecodes is necessarily slower than running machine instructions at full speed, so it isn’t clear that this is even a good idea. However, virtual machines have the option of translating the most frequently executed bytecode sequences into machine code, a process called just-in-time compilation. This strategy has proven so effective that even Microsoft’s .NET platform relies on a virtual machine.

The virtual machine has other advantages. It increases security because the virtual machine can check the behavior of instruction sequences. Some programs even produce bytecodes on the fly, dynamically enhancing the capabilities of a running program.

**Portable**

*Unlike C and C++, there are no “implementation-dependent” aspects of the specification. The sizes of the primitive data types are specified, as is the behavior of arithmetic on them.*

For example, an `int` in Java is always a 32-bit integer. In C/C++, `int` can mean a 16-bit integer, a 32-bit integer, or any other size that the compiler vendor likes. The only restriction is that the `int` type must have at least as many bytes as a `short int` and cannot have more bytes than a `long int`. Having a fixed size for number types eliminates a major porting headache. Binary data is stored and transmitted in a fixed format, eliminating confusion about byte ordering. Strings are saved in a standard Unicode format.

*The libraries that are a part of the system define portable interfaces. For example, there is an abstract `Window` class and implementations of it for UNIX, Windows, and the Macintosh.*

As anyone who has ever tried knows, it is an effort of heroic proportions to write a program that looks good on Windows, the Macintosh, and ten flavors of UNIX. Java 1.0 made the heroic effort, delivering a simple toolkit that mapped common user interface elements to a number of platforms. Unfortunately, the result was a library that, with a lot of work, could give barely acceptable results on different systems. (And there were often *different* bugs on the different platform graphics implementations.) But it was a start. There are many applications in which portability is more important than user interface slickness, and these applications did benefit from early versions of Java. By now, the user interface toolkit has been completely rewritten so that it no longer relies on the host user interface. The result is far more consistent and, we think, more attractive than in earlier versions of Java.

**Interpreted**

*The Java interpreter can execute Java bytecodes directly on any machine to which the interpreter has been ported. Since linking is a more incremental and lightweight process, the development process can be much more rapid and exploratory.*

Incremental linking has advantages, but its benefit for the development process is clearly overstated. Early Java development tools were, in fact, quite slow. Today, the bytecodes are translated into machine code by the just-in-time compiler.

**High Performance**

*While the performance of interpreted bytecodes is usually more than adequate, there are situations where higher performance is required. The bytecodes can be translated on the fly (at runtime) into machine code for the particular CPU the application is running on.*

In the early years of Java, many users disagreed with the statement that the performance was “more than adequate.” Today, however, the just-in-time compilers have become so good that they are competitive with traditional compilers and, in some cases, even outperform them because they have more information available. For example, a just-in-time compiler can monitor which code is executed frequently and optimize just

that code for speed. A more sophisticated optimization is the elimination (or “inlining”) of function calls. The just-in-time compiler knows which classes have been loaded. It can use inlining when, based upon the currently loaded collection of classes, a particular function is never overridden, and it can undo that optimization later if necessary.

### **Multithreaded**

*[The] benefits of multithreading are better interactive responsiveness and real-time behavior.*

If you have ever tried to do multithreading in another language, you will be pleasantly surprised at how easy it is in Java. Threads in Java also can take advantage of multi-processor systems if the base operating system does so. On the downside, thread implementations on the major platforms differ widely, and Java makes no effort to be platform independent in this regard. Only the code for calling multithreading remains the same across machines; Java offloads the implementation of multithreading to the underlying operating system or a thread library. Nonetheless, the ease of multithreading is one of the main reasons why Java is such an appealing language for server-side development.

### **Dynamic**

*In a number of ways, Java is a more dynamic language than C or C++. It was designed to adapt to an evolving environment. Libraries can freely add new methods and instance variables without any effect on their clients. In Java, finding out runtime type information is straightforward.*

This is an important feature in those situations in which code needs to be added to a running program. A prime example is code that is downloaded from the Internet to run in a browser. In Java 1.0, finding out runtime type information was anything but straightforward, but current versions of Java give the programmer full insight into both the structure and behavior of its objects. This is extremely useful for systems that need to analyze objects at runtime, such as Java GUI builders, smart debuggers, pluggable components, and object databases.



NOTE: Shortly after the initial success of Java, Microsoft released a product called J++ with a programming language and virtual machine that was almost identical to Java. At this point, Microsoft is no longer supporting J++ and has instead introduced another language called C# that also has many similarities with Java but runs on a different virtual machine. There is even a J# for migrating J++ applications to the virtual machine used by C#. We do not cover J++, C#, or J# in this book.

## **Java Applets and the Internet**

The idea here is simple: Users will download Java bytecodes from the Internet and run them on their own machines. Java programs that work on web pages are called *applets*. To use an applet, you only need a Java-enabled web browser, which will execute the bytecodes for you. You need not install any software. Because Sun licenses the Java source code and insists that there be no changes in the language and standard library, a Java applet should run on any browser that is advertised as Java-enabled. You get the latest version of the program whenever you visit the web page containing the applet.

Most important, thanks to the security of the virtual machine, you need never worry about attacks from hostile code.

When the user downloads an applet, it works much like embedding an image in a web page. The applet becomes a part of the page, and the text flows around the space used for the applet. The point is, the image is *alive*. It reacts to user commands, changes its appearance, and sends data between the computer presenting the applet and the computer serving it.

Figure 1-1 shows a good example of a dynamic web page that carries out sophisticated calculations. The Jmol applet displays molecular structures. By using the mouse, you can rotate and zoom each molecule to better understand its structure. This kind of direct manipulation is not achievable with static web pages, but applets make it possible. (You can find this applet at <http://jmol.sourceforge.net>.)

When applets first appeared, they created a huge amount of excitement. Many people believe that the lure of applets was responsible for the astonishing popularity of Java. However, the initial excitement soon turned into frustration. Various versions of Netscape and Internet Explorer ran different versions of Java, some of which were seriously outdated. This sorry situation made it increasingly difficult to develop applets that took advantage of the most current Java version. Today, most web pages simply use JavaScript or Flash when dynamic effects are desired in the browser. Java, on the other hand, has become the most popular language for developing the server-side applications that produce web pages and carry out the backend logic.

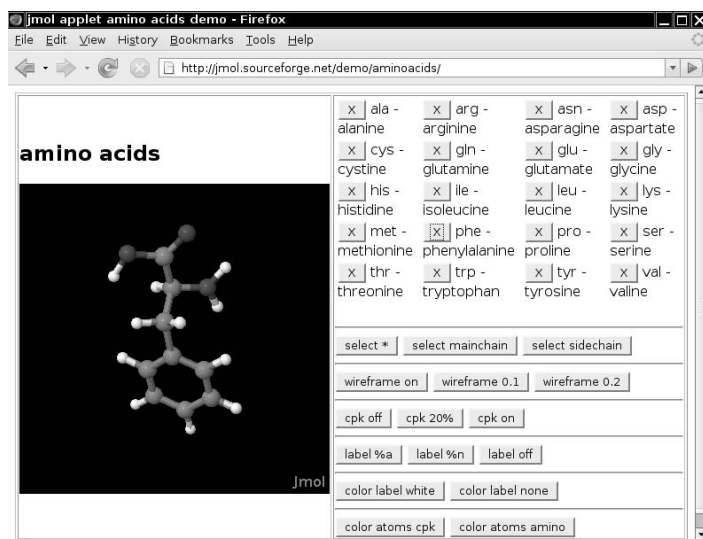


Figure 1-1 The Jmol applet



## A Short History of Java

This section gives a short history of Java's evolution. It is based on various published sources (most important, on an interview with Java's creators in the July 1995 issue of *SunWorld's* on-line magazine).

Java goes back to 1991, when a group of Sun engineers, led by Patrick Naughton and Sun Fellow (and all-around computer wizard) James Gosling, wanted to design a small computer language that could be used for consumer devices like cable TV switchboxes. Because these devices do not have a lot of power or memory, the language had to be small and generate very tight code. Also, because different manufacturers may choose different central processing units (CPUs), it was important that the language not be tied to any single architecture. The project was code-named "Green."

The requirements for small, tight, and platform-neutral code led the team to resurrect the model that some Pascal implementations tried in the early days of PCs. Niklaus Wirth, the inventor of Pascal, had pioneered the design of a portable language that generated intermediate code for a hypothetical machine. (These are often called *virtual machines*—hence, the Java virtual machine or JVM.) This intermediate code could then be used on any machine that had the correct interpreter. The Green project engineers used a virtual machine as well, so this solved their main problem.

The Sun people, however, come from a UNIX background, so they based their language on C++ rather than Pascal. In particular, they made the language object oriented rather than procedure oriented. But, as Gosling says in the interview, "All along, the language was a tool, not the end." Gosling decided to call his language "Oak" (presumably because he liked the look of an oak tree that was right outside his window at Sun). The people at Sun later realized that Oak was the name of an existing computer language, so they changed the name to Java. This turned out to be an inspired choice.

In 1992, the Green project delivered its first product, called "7." It was an extremely intelligent remote control. (It had the power of a SPARCstation in a box that was 6 inches by 4 inches by 4 inches.) Unfortunately, no one was interested in producing this at Sun, and the Green people had to find other ways to market their technology. However, none of the standard consumer electronics companies were interested. The group then bid on a project to design a cable TV box that could deal with new cable services such as video on demand. They did not get the contract. (Amusingly, the company that did was led by the same Jim Clark who started Netscape—a company that did much to make Java successful.)

The Green project (with a new name of "First Person, Inc.") spent all of 1993 and half of 1994 looking for people to buy its technology—no one was found. (Patrick Naughton, one of the founders of the group and the person who ended up doing most of the marketing, claims to have accumulated 300,000 air miles in trying to sell the technology.) First Person was dissolved in 1994.

While all of this was going on at Sun, the World Wide Web part of the Internet was growing bigger and bigger. The key to the Web is the browser that translates the hypertext page to the screen. In 1994, most people were using Mosaic, a noncommercial web browser that came out of the supercomputing center at the University of Illinois in 1993. (Mosaic was partially written by Marc Andreessen for \$6.85 an hour as

an undergraduate student on a work-study project. He moved on to fame and fortune as one of the cofounders and the chief of technology at Netscape.)

In the *SunWorld* interview, Gosling says that in mid-1994, the language developers realized that “We could build a real cool browser. It was one of the few things in the client/server mainstream that needed some of the weird things we’d done: architecture neutral, real-time, reliable, secure—issues that weren’t terribly important in the workstation world. So we built a browser.”

The actual browser was built by Patrick Naughton and Jonathan Payne and evolved into the HotJava browser. The HotJava browser was written in Java to show off the power of Java. But the builders also had in mind the power of what are now called applets, so they made the browser capable of executing code inside web pages. This “proof of technology” was shown at SunWorld ‘95 on May 23, 1995, and inspired the Java craze that continues today.

Sun released the first version of Java in early 1996. People quickly realized that Java 1.0 was not going to cut it for serious application development. Sure, you could use Java 1.0 to make a nervous text applet that moved text randomly around in a canvas. But you couldn’t even *print* in Java 1.0. To be blunt, Java 1.0 was not ready for prime time. Its successor, version 1.1, filled in the most obvious gaps, greatly improved the reflection capability, and added a new event model for GUI programming. It was still rather limited, though.

The big news of the 1998 JavaOne conference was the upcoming release of Java 1.2, which replaced the early toylike GUI and graphics toolkits with sophisticated and scalable versions that come a lot closer to the promise of “Write Once, Run Anywhere”™ than its predecessors. Three days after (!) its release in December 1998, Sun’s marketing department changed the name to the catchy *Java 2 Standard Edition Software Development Kit Version 1.2*.

Besides the Standard Edition, two other editions were introduced: the Micro Edition for embedded devices such as cell phones, and the Enterprise Edition for server-side processing. This book focuses on the Standard Edition.

Versions 1.3 and 1.4 of the Standard Edition are incremental improvements over the initial Java 2 release, with an ever-growing standard library, increased performance, and, of course, quite a few bug fixes. During this time, much of the initial hype about Java applets and client-side applications abated, but Java became the platform of choice for server-side applications.

Version 5.0 is the first release since version 1.1 that updates the Java *language* in significant ways. (This version was originally numbered 1.5, but the version number jumped to 5.0 at the 2004 JavaOne conference.) After many years of research, generic types (which are roughly comparable to C++ templates) have been added—the challenge was to add this feature without requiring changes in the virtual machine. Several other useful language features were inspired by C#: a “for each” loop, autoboxing, and metadata. Language changes are always a source of compatibility pain, but several of these new language features are so seductive that we think that programmers will embrace them eagerly.

Version 6 (without the .0 suffix) was released at the end of 2006. Again, there are no language changes but additional performance improvements and library enhancements.

Table 1-1 shows the evolution of the Java language and library. As you can see, the size of the application programming interface (API) has grown tremendously.

**Table 1-1 Evolution of the Java Language**

Version	Year	New Language Features	Number of Classes and Interfaces
1.0	1996	The language itself	211
1.1	1997	Inner classes	477
1.2	1998	None	1,524
1.3	2000	None	1,840
1.4	2004	Assertions	2,723
5.0	2004	Generic classes, “for each” loop, varargs, autoboxing, metadata, enumerations, static import	3,279
6	2006	None	3,777

### Common Misconceptions about Java

We close this chapter with a list of some common misconceptions about Java, along with commentary.

*Java is an extension of HTML.*

Java is a programming language; HTML is a way to describe the structure of a web page. They have nothing in common except that there are HTML extensions for placing Java applets on a web page.

*I use XML, so I don’t need Java.*

Java is a programming language; XML is a way to describe data. You can process XML data with any programming language, but the Java API contains excellent support for XML processing. In addition, many important third-party XML tools are implemented in Java. See Volume II for more information.

*Java is an easy programming language to learn.*

No programming language as powerful as Java is easy. You always have to distinguish between how easy it is to write toy programs and how hard it is to do serious work. Also, consider that only four chapters in this book discuss the Java language. The remaining chapters of both volumes show how to put the language to work, using the Java *libraries*. The Java libraries contain thousands of classes and interfaces, and tens of thousands of functions. Luckily, you do not need to know every one of them, but you do need to know surprisingly many to use Java for anything realistic.

*Java will become a universal programming language for all platforms.*

This is possible, in theory, and it is certainly the case that every vendor but Microsoft seems to want this to happen. However, many applications, already working perfectly well on desktops, would not work well on other devices or inside a browser. Also, these applications have been written to take advantage of the speed of the processor and the native user interface library and have been ported to all the important platforms anyway. Among these kinds of applications are word processors, photo editors, and web browsers. They are typically written in C or C++, and we see no benefit to the end user in rewriting them in Java.

*Java is just another programming language.*

Java is a nice programming language; most programmers prefer it over C, C++, or C#. But there have been hundreds of nice programming languages that never gained widespread popularity, whereas languages with obvious flaws, such as C++ and Visual Basic, have been wildly successful.

Why? The success of a programming language is determined far more by the utility of the *support system* surrounding it than by the elegance of its syntax. Are there useful, convenient, and standard libraries for the features that you need to implement? Are there tool vendors that build great programming and debugging environments? Does the language and the toolset integrate with the rest of the computing infrastructure? Java is successful because its class libraries let you easily do things that were hard before, such as networking and multithreading. The fact that Java reduces pointer errors is a bonus and so programmers seem to be more productive with Java, but these factors are not the source of its success.

*Now that C# is available, Java is obsolete.*

C# took many good ideas from Java, such as a clean programming language, a virtual machine, and garbage collection. But for whatever reasons, C# also left some good stuff behind, in particular security and platform independence. If you are tied to Windows, C# makes a lot of sense. But judging by the job ads, Java is still the language of choice for a majority of developers.

*Java is proprietary, and it should therefore be avoided.*

Sun Microsystems licenses Java to distributors and end users. Although Sun has ultimate control over Java through the “Java Community Process,” they have involved many other companies in the development of language revisions and the design of new libraries. Source code for the virtual machine and the libraries has always been freely available, but only for inspection, not for modification and redistribution. Up to this point, Java has been “closed source, but playing nice.”

This situation changed dramatically in 2007, when Sun announced that future versions of Java will be available under the General Public License, the same open source license that is used by Linux. It remains to be seen how Sun will manage the governance of Java in the future, but there is no question that the open sourcing of Java has been a very courageous move that will extend the life of Java by many years.

*Java is interpreted, so it is too slow for serious applications.*

In the early days of Java, the language was interpreted. Nowadays, except on “micro” platforms such as cell phones, the Java virtual machine uses a just-in-time compiler. The

“hot spots” of your code will run just as fast in Java as they would in C++, and in some cases, they will run faster.

Java does have some additional overhead over C++. Virtual machine startup time is slow, and Java GUIs are slower than their native counterparts because they are painted in a platform-independent manner.

People have complained for years that Java applications are too slow. However, today’s computers are much faster than they were when these complaints started. A slow Java program will still run quite a bit better than those blazingly fast C++ programs did a few years ago. At this point, these complaints sound like sour grapes, and some detractors have instead started to complain that Java user interfaces are ugly rather than slow.

*All Java programs run inside a web page.*

All Java *applets* run inside a web browser. That is the definition of an applet—a Java program running inside a browser. But most Java programs are stand-alone applications that run outside of a web browser. In fact, many Java programs run on web servers and produce the code for web pages.

Most of the programs in this book are stand-alone programs. Sure, applets can be fun. But stand-alone Java programs are more important and more useful in practice.

*Java programs are a major security risk.*

In the early days of Java, there were some well-publicized reports of failures in the Java security system. Most failures were in the implementation of Java in a specific browser. Researchers viewed it as a challenge to try to find chinks in the Java armor and to defy the strength and sophistication of the applet security model. The technical failures that they found have all been quickly corrected, and to our knowledge, no actual systems were ever compromised. To keep this in perspective, consider the literally millions of virus attacks in Windows executable files and Word macros that cause real grief but surprisingly little criticism of the weaknesses of the attacked platform. Also, the ActiveX mechanism in Internet Explorer would be a fertile ground for abuse, but it is so boringly obvious how to circumvent it that few researchers have bothered to publicize their findings.

Some system administrators have even deactivated Java in company browsers, while continuing to permit their users to download executable files, ActiveX controls, and Word documents. That is pretty ridiculous—currently, the risk of being attacked by hostile Java applets is perhaps comparable to the risk of dying from a plane crash; the risk of being infected by opening Word documents is comparable to the risk of dying while crossing a busy freeway on foot.

*JavaScript is a simpler version of Java.*

JavaScript, a scripting language that can be used inside web pages, was invented by Netscape and originally called LiveScript. JavaScript has a syntax that is reminiscent of Java, but otherwise there are no relationships (except for the name, of course). A subset of JavaScript is standardized as ECMA-262. JavaScript is more tightly integrated with browsers than Java applets are. In particular, a JavaScript program can modify the document that is being displayed, whereas an applet can only control the appearance of a limited area.

*With Java, I can replace my computer with a \$500 “Internet appliance.”*

When Java was first released, some people bet big that this was going to happen. Ever since the first edition of this book, we have believed it is absurd to think that home users are going to give up a powerful and convenient desktop for a limited machine with no local storage. We found the Java-powered network computer a plausible option for a “zero administration initiative” to cut the costs of computer ownership in a business, but even that has not happened in a big way.

On the other hand, Java has become widely distributed on cell phones. We must confess that we haven’t yet seen a must-have Java application running on cell phones, but the usual fare of games and screen savers seems to be selling well in many markets.



TIP: For answers to common Java questions, turn to one of the Java FAQ (frequently asked question) lists on the Web—see <http://www.ap1.jhu.edu/~hall/java/FAQs-and-Tutorials.html>.

---

# *Chapter*

# 2

## THE JAVA PROGRAMMING ENVIRONMENT

- ▼ INSTALLING THE JAVA DEVELOPMENT KIT
- ▼ CHOOSING A DEVELOPMENT ENVIRONMENT
- ▼ USING THE COMMAND-LINE TOOLS
- ▼ USING AN INTEGRATED DEVELOPMENT ENVIRONMENT
- ▼ RUNNING A GRAPHICAL APPLICATION
- ▼ BUILDING AND RUNNING APPLETS

In this chapter, you will learn how to install the Java Development Kit (JDK) and how to compile and run various types of programs: console programs, graphical applications, and applets. You run the JDK tools by typing commands in a shell window. However, many programmers prefer the comfort of an integrated development environment. We show you how to use a freely available development environment to compile and run Java programs. Although easier to learn, integrated development environments can be resource-hungry and tedious to use for small programs. As a middle ground, we show you how to use a text editor that can call the Java compiler and run Java programs. Once you have mastered the techniques in this chapter and picked your development tools, you are ready to move on to Chapter 3, where you will begin exploring the Java programming language.

### Installing the Java Development Kit

The most complete and up-to-date versions of the Java Development Kit (JDK) are available from Sun Microsystems for Solaris, Linux, and Windows. Versions in various states of development exist for the Macintosh and many other platforms, but those versions are licensed and distributed by the vendors of those platforms.



NOTE: Some Linux distributions have prepackaged versions of the JDK. For example, on Ubuntu, you can install the JDK by simply installing the `sun-java6-jdk` package with `apt-get` or the Synaptic GUI.

### Downloading the JDK

To download the Java Development Kit, you will need to navigate the Sun web site and decipher an amazing amount of jargon before you can get the software that you need. See Table 2-1 for a summary.

You already saw the abbreviation JDK for Java Development Kit. Somewhat confusingly, versions 1.2 through 1.4 of the kit were known as the Java SDK (Software Development Kit). You will still find occasional references to the old term. There is also a Java Runtime Environment (JRE) that contains the virtual machine but not the compiler. That is not what you want as a developer. It is intended for end users who have no need for the compiler.

Next, you'll see the term Java SE everywhere. That is the Java Standard Edition, in contrast to Java EE (Enterprise Edition) and Java ME (Micro Edition).

You will occasionally run into the term Java 2 that was coined in 1998 when the marketing folks at Sun felt that a fractional version number increment did not properly communicate the momentous advances of JDK 1.2. However, because they had that insight only after the release, they decided to keep the version number 1.2 for the *development kit*. Subsequent releases were numbered 1.3, 1.4, and 5.0. The *platform*, however, was renamed from Java to Java 2. Thus, we had Java 2 Standard Edition Software Development Kit Version 5.0, or J2SE SDK 5.0.

For engineers, all of this was a bit confusing, but that's why we never made it into marketing. Mercifully, in 2006, sanity prevailed. The useless Java 2 moniker was dropped and the current version of the Java Standard Edition was called Java SE 6. You will still see occasional references to versions 1.5 and 1.6—these are just synonyms for versions 5.0 and 6.



Finally, when Sun makes a minor version change to fix urgent issues, it refers to the change as an update. For example, the first update of the development kit for Java SE 6 is officially called JDK 6u1 and has the internal version number 1.6.0\_01.

If you use Solaris, Linux, or Windows, point your browser to <http://java.sun.com/javase> to download the JDK. Look for version 6 or later and pick your platform. Don't worry if the software is called an "update." The update bundles contain the most current version of the whole JDK.

Sometimes, Sun makes available bundles that contain both the Java Development Kit and an integrated development environment. That integrated environment has, at different times of its life, been named Forte, Sun ONE Studio, Sun Java Studio, and NetBeans. We do not know what the eager beavers in marketing will call it when you approach the Sun web site. We suggest that you install only the Java Development Kit at this time. If you later decide to use Sun's integrated development environment, simply download it from <http://netbeans.org>.

**Table 2-1 Java Jargon**

Name	Acronym	Explanation
Java Development Kit	JDK	The software for programmers who want to write Java programs
Java Runtime Environment	JRE	The software for consumers who want to run Java programs
Standard Edition	SE	The Java platform for use on desktops and simple server applications
Enterprise Edition	EE	The Java platform for complex server applications
Micro Edition	ME	The Java platform for use on cell phones and other small devices
Java 2	J2	An outdated term that described Java versions from 1998 until 2006
Software Development Kit	SDK	An outdated term that described the JDK from 1998 until 2006
Update	u	Sun's term for a bug fix release
NetBeans	—	Sun's integrated development environment

After downloading the JDK, follow the platform-dependent installation directions. At the time of this writing, they are available at <http://java.sun.com/javase/6/webnotes/install/index.html>.

Only the installation and compilation instructions for Java are system dependent. Once you get Java up and running, everything else in this book should apply to you. System independence is a major benefit of Java.



NOTE: The setup procedure offers a default for the installation directory that contains the JDK version number, such as `jdk1.6.0`. This sounds like a bother, but we have come to appreciate the version number—it makes it easier to install a new JDK release for testing.

Under Windows, we strongly recommend that you do not accept a default location with spaces in the path name, such as `c:\Program Files\jdk1.6.0`. Just take out the Program Files part of the path name.

In this book, we refer to the installation directory as *jdk*. For example, when we refer to the *jdk/bin* directory, we mean the directory with a name such as `/usr/local/jdk1.6.0/bin` or `c:\jdk1.6.0\bin`.

### Setting the Execution Path

After you are done installing the JDK, you need to carry out one additional step: Add the *jdk/bin* directory to the execution path, the list of directories that the operating system traverses to locate executable files. Directions for this step also vary among operating systems.

- In UNIX (including Solaris and Linux), the procedure for editing the execution path depends on the shell that you are using. If you use the C shell (which is the Solaris default), then add a line such as the following to the end of your `~/cshrc` file:

```
set path=(/usr/local/jdk/bin $path)
```

If you use the Bourne Again shell (which is the Linux default), then add a line such as the following to the end of your `~/bashrc` or `~/bash_profile` file:

```
export PATH=/usr/local/jdk/bin:$PATH
```

- Under Windows, log in as administrator. Start the Control Panel, switch to Classic View, and select the System icon. In Windows NT/2000/XP, you immediately get the system properties dialog. In Vista, you need to select Advanced System Settings (see Figure 2-1). In the system properties dialog, click the Advanced tab, then click on the Environment button. Scroll through the System Variables window until you find a variable named Path. Click the Edit button (see Figure 2-2). Add the *jdk\bin* directory to the beginning of the path, using a semicolon to separate the new entry, like this:

```
c:\jdk\bin;other stuff
```

Save your settings. Any new console windows that you start have the correct path.

Here is how you test whether you did it right: Start a shell window. Type the line

```
java -version
```

and press the ENTER key. You should get a display such as this one:

```
java version "1.6.0_01"
Java(TM) SE Runtime Environment (build 1.6.0_01-b06)
Java HotSpot(TM) Client VM (build 1.6.0_01-b06, mixed mode, sharing)
```

If instead you get a message such as “java: command not found” or “The name specified is not recognized as an internal or external command, operable program or batch file”, then you need to go back and double-check your installation.

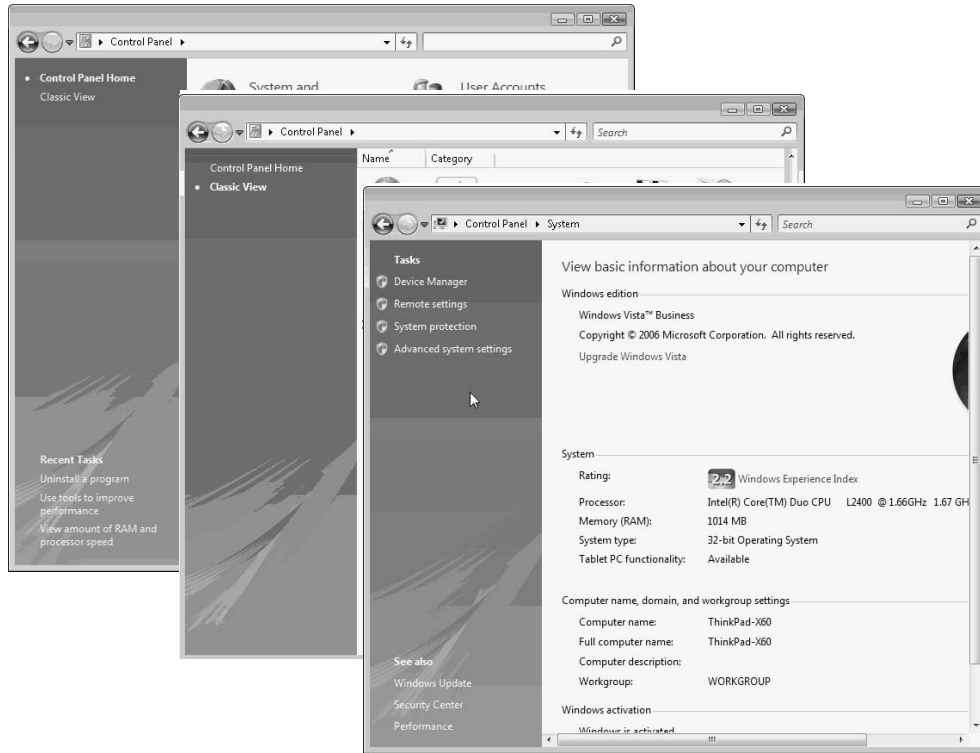


Figure 2-1 Launching the system properties dialog in Windows Vista

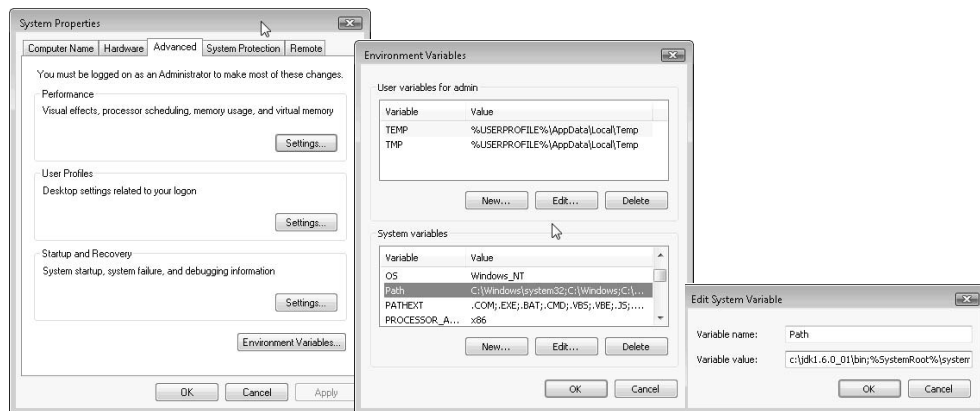


Figure 2-2 Setting the Path environment variable in Windows Vista



NOTE: In Windows, follow these instructions to open a shell window. If you use Windows NT/2000/XP, select the “Run” option from the Start menu and type `cmd`. In Vista, simply type `cmd` into the “Start Search” field in the Start menu. Press ENTER, and a shell window appears.

If you’ve never seen one of these, we suggest that you work through a tutorial that teaches the basics about the command line. Many computer science departments have tutorials on the Web, such as <http://www.cs.sjsu.edu/faculty/horstmann/CS46A/windows/tutorial.html>.

### **Installing the Library Source and Documentation**

The library source files are delivered in the JDK as a compressed file `src.zip`, and you must unpack that file to get access to the source code. We highly recommend that you do that. Simply do the following:

1. Make sure the JDK is installed and that the `jdk/bin` directory is on the execution path.
2. Open a shell window.
3. Change to the `jdk` directory (e.g., `cd /usr/local/jdk1.6.0` or `cd c:\jdk1.6.0`).
4. Make a subdirectory `src`

```
mkdir src
cd src
```
5. Execute the command

```
jar xvf ../src.zip
```

(or `jar xvf ..\src.zip` on Windows)



TIP: The `src.zip` file contains the source code for all public libraries. To obtain even more source (for the compiler, the virtual machine, the native methods, and the private helper classes), go to <http://download.java.net/jdk6>.

The documentation is contained in a compressed file that is separate from the JDK. You can download the documentation from <http://java.sun.com/javase/downloads>. Simply follow these steps:

1. Make sure the JDK is installed and that the `jdk/bin` directory is on the execution path.
2. Download the documentation zip file and move it into the `jdk` directory. The file is called `jdk-version-doc.zip`, where `version` is something like 6.
3. Open a shell window.
4. Change to the `jdk` directory.
5. Execute the command

```
jar xvf jdk-version-doc.zip
```

where `version` is the appropriate version number.

### **Installing the Core Java Program Examples**

You should also install the *Core Java* program examples. You can download them from <http://horstmann.com/corejava>. The programs are packaged into a zip file `corejava.zip`. You should unzip them into a separate directory—we recommend you call it `CoreJavaBook`. Here are the steps:

1. Make sure the JDK is installed and the `jdk/bin` directory is on the execution path.
2. Make a directory `CoreJavaBook`.
3. Download the `corejava.zip` file to that directory.
4. Open a shell window.
5. Change to the `CoreJavaBook` directory.
6. Execute the command
 

```
jar xvf corejava.zip
```

### Navigating the Java Directories

In your explorations of Java, you will occasionally want to peek inside the Java source files. And, of course, you will need to work extensively with the library documentation. Table 2-2 shows the JDK directory tree.

**Table 2-2 Java Directory Tree**

Directory Structure	Description
<code>jdk</code>	(The name may be different, for example, <code>jdk5.0</code> )
— <code>bin</code>	The compiler and tools
— <code>demo</code>	Look here for demos
— <code>docs</code>	Library documentation in HTML format (after expansion of <code>j2sdkversion-doc.zip</code> )
— <code>include</code>	Files for compiling native methods (see Volume II)
— <code>jre</code>	Java runtime environment files
— <code>lib</code>	Library files
— <code>src</code>	The library source (after expanding <code>src.zip</code> )

The two most useful subdirectories for learning Java are `docs` and `src`. The `docs` directory contains the Java library documentation in HTML format. You can view it with any web browser, such as Netscape.



**TIP:** Set a bookmark in your browser to the file `docs/api/index.html`. You will be referring to this page a lot as you explore the Java platform.

The `src` directory contains the source code for the public part of the Java libraries. As you become more comfortable with Java, you may find yourself in situations for which this book and the on-line information do not provide what you need to know. At this point, the source code for Java is a good place to begin digging. It is reassuring to know that you can always dig into the source to find out what a library function really does. For example, if you are curious about the inner workings of the `System` class, you can look inside `src/java/lang/System.java`.

### Choosing a Development Environment

If your programming experience comes from using Microsoft Visual Studio, you are accustomed to a development environment with a built-in text editor and menus to

compile and launch a program along with an integrated debugger. The basic JDK contains nothing even remotely similar. You do *everything* by typing in commands in a shell window. This sounds cumbersome, but it is nevertheless an essential skill. When you first install Java, you will want to troubleshoot your installation before you install a development environment. Moreover, by executing the basic steps yourself, you gain a better understanding of what the development environment does behind your back.

However, after you have mastered the basic steps of compiling and running Java programs, you will want to use a professional development environment. In the last decade, these environments have become so powerful and convenient that it simply doesn't make much sense to labor on without them. Two excellent choices are the freely available Eclipse and NetBeans programs. In this chapter, we show you how to get started with Eclipse since it is still a bit slicker than NetBeans, although NetBeans is catching up fast. Of course, if you prefer a different development environment, you can certainly use it with this book.

In the past, we recommended the use of a text editor such as Emacs, JEdit, or TextPad for simple programs. We no longer make this recommendation because the integrated development environments are now so fast and convenient.

In sum, we think that you should know how to use the basic JDK tools, and then you should become comfortable with an integrated development environment.

### Using the Command-Line Tools

Let us get started the hard way: compiling and launching a Java program from the command line.

1. Open a shell window.
2. Go to the `CoreJavaBook/v1ch02/Welcome` directory. (The `CoreJavaBook` directory is the directory into which you installed the source code for the book examples, as explained in the section "Installing the *Core Java* Program Examples" on page 20.)
3. Enter the following commands:

```
javac Welcome.java
java Welcome
```

You should see the output shown in Figure 2-3 in the shell window.

Congratulations! You have just compiled and run your first Java program.

What happened? The `javac` program is the Java compiler. It compiles the file `Welcome.java` into the file `Welcome.class`. The `java` program launches the Java virtual machine. It executes the bytecodes that the compiler placed in the class file.



NOTE: If you got an error message complaining about the line

```
for (String g : greeting)
```

then you probably use an older version of the Java compiler. Java SE 5.0 introduced a number of very desirable features to the Java programming language, and we take advantage of them in this book.

If you are using an older version of Java, you need to rewrite the loop as follows:

```
for (int i = 0; i < greeting.length; i++)
    System.out.println(greeting[i]);
```

---

A terminal window titled "Terminal" with a menu bar (File, Edit, View, Terminal, Tabs, Help). The terminal shows the following commands and output:

```
~$ cd CoreJavaBook/v1ch02/Welcome
~/CoreJavaBook/v1ch02/Welcome$ javac Welcome.java
~/CoreJavaBook/v1ch02/Welcome$ java Welcome
Welcome to Core Java
by Cay Horstmann
and Gary Cornell
~/CoreJavaBook/v1ch02/Welcome$
```

**Figure 2-3** Compiling and running `Welcome.java`

The `Welcome` program is extremely simple. It merely prints a message to the console. You may enjoy looking inside the program shown in Listing 2-1 (we explain how it works in the next chapter).

**Listing 2-1** `Welcome.java`

```
1. /**
2.  * This program displays a greeting from the authors.
3.  * @version 1.20 2004-02-28
4.  * @author Cay Horstmann
5.  */
6. public class Welcome
7. {
8.     public static void main(String[] args)
9.     {
10.         String[] greeting = new String[3];
11.         greeting[0] = "Welcome to Core Java";
12.         greeting[1] = "by Cay Horstmann";
13.         greeting[2] = "and Gary Cornell";
14.
15.         for (String g : greeting)
16.             System.out.println(g);
17.     }
18. }
```

**Troubleshooting Hints**

In the age of visual development environments, many programmers are unfamiliar with running programs in a shell window. Any number of things can go wrong, leading to frustrating results.

Pay attention to the following points:

- If you type in the program by hand, make sure you pay attention to uppercase and lowercase letters. In particular, the class name is `Welcome` and not `welcome` or `WELCOME`.
- The compiler requires a *file name* (`Welcome.java`). When you run the program, you specify a *class name* (`Welcome`) without a `.java` or `.class` extension.
- If you get a message such as “Bad command or file name” or “javac: command not found”, then go back and double-check your installation, in particular the execution path setting.
- If `javac` reports an error “cannot read: `Welcome.java`”, then you should check whether that file is present in the directory.

Under UNIX, check that you used the correct capitalization for `Welcome.java`. Under Windows, use the `dir` shell command, *not* the graphical Explorer tool. Some text editors (in particular Notepad) insist on adding an extension `.txt` after every file. If you use Notepad to edit `Welcome.java`, then it actually saves it as `Welcome.java.txt`. Under the default Windows settings, Explorer conspires with Notepad and hides the `.txt` extension because it belongs to a “known file type.” In that case, you need to rename the file, using the `ren` shell command, or save it again, placing quotes around the file name: “`Welcome.java`”.

- If you launch your program and get an error message complaining about a `java.lang.NoClassDefFoundError`, then carefully check the name of the offending class. If you get a complaint about `welcome` (with a lowercase `w`), then you should reissue the `java Welcome` command with an uppercase `W`. As always, case matters in Java. If you get a complaint about `Welcome/java`, then you accidentally typed `java Welcome.java`. Reissue the command as `java Welcome`.
- If you typed `java Welcome` and the virtual machine can’t find the `Welcome` class, then check if someone has set the `CLASSPATH` environment variable on your system. (It is usually not a good idea to set this variable globally, but some poorly written software installers in Windows do just that.) You can temporarily unset the `CLASSPATH` environment variable in the current shell window by typing

```
set CLASSPATH=
```

This command works on Windows and UNIX/Linux with the C shell. On UNIX/Linux with the Bourne/bash shell, use

```
export CLASSPATH=
```

- If you get an error message about a new language construct, make sure that your compiler supports Java SE 5.0.
- If you have too many errors in your program, then all the error messages fly by very quickly. The compiler sends the error messages to the standard error stream, so it’s a bit tricky to capture them if they fill more than the window can display.

Use the `2>` shell operator to redirect the errors to a file:

```
javac MyProg.java 2> errors.txt
```





TIP: The excellent tutorial at <http://java.sun.com/docs/books/tutorial/getStarted/cupojava/> goes into much greater detail about the “gotchas” that beginners can run into.

## Using an Integrated Development Environment

In this section, we show you how to compile a program with Eclipse, an integrated development environment that is freely available from <http://eclipse.org>. Eclipse is written in Java, but because it uses a nonstandard windowing library, it is not quite as portable as Java itself. Nevertheless, versions exist for Linux, Mac OS X, Solaris, and Windows.

There are other popular IDEs, but currently, Eclipse is the most commonly used. Here are the steps to get started:

1. After starting Eclipse, select File -> New Project from the menu.
2. Select “Java Project” from the wizard dialog (see Figure 2-4). These screen shots were taken with Eclipse 3.2. Don’t worry if your version of Eclipse looks slightly different.

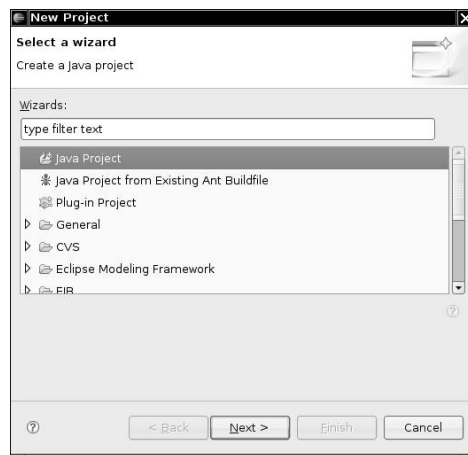
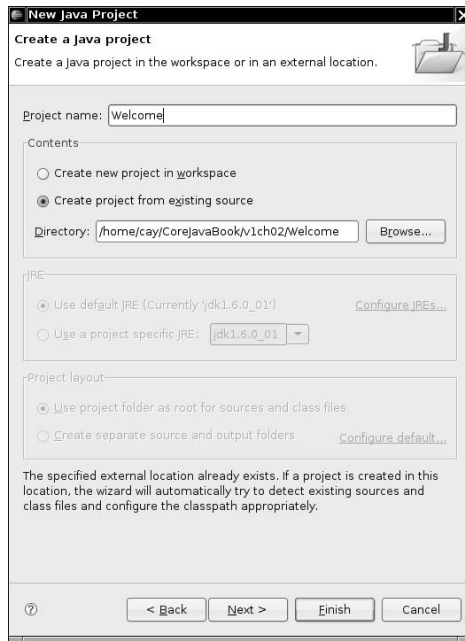


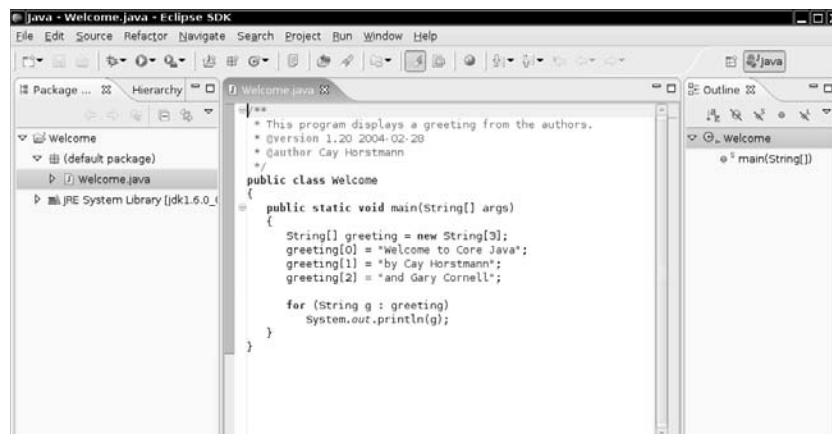
Figure 2-4 New Project dialog in Eclipse

3. Click the “Next” button. Supply the project name “Welcome” and type in the full path name of the directory that contains `Welcome.java` (see Figure 2-5).
4. Be sure to *uncheck* the option labeled “Create project in workspace”.
5. Click the “Finish” button. The project is now created.



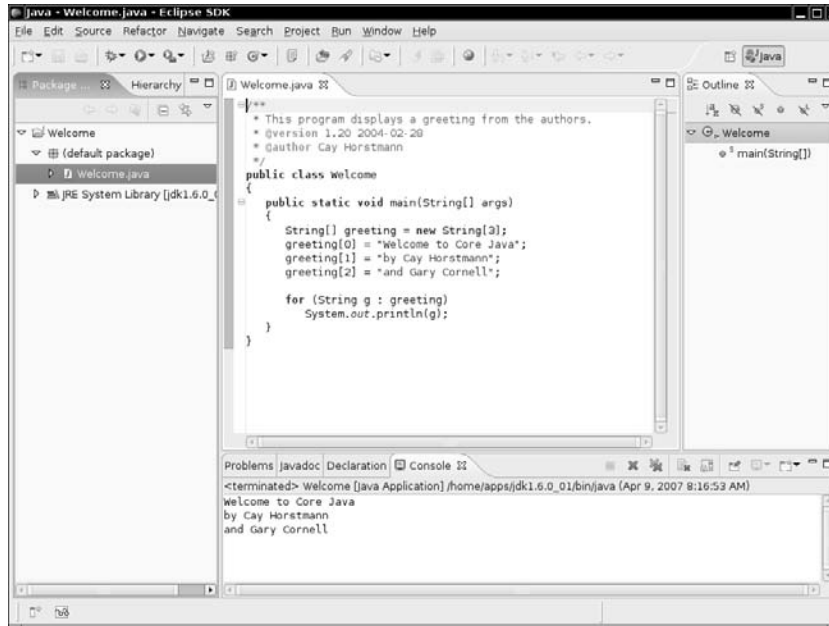
**Figure 2-5** Configuring an Eclipse project

6. Click on the triangle in the left pane next to the project window to open it, and then click on the triangle next to “Default package”. Double-click on `Welcome.java`. You should now see a window with the program code (see Figure 2-6).



**Figure 2-6** Editing a source file with Eclipse

- With the right mouse button, click on the project name (Welcome) in the leftmost pane. Select Run -> Run As -> Java Application. An output window appears at the bottom of the window. The program output is displayed in the output window (see Figure 2-7).



**Figure 2-7** Running a program in Eclipse

### Locating Compilation Errors

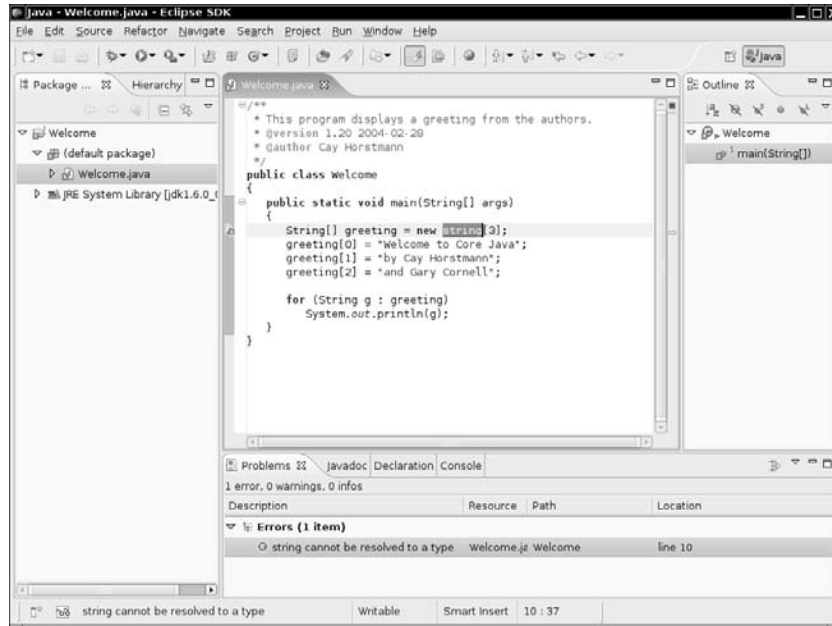
Presumably, this program did not have typos or bugs. (It was only a few lines of code, after all.) Let us suppose, for the sake of argument, that your code occasionally contains a typo (perhaps even a syntax error). Try it out—ruin our file, for example, by changing the capitalization of `String` as follows:

```
public static void main(string[] args)
```

Now, run the compiler again. You will get an error message that complains about an unknown string type (see Figure 2-8). Simply click on the error message. The cursor moves to the matching line in the edit window, where you can correct your error. This behavior allows you to fix your errors quickly.



**TIP:** Often, an Eclipse error report is accompanied by a lightbulb icon. Click on the lightbulb to get a list of suggested fixes.



**Figure 2-8** Error messages in Eclipse

These instructions should give you a taste of working in an integrated environment. We discuss the Eclipse debugger in Chapter 11.

### Running a Graphical Application

The `Welcome` program was not terribly exciting. Next, we will demonstrate a graphical application. This program is a simple image file viewer that just loads and displays an image. Again, let us first compile and run it from the command line.

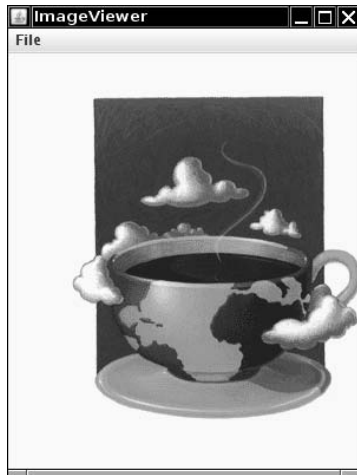
1. Open a shell window.
2. Change to the directory `CoreJavaBook/v1ch02/ImageViewer`.
3. Enter the following:
 

```
javac ImageViewer.java
java ImageViewer
```

A new program window pops up with our `ImageViewer` application (see Figure 2-9).

Now, select `File -> Open` and look for an image file to open. (We supplied a couple of sample files in the same directory.)

To close the program, click on the Close box in the title bar or pull down the system menu and close the program. (To compile and run this program inside a text editor or an integrated development environment, do the same as before. For example, for Emacs, choose `JDE -> Compile`, then choose `JDE -> Run App`.)



**Figure 2-9** Running the `ImageViewer` application

We hope that you find this program interesting and useful. Have a quick look at the source code. The program is substantially longer than the first program, but it is not terribly complex if you consider how much code it would take in C or C++ to write a similar application. In Visual Basic, of course, it is easy to write or, rather, drag and drop, such a program. The JDK does not have a visual interface builder, so you need to write code for everything, as shown in Listing 2-2. You learn how to write graphical programs like this in Chapters 7 through 9.

**Listing 2-2** `ImageViewer.java`

```
1. import java.awt.EventQueue;
2. import java.awt.event.*;
3. import java.io.*;
4. import javax.swing.*;
5.
6. /**
7.  * A program for viewing images.
8.  * @version 1.22 2007-05-21
9.  * @author Cay Horstmann
10. */
11. public class ImageViewer
12. {
13.     public static void main(String[] args)
14.     {
15.         EventQueue.invokeLater(new Runnable()
16.         {
17.             public void run()
18.             {
```

**Listing 2-2** ImageViewer.java (continued)

```
19.         JFrame frame = new ImageViewerFrame();
20.         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21.         frame.setVisible(true);
22.     }
23. });
24. }
25. }
26.
27. /**
28.  * A frame with a label to show an image.
29.  */
30. class ImageViewerFrame extends JFrame
31. {
32.     public ImageViewerFrame()
33.     {
34.         setTitle("ImageViewer");
35.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
36.
37.         // use a label to display the images
38.         label = new JLabel();
39.         add(label);
40.
41.         // set up the file chooser
42.         chooser = new JFileChooser();
43.         chooser.setCurrentDirectory(new File("."));
44.
45.         // set up the menu bar
46.         JMenuBar menuBar = new JMenuBar();
47.         setJMenuBar(menuBar);
48.
49.         JMenu menu = new JMenu("File");
50.         menuBar.add(menu);
51.
52.         JMenuItem openItem = new JMenuItem("Open");
53.         menu.add(openItem);
54.         openItem.addActionListener(new ActionListener()
55.         {
56.             public void actionPerformed(ActionEvent event)
57.             {
58.                 // show file chooser dialog
59.                 int result = chooser.showOpenDialog(null);
60.
61.                 // if file selected, set it as icon of the label
62.                 if (result == JFileChooser.APPROVE_OPTION)
63.                 {
64.                     String name = chooser.getSelectedFile().getPath();
65.                     label.setIcon(new ImageIcon(name));
66.                 }
67.             }
68.         });
```

**Listing 2-2** ImageViewer.java (continued)

```
69. JMenuItem exitItem = new JMenuItem("Exit");
70. menu.add(exitItem);
71. exitItem.addActionListener(new ActionListener()
72.     {
73.         public void actionPerformed(ActionEvent event)
74.         {
75.             System.exit(0);
76.         }
77.     });
78. }
79.
80. private JLabel label;
81. private JFileChooser chooser;
82. private static final int DEFAULT_WIDTH = 300;
83. private static final int DEFAULT_HEIGHT = 400;
84. }
```

**Building and Running Applets**

The first two programs presented in this book are Java *applications*, stand-alone programs like any native programs. On the other hand, as we mentioned in the last chapter, most of the hype about Java comes from its ability to run *applets* inside a web browser. We want to show you how to build and run an applet from the command line. Then we will load the applet into the applet viewer that comes with the JDK. Finally, we will display it in a web browser.

First, open a shell window and go to the directory `CoreJavaBook/v1ch02/WelcomeApplet`, then enter the following commands:

```
javac WelcomeApplet.java
appletviewer WelcomeApplet.html
```

Figure 2-10 shows what you see in the applet viewer window.



**Figure 2-10** WelcomeApplet applet as viewed by the applet viewer

The first command is the now-familiar command to invoke the Java compiler. This compiles the `WelcomeApplet.java` source into the bytecode file `WelcomeApplet.class`.

This time, however, you do not run the java program. You invoke the appletviewer program instead. This program is a special tool included with the JDK that lets you quickly test an applet. You need to give this program an HTML file name, rather than the name of a Java class file. The contents of the WelcomeApplet.html file are shown below in Listing 2-3.

**Listing 2-3** WelcomeApplet.html

```
1. <html>
2.   <head>
3.     <title>WelcomeApplet</title>
4.   </head>
5.   <body>
6.     <hr/>
7.     <p>
8.       This applet is from the book
9.       <a href="http://www.horstmann.com/corejava.html">Core Java</a>
10.      by <em>Cay Horstmann</em> and <em>Gary Cornell</em>,
11.      published by Sun Microsystems Press.
12.     </p>
13.     <applet code="WelcomeApplet.class" width="400" height="200">
14.       <param name="greeting" value="Welcome to Core Java!"/>
15.     </applet>
16.     <hr/>
17.     <p><a href="WelcomeApplet.java">The source.</a></p>
18.   </body>
19. </html>
```

If you are familiar with HTML, you will notice some standard HTML instructions and the applet tag, telling the applet viewer to load the applet whose code is stored in WelcomeApplet.class. The applet viewer ignores all HTML tags except for the applet tag.

Unfortunately, the browser situation is a bit messy.

- Firefox supports Java on Windows, Linux, and Mac OS X. To experiment with applets, just download the latest version, visit <http://java.com>, and use the version checker to see whether you need to install the Java Plug-in.
- Some versions of Internet Explorer have no support for Java at all. Others only support the very outdated Microsoft Java Virtual Machine. If you run Internet Explorer, go to <http://java.com> and install the Java Plug-in.
- If you have a Macintosh running OS X, then Safari is integrated with the Macintosh Java implementation, which supports Java SE 5.0 at the time of this writing.

Provided you have a browser that supports a modern version of Java, you can try loading the applet inside the browser.

1. Start your browser.
2. Select File -> Open File (or the equivalent).
3. Go to the CoreJavaBook/v1ch02/WelcomeApplet directory. You should see the WelcomeApplet.html file in the file dialog. Load the file.
4. Your browser now loads the applet, including the surrounding text. It will look something like Figure 2-11.



You can see that this application is actually alive and willing to interact with the Internet. Click on the Cay Horstmann button. The applet directs the browser to display Cay's web page. Click on the Gary Cornell button. The applet directs the browser to pop up a mail window, with Gary's e-mail address already filled in.



**Figure 2-11** Running the WelcomeApplet applet in a browser

Notice that neither of these two buttons works in the applet viewer. The applet viewer has no capabilities to send mail or display a web page, so it ignores your requests. The applet viewer is good for testing applets in isolation, but you need to put applets inside a browser to see how they interact with the browser and the Internet.

**!** TIP: You can also run applets from inside your editor or integrated development environment. In Emacs, select JDE -> Run Applet from the menu. In Eclipse, use the Run -> Run as -> Java Applet menu option.

Finally, the code for the applet is shown in Listing 2-4. At this point, do not give it more than a glance. We come back to writing applets in Chapter 10.

**Listing 2-4** WelcomeApplet.java

```

1. import java.awt.*;
2. import java.awt.event.*;
3. import java.net.*;
4. import javax.swing.*;
5.
6. /**
7.  * This applet displays a greeting from the authors.
8.  * @version 1.22 2007-04-08
9.  * @author Cay Horstmann
10. */

```

**Listing 2-4** WelcomeApplet.java (continued)

```
11. public class WelcomeApplet extends JApplet
12. {
13.     public void init()
14.     {
15.         EventQueue.invokeLater(new Runnable()
16.         {
17.             public void run()
18.             {
19.                 setLayout(new BorderLayout());
20.
21.                 JLabel label = new JLabel(getParameter("greeting"), SwingConstants.CENTER);
22.                 label.setFont(new Font("Serif", Font.BOLD, 18));
23.                 add(label, BorderLayout.CENTER);
24.
25.                 JPanel panel = new JPanel();
26.
27.                 JButton cayButton = new JButton("Cay Horstmann");
28.                 cayButton.addActionListener(makeAction("http://www.horstmann.com"));
29.                 panel.add(cayButton);
30.
31.                 JButton garyButton = new JButton("Gary Cornell");
32.                 garyButton.addActionListener(makeAction("mailto:gary_cornell@apress.com"));
33.                 panel.add(garyButton);
34.
35.                 add(panel, BorderLayout.SOUTH);
36.             }
37.         });
38.     }
39.
40.     private ActionListener makeAction(final String urlString)
41.     {
42.         return new ActionListener()
43.         {
44.             public void actionPerformed(ActionEvent event)
45.             {
46.                 try
47.                 {
48.                     getAppletContext().showDocument(new URL(urlString));
49.                 }
50.                 catch (MalformedURLException e)
51.                 {
52.                     e.printStackTrace();
53.                 }
54.             }
55.         };
56.     }
57. }
```

---

In this chapter, you learned about the mechanics of compiling and running Java programs. You are now ready to move on to Chapter 3, where you will start learning the Java language.

# *Chapter*

# 3

## FUNDAMENTAL PROGRAMMING STRUCTURES IN JAVA

- ▼ A SIMPLE JAVA PROGRAM
- ▼ COMMENTS
- ▼ DATA TYPES
- ▼ VARIABLES
- ▼ OPERATORS
- ▼ STRINGS
- ▼ INPUT AND OUTPUT
- ▼ CONTROL FLOW
- ▼ BIG NUMBERS
- ▼ ARRAYS

**A**t this point, we are assuming that you successfully installed the JDK and were able to run the sample programs that we showed you in Chapter 2. It's time to start programming. This chapter shows you how the basic programming concepts such as data types, branches, and loops are implemented in Java.

Unfortunately, in Java you can't easily write a program that uses a GUI—you need to learn a fair amount of machinery to put up windows, add text boxes and buttons that respond to them, and so on. Because introducing the techniques needed to write GUI-based Java programs would take us too far away from our goal of introducing the basic programming concepts, the sample programs in this chapter are “toy” programs, designed to illustrate a concept. All these examples simply use a shell window for input and output.

Finally, if you are an experienced C++ programmer, you can get away with just skimming this chapter: Concentrate on the C/C++ notes that are interspersed throughout the text. Programmers coming from another background, such as Visual Basic, will find most of the concepts familiar, but all of the syntax very different—you should read this chapter very carefully.

### A Simple Java Program

Let's look more closely at about the simplest Java program you can have—one that simply prints a message to the console window:

```
public class FirstSample
{
    public static void main(String[] args)
    {
        System.out.println("We will not use 'Hello, World!'");
    }
}
```

It is worth spending all the time that you need to become comfortable with the framework of this sample; the pieces will recur in all applications. First and foremost, *Java is case sensitive*. If you made any mistakes in capitalization (such as typing `Main` instead of `main`), the program will not run.

Now let's look at this source code line by line. The keyword `public` is called an *access modifier*; these modifiers control the level of access other parts of a program have to this code. We have more to say about access modifiers in Chapter 5. The keyword `class` reminds you that everything in a Java program lives inside a class. Although we spend a lot more time on classes in the next chapter, for now think of a class as a container for the program logic that defines the behavior of an application. As mentioned in Chapter 1, classes are the building blocks with which all Java applications and applets are built. *Everything* in a Java program must be inside a class.

Following the keyword `class` is the name of the class. The rules for class names in Java are quite generous. Names must begin with a letter, and after that, they can have any combination of letters and digits. The length is essentially unlimited. You cannot use a Java reserved word (such as `public` or `class`) for a class name. (See the Appendix for a list of reserved words.)

The standard naming convention (which we follow in the name `FirstSample`) is that class names are nouns that start with an uppercase letter. If a name consists of multiple words, use an initial uppercase letter in each of the words. (This use of uppercase letters in the middle of a word is sometimes called “camel case” or, self-referentially, “CamelCase.”)

You need to make the file name for the source code the same as the name of the public class, with the extension `.java` appended. Thus, you must store this code in a file called `FirstSample.java`. (Again, case is important—don’t use `firstsample.java`.)

If you have named the file correctly and not made any typos in the source code, then when you compile this source code, you end up with a file containing the bytecodes for this class. The Java compiler automatically names the bytecode file `FirstSample.class` and stores it in the same directory as the source file. Finally, launch the program by issuing the following command:

```
java FirstSample
```

(Remember to leave off the `.class` extension.) When the program executes, it simply displays the string `We will not use 'Hello, World!'` on the console.

When you use

```
java ClassName
```

to run a compiled program, the Java virtual machine always starts execution with the code in the `main` method in the class you indicate. (The term “method” is Java-speak for a function.) Thus, you *must* have a `main` method in the source file for your class for your code to execute. You can, of course, add your own methods to a class and call them from the `main` method. (We cover writing your own methods in the next chapter.)



**NOTE:** According to the Java Language Specification, the `main` method must be declared `public`. (The Java Language Specification is the official document that describes the Java language. You can view or download it from <http://java.sun.com/docs/books/jls>.)

However, several versions of the Java launcher were willing to execute Java programs even when the `main` method was not `public`. A programmer filed a bug report. To see it, visit the site <http://bugs.sun.com/bugdatabase/index.jsp> and enter the bug identification number 4252539. That bug was marked as “closed, will not be fixed.” A Sun engineer added an explanation that the Java Virtual Machine Specification (at <http://java.sun.com/docs/books/vmspec>) does not mandate that `main` is `public` and that “fixing it will cause potential troubles.” Fortunately, sanity finally prevailed. The Java launcher in Java SE 1.4 and beyond enforces that the `main` method is `public`.

There are a couple of interesting aspects about this story. On the one hand, it is frustrating to have quality assurance engineers, who are often overworked and not always experts in the fine points of Java, make questionable decisions about bug reports. On the other hand, it is remarkable that Sun puts the bug reports and their resolutions onto the Web, for anyone to scrutinize. The “bug parade” is a very useful resource for programmers. You can even vote for your favorite bug. Bugs with lots of votes have a high chance of being fixed in the next JDK release.

Notice the braces { } in the source code. In Java, as in C/C++, braces delineate the parts (usually called *blocks*) in your program. In Java, the code for any method must be started by an opening brace { and ended by a closing brace }.

Brace styles have inspired an inordinate amount of useless controversy. We use a style that lines up matching braces. Because whitespace is irrelevant to the Java compiler, you can use whatever brace style you like. We will have more to say about the use of braces when we talk about the various kinds of loops.

For now, don't worry about the keywords `static void`—just think of them as part of what you need to get a Java program to compile. By the end of Chapter 4, you will understand this incantation completely. The point to remember for now is that every Java application must have a `main` method that is declared in the following way:

```
public class ClassName
{
    public static void main(String[] args)
    {
        program statements
    }
}
```



**C++ NOTE:** As a C++ programmer, you know what a class is. Java classes are similar to C++ classes, but there are a few differences that can trap you. For example, in Java *all* functions are methods of some class. (The standard terminology refers to them as methods, not member functions.) Thus, in Java you must have a shell class for the `main` method. You may also be familiar with the idea of *static member functions* in C++. These are member functions defined inside a class that do not operate on objects. The `main` method in Java is always static. Finally, as in C/C++, the `void` keyword indicates that this method does not return a value. Unlike C/C++, the `main` method does not return an “exit code” to the operating system. If the `main` method exits normally, the Java program has the exit code 0, indicating successful completion. To terminate the program with a different exit code, use the `System.exit` method.

Next, turn your attention to this fragment:

```
{
    System.out.println("We will not use 'Hello, World!'");
}
```

Braces mark the beginning and end of the *body* of the method. This method has only one statement in it. As with most programming languages, you can think of Java statements as being the sentences of the language. In Java, every statement must end with a semicolon. In particular, carriage returns do not mark the end of a statement, so statements can span multiple lines if need be.

The body of the `main` method contains a statement that outputs a single line of text to the console.

Here, we are using the `System.out` object and calling its `println` method. Notice the periods used to invoke a method. Java uses the general syntax

```
object.method(parameters)
```

for its equivalent of function calls.

In this case, we are calling the `println` method and passing it a string parameter. The method displays the string parameter on the console. It then terminates the output line so that each call to `println` displays its output on a new line. Notice that Java, like C/C++, uses double quotes to delimit strings. (You can find more information about strings later in this chapter.)

Methods in Java, like functions in any programming language, can use zero, one, or more *parameters* (some programmers call them *arguments*). Even if a method takes no parameters, you must still use empty parentheses. For example, a variant of the `println` method with no parameters just prints a blank line. You invoke it with the call

```
System.out.println();
```



**NOTE:** `System.out` also has a `print` method that doesn't add a new line character to the output. For example, `System.out.print("Hello")` prints `Hello` without a new line. The next output appears immediately after the letter `o`.

## Comments

Comments in Java, like comments in most programming languages, do not show up in the executable program. Thus, you can add as many comments as needed without fear of bloating the code. Java has three ways of marking comments. The most common method is a `//`. You use this for a comment that will run from the `//` to the end of the line.

```
System.out.println("We will not use 'Hello, World!'); // is this too cute?
```

When longer comments are needed, you can mark each line with a `//`. Or you can use the `/*` and `*/` comment delimiters that let you block off a longer comment. This is shown in Listing 3-1.

### Listing 3-1 FirstSample.java

```
1. /**
2.  * This is the first sample program in Core Java Chapter 3
3.  * @version 1.01 1997-03-22
4.  * @author Gary Cornell
5.  */
6. public class FirstSample
7. {
8.     public static void main(String[] args)
9.     {
10.         System.out.println("We will not use 'Hello, World!');
11.     }
12. }
```

Finally, a third kind of comment can be used to generate documentation automatically. This comment uses a `/**` to start and a `*/` to end. For more on this type of comment and on automatic documentation generation, see Chapter 4.



CAUTION: `/* */` comments do not nest in Java. That is, you cannot deactivate code simply by surrounding it with `/*` and `*/` because the code that you want to deactivate might itself contain a `*/` delimiter.

## Data Types

Java is a *strongly typed language*. This means that every variable must have a declared type. There are eight *primitive types* in Java. Four of them are integer types; two are floating-point number types; one is the character type `char`, used for code units in the Unicode encoding scheme (see the section “The `char` Type” on page 42); and one is a `boolean` type for truth values.



NOTE: Java has an arbitrary precision arithmetic package. However, “big numbers,” as they are called, are Java *objects* and not a new Java type. You see how to use them later in this chapter.

## Integer Types

The integer types are for numbers without fractional parts. Negative values are allowed. Java provides the four integer types shown in Table 3–1.

**Table 3–1 Java Integer Types**

Type	Storage Requirement	Range (Inclusive)
<code>int</code>	4 bytes	–2,147,483,648 to 2,147,483,647 (just over 2 billion)
<code>short</code>	2 bytes	–32,768 to 32,767
<code>long</code>	8 bytes	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
<code>byte</code>	1 byte	–128 to 127

In most situations, the `int` type is the most practical. If you want to represent the number of inhabitants of our planet, you’ll need to resort to a `long`. The `byte` and `short` types are mainly intended for specialized applications, such as low-level file handling, or for large arrays when storage space is at a premium.

Under Java, the ranges of the integer types do not depend on the machine on which you will be running the Java code. This alleviates a major pain for the programmer who wants to move software from one platform to another, or even between operating systems on the same platform. In contrast, C and C++ programs use the most efficient integer type for each processor. As a result, a C program that runs well on a 32-bit processor may exhibit integer overflow on a 16-bit system. Because Java programs must run with the same results on all machines, the ranges for the various types are fixed.

Long integer numbers have a suffix `L` (for example, `4000000000L`). Hexadecimal numbers have a prefix `0x` (for example, `0xCAFE`). Octal numbers have a prefix `0`. For example, `010` is 8. Naturally, this can be confusing, and we recommend against the use of octal constants.





**C++ NOTE:** In C and C++, `int` denotes the integer type that depends on the target machine. On a 16-bit processor, like the 8086, integers are 2 bytes. On a 32-bit processor like the Sun SPARC, they are 4-byte quantities. On an Intel Pentium, the integer type of C and C++ depends on the operating system: For DOS and Windows 3.1, integers are 2 bytes. When 32-bit mode is used for Windows programs, integers are 4 bytes. In Java, the sizes of all numeric types are platform independent.

Note that Java does not have any unsigned types.

### Floating-Point Types

The floating-point types denote numbers with fractional parts. The two floating-point types are shown in Table 3–2.

**Table 3–2 Floating-Point Types**

Type	Storage Requirement	Range
<code>float</code>	4 bytes	approximately $\pm 3.40282347\text{E}+38\text{F}$ (6–7 significant decimal digits)
<code>double</code>	8 bytes	approximately $\pm 1.79769313486231570\text{E}+308$ (15 significant decimal digits)

The name `double` refers to the fact that these numbers have twice the precision of the `float` type. (Some people call these *double-precision* numbers.) Here, the type to choose in most applications is `double`. The limited precision of `float` is simply not sufficient for many situations. Seven significant (decimal) digits may be enough to precisely express your annual salary in dollars and cents, but it won't be enough for your company president's salary. The only reasons to use `float` are in the rare situations in which the slightly faster processing of single-precision numbers is important or when you need to store a large number of them.

Numbers of type `float` have a suffix `F` (for example, `3.402F`). Floating-point numbers without an `F` suffix (such as `3.402`) are always considered to be of type `double`. You can optionally supply the `D` suffix (for example, `3.402D`).



**NOTE:** As of Java SE 5.0, you can specify floating-point numbers in hexadecimal! For example,  $0.125 = 2^{-3}$  can be written as `0x1.0p-3`. In hexadecimal notation, you use a `p`, not an `e`, to denote the exponent. Note that the mantissa is written in hexadecimal and the exponent in decimal. The base of the exponent is 2, not 10.

All floating-point computations follow the IEEE 754 specification. In particular, there are three special floating-point values to denote overflows and errors:

- Positive infinity
- Negative infinity
- NaN (not a number)

For example, the result of dividing a positive number by 0 is positive infinity. Computing 0/0 or the square root of a negative number yields NaN.



**NOTE:** The constants `Double.POSITIVE_INFINITY`, `Double.NEGATIVE_INFINITY`, and `Double.NaN` (as well as corresponding `Float` constants) represent these special values, but they are rarely used in practice. In particular, you cannot test

```
if (x == Double.NaN) // is never true
```

to check whether a particular result equals `Double.NaN`. All “not a number” values are considered distinct. However, you can use the `Double.isNaN` method:

```
if (Double.isNaN(x)) // check whether x is "not a number"
```



**CAUTION:** Floating-point numbers are *not* suitable for financial calculation in which roundoff errors cannot be tolerated. For example, the command `System.out.println(2.0 - 1.1)` prints `0.8999999999999999`, not `0.9` as you would expect. Such roundoff errors are caused by the fact that floating-point numbers are represented in the binary number system. There is no precise binary representation of the fraction  $1/10$ , just as there is no accurate representation of the fraction  $1/3$  in the decimal system. If you need precise numerical computations without roundoff errors, use the `BigDecimal` class, which is introduced later in this chapter.

### The char Type

The `char` type is used to describe individual characters. Most commonly, these will be character constants. For example, `'A'` is a character constant with value 65. It is different from `"A"`, a string containing a single character. Unicode code units can be expressed as hexadecimal values that run from `\u0000` to `\uFFFF`. For example, `\u2122` is the trademark symbol (™) and `\u03C0` is the Greek letter pi ( $\pi$ ).

Besides the `\u` escape sequences that indicate the encoding of Unicode code units, there are several escape sequences for special characters, as shown in Table 3–3. You can use these escape sequences inside quoted character constants and strings, such as `'\u2122'` or `"Hello\n"`. The `\u` escape sequence (but none of the other escape sequences) can even be used *outside* quoted character constants and strings. For example,

```
public static void main(String\u005B\u005D args)
```

is perfectly legal—`\u005B` and `\u005D` are the encodings for `[` and `]`.

**Table 3–3 Escape Sequences for Special Characters**

Escape Sequence	Name	Unicode Value
<code>\b</code>	Backspace	<code>\u0008</code>
<code>\t</code>	Tab	<code>\u0009</code>
<code>\n</code>	Linefeed	<code>\u000a</code>
<code>\r</code>	Carriage return	<code>\u000d</code>

**Table 3–3 Escape Sequences for Special Characters (continued)**

Escape Sequence	Name	Unicode Value
<code>\"</code>	Double quote	<code>\u0022</code>
<code>\'</code>	Single quote	<code>\u0027</code>
<code>\\</code>	Backslash	<code>\u005c</code>

To fully understand the `char` type, you have to know about the Unicode encoding scheme. Unicode was invented to overcome the limitations of traditional character encoding schemes. Before Unicode, there were many different standards: ASCII in the United States, ISO 8859-1 for Western European languages, KOI-8 for Russian, GB18030 and BIG-5 for Chinese, and so on. This causes two problems. A particular code value corresponds to different letters in the various encoding schemes. Moreover, the encodings for languages with large character sets have variable length: Some common characters are encoded as single bytes, others require two or more bytes.

Unicode was designed to solve these problems. When the unification effort started in the 1980s, a fixed 2-byte width code was more than sufficient to encode all characters used in all languages in the world, with room to spare for future expansion—or so everyone thought at the time. In 1991, Unicode 1.0 was released, using slightly less than half of the available 65,536 code values. Java was designed from the ground up to use 16-bit Unicode characters, which was a major advance over other programming languages that used 8-bit characters.

Unfortunately, over time, the inevitable happened. Unicode grew beyond 65,536 characters, primarily due to the addition of a very large set of ideographs used for Chinese, Japanese, and Korean. Now, the 16-bit `char` type is insufficient to describe all Unicode characters.

We need a bit of terminology to explain how this problem is resolved in Java, beginning with Java SE 5.0. A *code point* is a code value that is associated with a character in an encoding scheme. In the Unicode standard, code points are written in hexadecimal and prefixed with `U+`, such as `U+0041` for the code point of the letter A. Unicode has code points that are grouped into 17 *code planes*. The first code plane, called the *basic multilingual plane*, consists of the “classic” Unicode characters with code points `U+0000` to `U+FFFF`. Sixteen additional planes, with code points `U+10000` to `U+10FFFF`, hold the *supplementary characters*.

The UTF-16 encoding is a method of representing all Unicode code points in a variable-length code. The characters in the basic multilingual plane are represented as 16-bit values, called *code units*. The supplementary characters are encoded as consecutive pairs of code units. Each of the values in such an encoding pair falls into a range of 2048 unused values of the basic multilingual plane, called the *surrogates area* (`U+D800` to `U+DBFF` for the first code unit, `U+DC00` to `U+DFFF` for the second code unit). This is rather clever, because you can immediately tell whether a code unit encodes a single character or whether it is the first or second part of a supplementary character. For example, the mathematical symbol for the set of integers  $\mathbb{Z}$  has code point `U+1D56B`

and is encoded by the two code units U+D835 and U+DD6B. (See <http://en.wikipedia.org/wiki/UTF-16> for a description of the encoding algorithm.)

In Java, the `char` type describes a *code unit* in the UTF-16 encoding.

Our strong recommendation is not to use the `char` type in your programs unless you are actually manipulating UTF-16 code units. You are almost always better off treating strings (which we will discuss in the section “Strings” on page 53) as abstract data types.

### The boolean Type

The `boolean` type has two values, `false` and `true`. It is used for evaluating logical conditions. You cannot convert between integers and `boolean` values.



**C++ NOTE:** In C++, numbers and even pointers can be used in place of `boolean` values. The value 0 is equivalent to the `bool` value `false`, and a non-zero value is equivalent to `true`. This is *not* the case in Java. Thus, Java programmers are shielded from accidents such as

```
if (x = 0) // oops...meant x == 0
```

In C++, this test compiles and runs, always evaluating to `false`. In Java, the test does not compile because the integer expression `x = 0` cannot be converted to a `boolean` value.

### Variables

In Java, every variable has a *type*. You declare a variable by placing the type first, followed by the name of the variable. Here are some examples:

```
double salary;
int vacationDays;
long earthPopulation;
boolean done;
```

Notice the semicolon at the end of each declaration. The semicolon is necessary because a declaration is a complete Java statement.

A variable name must begin with a letter and must be a sequence of letters or digits. Note that the terms “letter” and “digit” are much broader in Java than in most languages. A letter is defined as 'A'-'Z', 'a'-'z', '\_', or *any* Unicode character that denotes a letter in a language. For example, German users can use umlauts such as 'ä' in variable names; Greek speakers could use a  $\pi$ . Similarly, digits are '0'-'9' and *any* Unicode characters that denote a digit in a language. Symbols like '+' or '@' cannot be used inside variable names, nor can spaces. *All* characters in the name of a variable are significant and *case is also significant*. The length of a variable name is essentially unlimited.



**TIP:** If you are really curious as to what Unicode characters are “letters” as far as Java is concerned, you can use the `isJavaIdentifierStart` and `isJavaIdentifierPart` methods in the `Character` class to check.

You also cannot use a Java reserved word for a variable name. (See the Appendix for a list of reserved words.)

You can have multiple declarations on a single line:

```
int i, j; // both are integers
```

However, we don't recommend this style. If you declare each variable separately, your programs are easier to read.



**NOTE:** As you saw, names are case sensitive, for example, `hireDay` and `hireDay` are two separate names. In general, you should not have two names that only differ in their letter case. However, sometimes it is difficult to come up with a good name for a variable. Many programmers then give the variable the same name of the type, such as

```
Box box; // ok--Box is the type and box is the variable name
```

Other programmers prefer to use an "a" prefix for the variable:

```
Box aBox;
```

### Initializing Variables

After you declare a variable, you must explicitly initialize it by means of an assignment statement—you can never use the values of uninitialized variables. For example, the Java compiler flags the following sequence of statements as an error:

```
int vacationDays;  
System.out.println(vacationDays); // ERROR--variable not initialized
```

You assign to a previously declared variable by using the variable name on the left, an equal sign (=), and then some Java expression that has an appropriate value on the right.

```
int vacationDays;  
vacationDays = 12;
```

You can both declare and initialize a variable on the same line. For example:

```
int vacationDays = 12;
```

Finally, in Java you can put declarations anywhere in your code. For example, the following is valid code in Java:

```
double salary = 65000.0;  
System.out.println(salary);  
int vacationDays = 12; // ok to declare a variable here
```

In Java, it is considered good style to declare variables as closely as possible to the point where they are first used.



**C++ NOTE:** C and C++ distinguish between the *declaration* and *definition* of variables. For example,

```
int i = 10;  
is a definition, whereas
```

```
extern int i;
```

is a declaration. In Java, no declarations are separate from definitions.

### Constants

In Java, you use the keyword `final` to denote a constant. For example:

```
public class Constants
{
    public static void main(String[] args)
    {
        final double CM_PER_INCH = 2.54;
        double paperWidth = 8.5;
        double paperHeight = 11;
        System.out.println("Paper size in centimeters: "
            + paperWidth * CM_PER_INCH + " by " + paperHeight * CM_PER_INCH);
    }
}
```

The keyword `final` indicates that you can assign to the variable once, and then its value is set once and for all. It is customary to name constants in all uppercase.

It is probably more common in Java to want a constant that is available to multiple methods inside a single class. These are usually called *class constants*. You set up a class constant with the keywords `static final`. Here is an example of using a class constant:

```
public class Constants2
{
    public static void main(String[] args)
    {
        double paperWidth = 8.5;
        double paperHeight = 11;
        System.out.println("Paper size in centimeters: "
            + paperWidth * CM_PER_INCH + " by " + paperHeight * CM_PER_INCH);
    }

    public static final double CM_PER_INCH = 2.54;
}
```

Note that the definition of the class constant appears *outside* the `main` method. Thus, the constant can also be used in other methods of the same class. Furthermore, if (as in our example) the constant is declared `public`, methods of other classes can also use the constant—in our example, as `Constants2.CM_PER_INCH`.



**C++ NOTE:** `const` is a reserved Java keyword, but it is not currently used for anything. You must use `final` for a constant.

### Operators

The usual arithmetic operators `+` `-` `*` `/` are used in Java for addition, subtraction, multiplication, and division. The `/` operator denotes integer division if both arguments are integers, and floating-point division otherwise. Integer remainder (sometimes called *modulus*) is denoted by `%`. For example, `15 / 2` is 7, `15 % 2` is 1, and `15.0 / 2` is 7.5.

Note that integer division by 0 raises an exception, whereas floating-point division by 0 yields an infinite or NaN result.

There is a convenient shortcut for using binary arithmetic operators in an assignment. For example,

```
x += 4;
```

is equivalent to

```
x = x + 4;
```

(In general, place the operator to the left of the = sign, such as \*= or %=.)



**NOTE:** One of the stated goals of the Java programming language is portability. A computation should yield the same results no matter which virtual machine executes it. For arithmetic computations with floating-point numbers, it is surprisingly difficult to achieve this portability. The `double` type uses 64 bits to store a numeric value, but some processors use 80-bit floating-point registers. These registers yield added precision in intermediate steps of a computation. For example, consider the following computation:

```
double w = x * y / z;
```

Many Intel processors compute  $x * y$  and leave the result in an 80-bit register, then divide by  $z$ , and finally truncate the result back to 64 bits. That can yield a more accurate result, and it can avoid exponent overflow. But the result may be *different* than a computation that uses 64 bits throughout. For that reason, the initial specification of the Java virtual machine mandated that all intermediate computations must be truncated. The numeric community hated it. Not only can the truncated computations cause overflow, they are actually *slower* than the more precise computations because the truncation operations take time. For that reason, the Java programming language was updated to recognize the conflicting demands for optimum performance and perfect reproducibility. By default, virtual machine designers are now permitted to use extended precision for intermediate computations. However, methods tagged with the `strictfp` keyword must use strict floating-point operations that yield reproducible results. For example, you can tag `main` as

```
public static strictfp void main(String[] args)
```

Then all instructions inside the `main` method use strict floating-point computations. If you tag a class as `strictfp`, then all of its methods use strict floating-point computations.

The gory details are very much tied to the behavior of the Intel processors. In default mode, intermediate results are allowed to use an extended exponent, but not an extended mantissa. (The Intel chips support truncation of the mantissa without loss of performance.) Therefore, the only difference between default and strict mode is that strict computations may overflow when default computations don't.

If your eyes glazed over when reading this note, don't worry. Floating-point overflow isn't a problem that one encounters for most common programs. We don't use the `strictfp` keyword in this book.

### ***Increment and Decrement Operators***

Programmers, of course, know that one of the most common operations with a numeric variable is to add or subtract 1. Java, following in the footsteps of C and C++, has both increment and decrement operators: `n++` adds 1 to the current value of the variable `n`, and `n--` subtracts 1 from it. For example, the code

```
int n = 12;  
n++;
```

changes `n` to 13. Because these operators change the value of a variable, they cannot be applied to numbers themselves. For example, `4++` is not a legal statement.

There are actually two forms of these operators; you have seen the “postfix” form of the operator that is placed after the operand. There is also a prefix form, `++n`. Both change the value of the variable by 1. The difference between the two only appears when they are used inside expressions. The prefix form does the addition first; the postfix form evaluates to the old value of the variable.

```
int m = 7;
int n = 7;
int a = 2 * ++m; // now a is 16, m is 8
int b = 2 * n++; // now b is 14, n is 8
```

We recommend against using `++` inside other expressions because this often leads to confusing code and annoying bugs.

(Of course, while it is true that the `++` operator gives the C++ language its name, it also led to the first joke about the language. C++ haters point out that even the name of the language contains a bug: “After all, it should really be called `++C`, because we only want to use a language after it has been improved.”)

### **Relational and boolean Operators**

Java has the full complement of relational operators. To test for equality you use a double equal sign, `==`. For example, the value of

```
3 == 7
```

is `false`.

Use a `!=` for inequality. For example, the value of

```
3 != 7
```

is `true`.

Finally, you have the usual `<` (less than), `>` (greater than), `<=` (less than or equal), and `>=` (greater than or equal) operators.

Java, following C++, uses `&&` for the logical “and” operator and `||` for the logical “or” operator. As you can easily remember from the `!=` operator, the exclamation point `!` is the logical negation operator. The `&&` and `||` operators are evaluated in “short circuit” fashion. The second argument is not evaluated if the first argument already determines the value. If you combine two expressions with the `&&` operator,

```
expression1 && expression2
```

and the truth value of the first expression has been determined to be `false`, then it is impossible for the result to be `true`. Thus, the value for the second expression is *not* calculated. This behavior can be exploited to avoid errors. For example, in the expression

```
x != 0 && 1 / x > x + y // no division by 0
```

the second part is never evaluated if `x` equals zero. Thus, `1 / x` is not computed if `x` is zero, and no divide-by-zero error can occur.

Similarly, the value of `expression1 || expression2` is automatically `true` if the first expression is `true`, without evaluation of the second expression.



Finally, Java supports the ternary `?:` operator that is occasionally useful. The expression

```
condition ? expression1 : expression2
```

evaluates to the first expression if the condition is true, to the second expression otherwise. For example,

```
x < y ? x : y
```

gives the smaller of `x` and `y`.

### Bitwise Operators

When working with any of the integer types, you have operators that can work directly with the bits that make up the integers. This means that you can use masking techniques to get at individual bits in a number. The bitwise operators are

```
& ("and")   | ("or")   ^ ("xor")   ~ ("not")
```

These operators work on bit patterns. For example, if `n` is an integer variable, then

```
int fourthBitFromRight = (n & 8) / 8;
```

gives you a 1 if the fourth bit from the right in the binary representation of `n` is 1, and 0 if not. Using `&` with the appropriate power of 2 lets you mask out all but a single bit.



**NOTE:** When applied to boolean values, the `&` and `|` operators yield a boolean value. These operators are similar to the `&&` and `||` operators, except that the `&` and `|` operators are not evaluated in “short circuit” fashion. That is, both arguments are first evaluated before the result is computed.

There are also `>>` and `<<` operators, which shift a bit pattern to the right or left. These operators are often convenient when you need to build up bit patterns to do bit masking:

```
int fourthBitFromRight = (n & (1 << 3)) >> 3;
```

Finally, a `>>>` operator fills the top bits with zero, whereas `>>` extends the sign bit into the top bits. There is no `<<<` operator.



**CAUTION:** The right-hand side argument of the shift operators is reduced modulo 32 (unless the left-hand side is a `long`, in which case the right-hand side is reduced modulo 64). For example, the value of `1 << 35` is the same as `1 << 3` or 8.



**C++ NOTE:** In C/C++, there is no guarantee as to whether `>>` performs an arithmetic shift (extending the sign bit) or a logical shift (filling in with zeroes). Implementors are free to choose whatever is more efficient. That means the C/C++ `>>` operator is really only defined for non-negative numbers. Java removes that ambiguity.


### Mathematical Functions and Constants

The `Math` class contains an assortment of mathematical functions that you may occasionally need, depending on the kind of programming that you do.

To take the square root of a number, you use the `sqrt` method:

```
double x = 4;
double y = Math.sqrt(x);
System.out.println(y); // prints 2.0
```

---

 **NOTE:** There is a subtle difference between the `println` method and the `sqrt` method. The `println` method operates on an object, `System.out`, defined in the `System` class. But the `sqrt` method in the `Math` class does not operate on any object. Such a method is called a *static* method. You can learn more about static methods in Chapter 4.

---

The Java programming language has no operator for raising a quantity to a power: You must use the `pow` method in the `Math` class. The statement

```
double y = Math.pow(x, a);
```

sets `y` to be `x` raised to the power `a` ( $x^a$ ). The `pow` method has parameters that are both of type `double`, and it returns a `double` as well.

The `Math` class supplies the usual trigonometric functions

```
Math.sin
Math.cos
Math.tan
Math.atan
Math.atan2
```


and the exponential function and its inverse, the natural log:

```
Math.exp
Math.log
```

Finally, two constants denote the closest possible approximations to the mathematical constants  $\pi$  and  $e$ :

```
Math.PI
Math.E
```

---

 **TIP:** Starting with Java SE 5.0, you can avoid the `Math` prefix for the mathematical methods and constants by adding the following line to the top of your source file:

```
import static java.lang.Math.*;
```


For example:

```
System.out.println("The square root of \u03C0 is " + sqrt(PI));
```

We discuss static imports in Chapter 4.

---

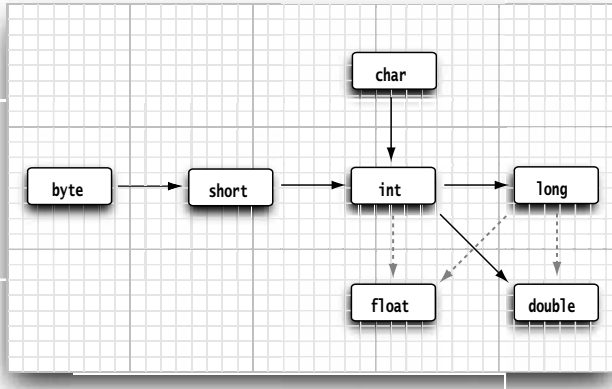
---

 **NOTE:** The functions in the `Math` class use the routines in the computer's floating-point unit for fastest performance. If completely predictable results are more important than fast performance, use the `StrictMath` class instead. It implements the algorithms from the "Freely Distributable Math Library" `fdlibm`, guaranteeing identical results on all platforms. See <http://www.netlib.org/fdlibm/index.html> for the source of these algorithms. (Whenever `fdlibm` provides more than one definition for a function, the `StrictMath` class follows the IEEE 754 version whose name starts with an "e".)

---

### Conversions between Numeric Types

It is often necessary to convert from one numeric type to another. Figure 3–1 shows the legal conversions.



**Figure 3–1** Legal conversions between numeric types

The six solid arrows in Figure 3–1 denote conversions without information loss. The three dotted arrows denote conversions that may lose precision. For example, a large integer such as 123456789 has more digits than the `float` type can represent. When the integer is converted to a `float`, the resulting value has the correct magnitude but it loses some precision.

```
int n = 123456789;
float f = n; // f is 1.23456792E8
```

When two values with a binary operator (such as `n + f` where `n` is an integer and `f` is a floating-point value) are combined, both operands are converted to a common type before the operation is carried out.

- If either of the operands is of type `double`, the other one will be converted to a `double`.
- Otherwise, if either of the operands is of type `float`, the other one will be converted to a `float`.
- Otherwise, if either of the operands is of type `long`, the other one will be converted to a `long`.
- Otherwise, both operands will be converted to an `int`.

### Casts

In the preceding section, you saw that `int` values are automatically converted to `double` values when necessary. On the other hand, there are obviously times when you want to consider a `double` as an integer. Numeric conversions are possible in Java, but of course information may be lost. Conversions in which loss of information is possible are done by means of *casts*. The syntax for casting is to give the target type in parentheses, followed by the variable name. For example:

```
double x = 9.997;
int nx = (int) x;
```

Then, the variable `nx` has the value 9 because casting a floating-point value to an integer discards the fractional part.

If you want to *round* a floating-point number to the *nearest* integer (which is the more useful operation in most cases), use the `Math.round` method:

```
double x = 9.997;
int nx = (int) Math.round(x);
```

Now the variable `nx` has the value 10. You still need to use the cast `(int)` when you call `round`. The reason is that the return value of the `round` method is a `long`, and a `long` can only be assigned to an `int` with an explicit cast because there is the possibility of information loss.



**CAUTION:** If you try to cast a number of one type to another that is out of the range for the target type, the result will be a truncated number that has a different value. For example, (byte) 300 is actually 44.



**C++ NOTE:** You cannot cast between `boolean` values and any numeric type. This convention prevents common errors. In the rare case that you want to convert a `boolean` value to a number, you can use a conditional expression such as `b ? 1 : 0`.

### Parentheses and Operator Hierarchy

Table 3–4 on the following page shows the precedence of operators. If no parentheses are used, operations are performed in the hierarchical order indicated. Operators on the same level are processed from left to right, except for those that are right associative, as indicated in the table. For example, because `&&` has a higher precedence than `||`, the expression

```
a && b || c
```

means

```
(a && b) || c
```

Because `+=` associates right to left, the expression

```
a += b += c
```

means

```
a += (b += c)
```

That is, the value of `b += c` (which is the value of `b` after the addition) is added to `a`.



**C++ NOTE:** Unlike C or C++, Java does not have a comma operator. However, you can use a *comma-separated list of expressions* in the first and third slot of a `for` statement.

**Table 3–4 Operator Precedence**

Operators	Associativity
[ ] . ( ) (method call)	Left to right
! ~ ++ -- + (unary) - (unary) ( ) (cast) new	Right to left
* / %	Left to right
+ -	Left to right
<< >> >>>	Left to right
< <= > >= instanceof	Left to right
== !=	Left to right
&	Left to right
^	Left to right
	Left to right
&&	Left to right
	Left to right
?:	Right to left
= += -= *= /= %= &=  = ^= <<= >>= >>>=	Right to left

### Enumerated Types

Sometimes, a variable should only hold a restricted set of values. For example, you may sell clothes or pizza in four sizes: small, medium, large, and extra large. Of course, you could encode these sizes as integers 1, 2, 3, 4, or characters S, M, L, and X. But that is an error-prone setup. It is too easy for a variable to hold a wrong value (such as 0 or m).

Starting with Java SE 5.0, you can define your own *enumerated type* whenever such a situation arises. An enumerated type has a finite number of named values. For example:

```
enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };
```

Now you can declare variables of this type:

```
Size s = Size.MEDIUM;
```

A variable of type `Size` can hold only one of the values listed in the type declaration or the special value `null` that indicates that the variable is not set to any value at all.

We discuss enumerated types in greater detail in Chapter 5.

### Strings

Conceptually, Java strings are sequences of Unicode characters. For example, the string `"Java\u2122"` consists of the five Unicode characters J, a, v, a, and <sup>TM</sup>. Java does not have a built-in string type. Instead, the standard Java library contains a predefined class called, naturally enough, `String`. Each quoted string is an instance of the `String` class:

```
String e = ""; // an empty string
String greeting = "Hello";
```

### Substrings

You extract a substring from a larger string with the `substring` method of the `String` class. For example,

```
String greeting = "Hello";
String s = greeting.substring(0, 3);
```

creates a string consisting of the characters "Hel".

The second parameter of `substring` is the first position that you *do not* want to copy. In our case, we want to copy positions 0, 1, and 2 (from position 0 to position 2 inclusive). As `substring` counts it, this means from position 0 inclusive to position 3 *exclusive*.

There is one advantage to the way `substring` works: Computing the length of the substring is easy. The string `s.substring(a, b)` always has length  $b - a$ . For example, the substring "Hel" has length  $3 - 0 = 3$ .

### Concatenation

Java, like most programming languages, allows you to use the `+` sign to join (concatenate) two strings.

```
String expletive = "Expletive";
String PG13 = "deleted";
String message = expletive + PG13;
```

The preceding code sets the variable `message` to the string "Expletivedeleted". (Note the lack of a space between the words: The `+` sign joins two strings in the order received, *exactly* as they are given.)

When you concatenate a string with a value that is not a string, the latter is converted to a string. (As you will see in Chapter 5, every Java object can be converted to a string.) For example,

```
int age = 13;
String rating = "PG" + age;
```

sets `rating` to the string "PG13".

This feature is commonly used in output statements. For example,

```
System.out.println("The answer is " + answer);
```

is perfectly acceptable and will print what one would want (and with the correct spacing because of the space after the word `is`).

### Strings Are Immutable

The `String` class gives no methods that let you *change* a character in an existing string. If you want to turn `greeting` into "Help!", you cannot directly change the last positions of `greeting` into 'p' and '!'. If you are a C programmer, this will make you feel pretty helpless. How are you going to modify the string? In Java, it is quite easy: Concatenate the substring that you want to keep with the characters that you want to replace.

```
greeting = greeting.substring(0, 3) + "p!";
```

This declaration changes the current value of the `greeting` variable to "Help!".

Because you cannot change the individual characters in a Java string, the documentation refers to the objects of the `String` class as being *immutable*. Just as the number 3 is always 3, the string "Hello" will always contain the code unit sequence describing the characters H, e, l, l, o. You cannot change these values. You can, as you just saw however, change the contents of the string *variable* `greeting` and make it refer to a different string, just as you can make a numeric variable currently holding the value 3 hold the value 4.

Isn't that a lot less efficient? It would seem simpler to change the code units than to build up a whole new string from scratch. Well, yes and no. Indeed, it isn't efficient to generate a new string that holds the concatenation of "Hel" and "p!". But immutable strings have one great advantage: the compiler can arrange that strings are *shared*.

To understand how this works, think of the various strings as sitting in a common pool. String variables then point to locations in the pool. If you copy a string variable, both the original and the copy share the same characters.

Overall, the designers of Java decided that the efficiency of sharing outweighs the inefficiency of string editing by extracting substrings and concatenating. Look at your own programs; we suspect that most of the time, you don't change strings—you just compare them. (There is one common exception—assembling strings from individual characters or shorter strings that come from the keyboard or a file. For these situations, Java provides a separate class that we describe in the section "Building Strings" on page 62.)



**C++ NOTE:** C programmers generally are bewildered when they see Java strings for the first time because they think of strings as arrays of characters:

```
char greeting[] = "Hello";
```

That is the wrong analogy: A Java string is roughly analogous to a `char*` pointer,

```
char* greeting = "Hello";
```

When you replace `greeting` with another string, the Java code does roughly the following:

```
char* temp = malloc(6);
strncpy(temp, greeting, 3);
strncpy(temp + 3, "p!", 3);
greeting = temp;
```

Sure, now `greeting` points to the string "Help!". And even the most hardened C programmer must admit that the Java syntax is more pleasant than a sequence of `strncpy` calls. But what if we make another assignment to `greeting`?

```
greeting = "Howdy";
```

Don't we have a memory leak? After all, the original string was allocated on the heap. Fortunately, Java does automatic garbage collection. If a block of memory is no longer needed, it will eventually be recycled.

If you are a C++ programmer and use the `string` class defined by ANSI C++, you will be much more comfortable with the Java `String` type. C++ string objects also perform automatic allocation and deallocation of memory. The memory management is performed explicitly by constructors, assignment operators, and destructors. However, C++ strings are mutable—you can modify individual characters in a string.

**Testing Strings for Equality**

To test whether two strings are equal, use the `equals` method. The expression

```
s.equals(t)
```

returns `true` if the strings `s` and `t` are equal, `false` otherwise. Note that `s` and `t` can be string variables or string constants. For example, the expression

```
"Hello".equals(greeting)
```

is perfectly legal. To test whether two strings are identical except for the upper/lower-case letter distinction, use the `equalsIgnoreCase` method.

```
"Hello".equalsIgnoreCase("hello")
```

Do *not* use the `==` operator to test whether two strings are equal! It only determines whether or not the strings are stored in the same location. Sure, if strings are in the same location, they must be equal. But it is entirely possible to store multiple copies of identical strings in different places.

```
String greeting = "Hello"; //initialize greeting to a string
if (greeting == "Hello") . . .
    // probably true
if (greeting.substring(0, 3) == "Hel") . . .
    // probably false
```

If the virtual machine would always arrange for equal strings to be shared, then you could use the `==` operator for testing equality. But only string *constants* are shared, not strings that are the result of operations like `+` or `substring`. Therefore, *never* use `==` to compare strings lest you end up with a program with the worst kind of bug—an intermittent one that seems to occur randomly.



**C++ NOTE:** If you are used to the C++ string class, you have to be particularly careful about equality testing. The C++ string class does overload the `==` operator to test for equality of the string contents. It is perhaps unfortunate that Java goes out of its way to give strings the same “look and feel” as numeric values but then makes strings behave like pointers for equality testing. The language designers could have redefined `==` for strings, just as they made a special arrangement for `+`. Oh well, every language has its share of inconsistencies.

C programmers never use `==` to compare strings but use `strcmp` instead. The Java method `compareTo` is the exact analog to `strcmp`. You can use

```
if (greeting.compareTo("Hello") == 0) . . .
```

but it seems clearer to use `equals` instead.

**Code Points and Code Units**

Java strings are implemented as sequences of `char` values. As we discussed in the section “The `char` Type” on page 42, the `char` data type is a code unit for representing Unicode code points in the UTF-16 encoding. The most commonly used Unicode characters can be represented with a single code unit. The supplementary characters require a pair of code units.



The `length` method yields the number of code units required for a given string in the UTF-16 encoding. For example:

```
String greeting = "Hello";
int n = greeting.length(); // is 5.
```

To get the true length, that is, the number of code points, call

```
int cpCount = greeting.codePointCount(0, greeting.length());
```

The call `s.charAt(n)` returns the code unit at position `n`, where `n` is between 0 and `s.length() - 1`.

For example:

```
char first = greeting.charAt(0); // first is 'H'
char last = greeting.charAt(4); // last is 'o'
```

To get at the `i`th code point, use the statements

```
int index = greeting.offsetByCodePoints(0, i);
int cp = greeting.codePointAt(index);
```



**NOTE:** Java counts the code units in strings in a peculiar fashion: the first code unit in a string has position 0. This convention originated in C, where there was a technical reason for counting positions starting at 0. That reason has long gone away and only the nuisance remains. However, so many programmers are used to this convention that the Java designers decided to keep it.

Why are we making a fuss about code units? Consider the sentence

$\mathbb{Z}$  is the set of integers

The  $\mathbb{Z}$  character requires two code units in the UTF-16 encoding. Calling

```
char ch = sentence.charAt(1)
```

doesn't return a space but the second code unit of  $\mathbb{Z}$ . To avoid this problem, you should not use the `char` type. It is too low-level.

If your code traverses a string, and you want to look at each code point in turn, use these statements:

```
int cp = sentence.codePointAt(i);
if (Character.isSupplementaryCodePoint(cp)) i += 2;
else i++;
```

Fortunately, the `codePointAt` method can tell whether a code unit is the first or second half of a supplementary character, and it returns the right result either way. That is, you can move backwards with the following statements:

```
i--;
int cp = sentence.codePointAt(i);
if (Character.isSupplementaryCodePoint(cp)) i--;
```

### The String API

The `String` class in Java contains more than 50 methods. A surprisingly large number of them are sufficiently useful so that we can imagine using them frequently. The following API note summarizes the ones we found most useful.



NOTE: You will find these API notes throughout the book to help you understand the Java Application Programming Interface (API). Each API note starts with the name of a class such as `java.lang.String`—the significance of the so-called *package* name `java.lang` is explained in Chapter 4. The class name is followed by the names, explanations, and parameter descriptions of one or more methods.

We typically do not list all methods of a particular class but instead select those that are most commonly used, and describe them in a concise form. For a full listing, consult the on-line documentation (see “Reading the On-Line API Documentation” on page 59).

We also list the version number in which a particular class was introduced. If a method has been added later, it has a separate version number.



#### **API** `java.lang.String` 1.0

- `char charAt(int index)`  
returns the code unit at the specified location. You probably don’t want to call this method unless you are interested in low-level code units.
- `int codePointAt(int index)` 5.0  
returns the code point that starts or ends at the specified location.
- `int offsetByCodePoints(int startIndex, int cpCount)` 5.0  
returns the index of the code point that is `cpCount` code points away from the code point at `startIndex`.
- `int compareTo(String other)`  
returns a negative value if the string comes before `other` in dictionary order, a positive value if the string comes after `other` in dictionary order, or 0 if the strings are equal.
- `boolean endsWith(String suffix)`  
returns true if the string ends with `suffix`.
- `boolean equals(Object other)`  
returns true if the string equals `other`.
- `boolean equalsIgnoreCase(String other)`  
returns true if the string equals `other`, except for upper/lowercase distinction.
- `int indexOf(String str)`
- `int indexOf(String str, int fromIndex)`
- `int indexOf(int cp)`
- `int indexOf(int cp, int fromIndex)`  
returns the start of the first substring equal to the string `str` or the code point `cp`, starting at index 0 or at `fromIndex`, or -1 if `str` does not occur in this string.
- `int lastIndexOf(String str)`
- `int lastIndexOf(String str, int fromIndex)`
- `int lastIndexOf(int cp)`
- `int lastIndexOf(int cp, int fromIndex)`  
returns the start of the last substring equal to the string `str` or the code point `cp`, starting at the end of the string or at `fromIndex`.

- `int length()`  
returns the length of the string.
- `int codePointCount(int startIndex, int endIndex)` **5.0**  
returns the number of code points between `startIndex` and `endIndex - 1`. Unpaired surrogates are counted as code points.
- `String replace(CharSequence oldString, CharSequence newString)`  
returns a new string that is obtained by replacing all substrings matching `oldString` in the string with the string `newString`. You can supply `String` or `StringBuilder` objects for the `CharSequence` parameters.
- `boolean startsWith(String prefix)`  
returns true if the string begins with prefix.
- `String substring(int beginIndex)`
- `String substring(int beginIndex, int endIndex)`  
returns a new string consisting of all code units from `beginIndex` until the end of the string or until `endIndex - 1`.
- `String toLowerCase()`  
returns a new string containing all characters in the original string, with uppercase characters converted to lowercase.
- `String toUpperCase()`  
returns a new string containing all characters in the original string, with lowercase characters converted to uppercase.
- `String trim()`  
returns a new string by eliminating all leading and trailing spaces in the original string.

### **Reading the On-Line API Documentation**

As you just saw, the `String` class has lots of methods. Furthermore, there are thousands of classes in the standard libraries, with many more methods. It is plainly impossible to remember all useful classes and methods. Therefore, it is essential that you become familiar with the on-line API documentation that lets you look up all classes and methods in the standard library. The API documentation is part of the JDK. It is in HTML format. Point your web browser to the `docs/api/index.html` subdirectory of your JDK installation. You will see a screen like that in Figure 3-2.

The screen is organized into three frames. A small frame on the top left shows all available packages. Below it, a larger frame lists all classes. Click on any class name, and the API documentation for the class is displayed in the large frame to the right (see Figure 3-3). For example, to get more information on the methods of the `String` class, scroll the second frame until you see the `String` link, then click on it.

Then scroll the frame on the right until you reach a summary of all methods, sorted in alphabetical order (see Figure 3-4). Click on any method name for a detailed description of that method (see Figure 3-5). For example, if you click on the `compareToIgnoreCase` link, you get the description of the `compareToIgnoreCase` method.

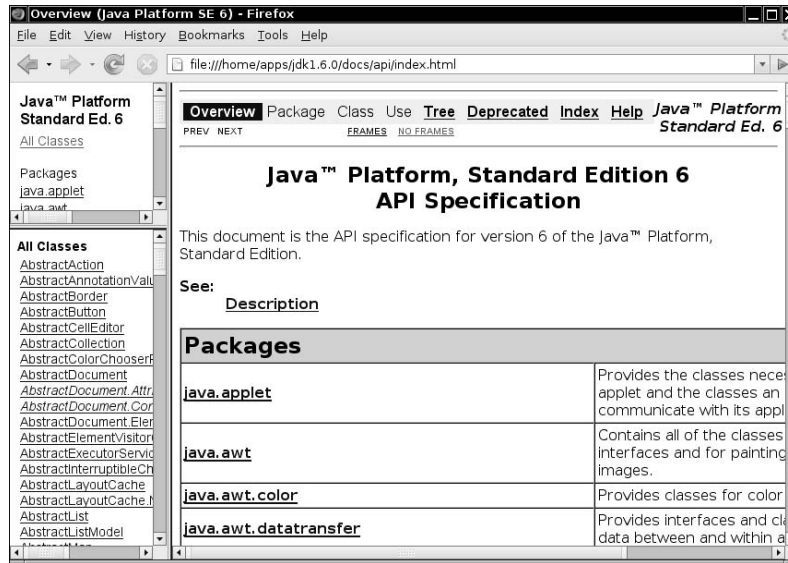


Figure 3-2 The three panes of the API documentation



Figure 3-3 Class description for the String class

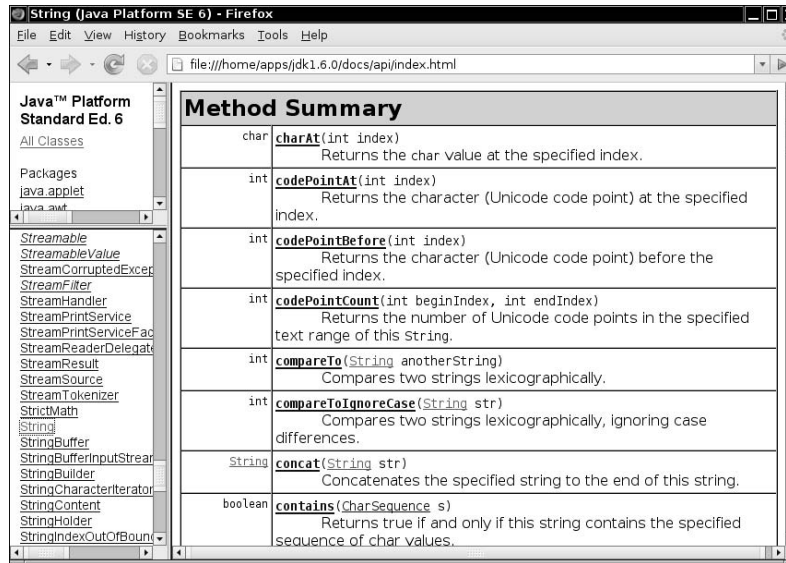


Figure 3-4 Method summary of the String class

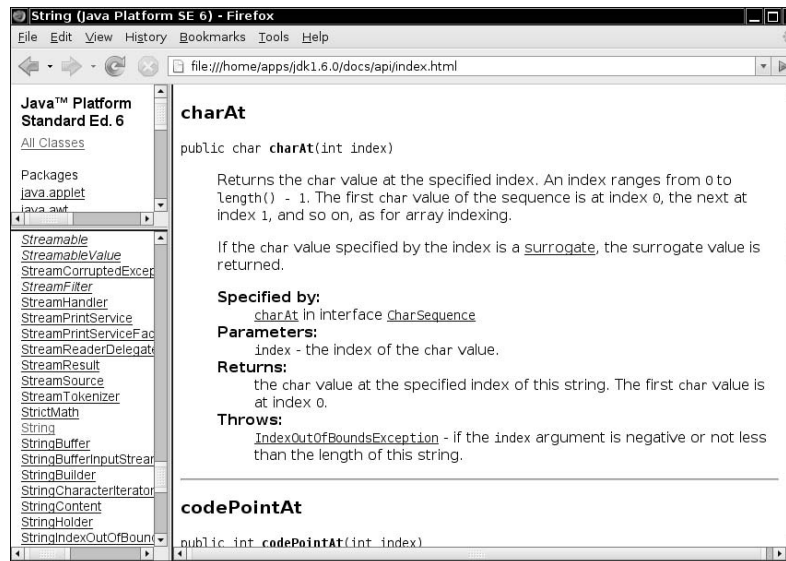


Figure 3-5 Detailed description of a String method



TIP: Bookmark the `docs/api/index.html` page in your browser right now.

### Building Strings

Occasionally, you need to build up strings from shorter strings, such as keystrokes or words from a file. It would be inefficient to use string concatenation for this purpose. Every time you concatenate strings, a new `String` object is constructed. This is time consuming and it wastes memory. Using the `StringBuilder` class avoids this problem.

Follow these steps if you need to build a string from many small pieces. First, construct an empty string builder:

```
StringBuilder builder = new StringBuilder();
```

(We discuss constructors and the `new` operator in detail in Chapter 4.)

Each time you need to add another part, call the `append` method.

```
builder.append(ch); // appends a single character
builder.append(str); // appends a string
```

When you are done building the string, call the `toString` method. You will get a `String` object with the character sequence contained in the builder.

```
String completedString = builder.toString();
```



NOTE: The `StringBuilder` class was introduced in JDK 5.0. Its predecessor, `StringBuffer`, is slightly less efficient, but it allows multiple threads to add or remove characters. If all string editing happens in a single thread (which is usually the case), you should use `StringBuilder` instead. The APIs of both classes are identical.

The following API notes contain the most important methods for the `StringBuilder` class.

#### API `java.lang.StringBuilder` 5.0

- `StringBuilder()`  
constructs an empty string builder.
- `int length()`  
returns the number of code units of the builder or buffer.
- `StringBuilder append(String str)`  
appends a string and returns this.
- `StringBuilder append(char c)`  
appends a code unit and returns this.
- `StringBuilder appendCodePoint(int cp)`  
appends a code point, converting it into one or two code units, and returns this.
- `void setCharAt(int i, char c)`  
sets the *i*th code unit to *c*.
- `StringBuilder insert(int offset, String str)`  
inserts a string at position *offset* and returns this.

- `StringBuilder insert(int offset, char c)`  
inserts a code unit at position `offset` and returns this.
- `StringBuilder delete(int startIndex, int endIndex)`  
deletes the code units with offsets `startIndex` to `endIndex - 1` and returns this.
- `String toString()`  
returns a string with the same data as the builder or buffer contents.

## Input and Output

To make our example programs more interesting, we want to accept input and properly format the program output. Of course, modern programs use a GUI for collecting user input. However, programming such an interface requires more tools and techniques than we have at our disposal at this time. Because the first order of business is to become more familiar with the Java programming language, we make do with the humble console for input and output for now. GUI programming is covered in Chapters 7 through 9.

### Reading Input

You saw that it is easy to print output to the “standard output stream” (that is, the console window) just by calling `System.out.println`. Reading from the “standard input stream” `System.in` isn’t quite as simple. To read console input, you first construct a `Scanner` that is attached to `System.in`:

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

(We discuss constructors and the `new` operator in detail in Chapter 4.)

Now you use the various methods of the `Scanner` class to read input. For example, the `nextLine` method reads a line of input.

```
System.out.print("What is your name? ");  
String name = in.nextLine();
```

Here, we use the `nextLine` method because the input might contain spaces. To read a single word (delimited by whitespace), call

```
String firstName = in.next();
```

To read an integer, use the `nextInt` method.

```
System.out.print("How old are you? ");  
int age = in.nextInt();
```

Similarly, the `nextDouble` method reads the next floating-point number.

The program in Listing 3–2 asks for the user’s name and age and then prints a message like

```
Hello, Cay. Next year, you'll be 46
```

Finally, note the line

```
import java.util.*;
```

at the beginning of the program. The `Scanner` class is defined in the `java.util` package. Whenever you use a class that is not defined in the basic `java.lang` package, you need to use an `import` directive. We look at packages and `import` directives in more detail in Chapter 4.

**Listing 3-2** InputTest.java

```

1. import java.util.*;
2.
3. /**
4.  * This program demonstrates console input.
5.  * @version 1.10 2004-02-10
6.  * @author Cay Horstmann
7.  */
8. public class InputTest
9. {
10.     public static void main(String[] args)
11.     {
12.         Scanner in = new Scanner(System.in);
13.
14.         // get first input
15.         System.out.print("What is your name? ");
16.         String name = in.nextLine();
17.
18.         // get second input
19.         System.out.print("How old are you? ");
20.         int age = in.nextInt();
21.
22.         // display output on console
23.         System.out.println("Hello, " + name + ". Next year, you'll be " + (age + 1));
24.     }
25. }

```



**NOTE:** The `Scanner` class is not suitable for reading a password from a console since the input is plainly visible to anyone. Java SE 6 introduces a `Console` class specifically for this purpose. To read a password, use the following code:

```

Console cons = System.console();
String username = cons.readLine("User name: ");
char[] passwd = cons.readPassword("Password: ");

```

For security reasons, the password is returned in an array of characters rather than a string. After you are done processing the password, you should immediately overwrite the array elements with a filler value. (Array processing is discussed later in this chapter.)

Input processing with a `Console` object is not as convenient as with a `Scanner`. You can only read a line of input at a time. There are no methods for reading individual words or numbers.

**API** `java.util.Scanner` 5.0

- `Scanner(InputStream in)`  
constructs a `Scanner` object from the given input stream.
- `String nextLine()`  
reads the next line of input.



- `String next()`  
reads the next word of input (delimited by whitespace).
- `int nextInt()`
- `double nextDouble()`  
reads and converts the next character sequence that represents an integer or floating-point number.
- `boolean hasNext()`  
tests whether there is another word in the input.
- `boolean hasNextInt()`
- `boolean hasNextDouble()`  
tests whether the next character sequence represents an integer or floating-point number.

**API** `java.lang.System` 1.0

- `static Console console()` 6  
returns a `Console` object for interacting with the user through a console window if such an interaction is possible, `null` otherwise. A `Console` object is available for any program that is launched in a console window. Otherwise, the availability is system-dependent.

**API** `java.io.Console` 6

- `static char[] readPassword(String prompt, Object... args)`
- `static String readLine(String prompt, Object... args)`  
displays the prompt and reads the user input until the end of the input line. The `args` parameters can be used to supply formatting arguments, as described in the next section.

**Formatting Output**

You can print a number `x` to the console with the statement `System.out.print(x)`. That command will print `x` with the maximum number of non-zero digits for that type. For example,

```
double x = 10000.0 / 3.0;
System.out.print(x);
```

prints

```
3333.3333333333335
```

That is a problem if you want to display, for example, dollars and cents.

In early versions of Java, formatting numbers was a bit of a hassle. Fortunately, Java SE 5.0 brought back the venerable `printf` method from the C library. For example, the call

```
System.out.printf("%8.2f", x);
```

prints `x` with a *field width* of 8 characters and a *precision* of 2 characters. That is, the print-out contains a leading space and the seven characters

```
3333.33
```

You can supply multiple parameters to `printf`. For example:

```
System.out.printf("Hello, %s. Next year, you'll be %d", name, age);
```

Each of the *format specifiers* that start with a % character is replaced with the corresponding argument. The *conversion character* that ends a format specifier indicates the type of the value to be formatted: *f* is a floating-point number, *s* a string, and *d* a decimal integer. Table 3–5 shows all conversion characters.

**Table 3–5 Conversions for printf**

Conversion Character	Type	Example
d	Decimal integer	159
x	Hexadecimal integer	9f
o	Octal integer	237
f	Fixed-point floating-point	15.9
e	Exponential floating-point	1.59e+01
g	General floating-point (the shorter of e and f)	—
a	Hexadecimal floating-point	0x1.fccdp3
s	String	Hello
c	Character	H
b	boolean	true
h	Hash code	42628b2
tx	Date and time	See Table 3–7
%	The percent symbol	%
n	The platform-dependent line separator	—

In addition, you can specify *flags* that control the appearance of the formatted output. Table 3–6 shows all flags. For example, the comma flag adds group separators. That is,

```
System.out.printf("%.2f", 10000.0 / 3.0);
```

prints

```
3,333.33
```

You can use multiple flags, for example, "%.2f", to use group separators and enclose negative numbers in parentheses.



**NOTE:** You can use the *s* conversion to format arbitrary objects. If an arbitrary object implements the `Formattable` interface, the object's `formatTo` method is invoked. Otherwise, the `toString` method is invoked to turn the object into a string. We discuss the `toString` method in Chapter 5 and interfaces in Chapter 6.

**Table 3-6** Flags for printf

Flag	Purpose	Example
+	Prints sign for positive and negative numbers	+3333.33
space	Adds a space before positive numbers	3333.33
0	Adds leading zeroes	003333.33
-	Left-justifies field	3333.33
(	Encloses negative number in parentheses	(3333.33)
,	Adds group separators	3,333.33
# (for f format)	Always includes a decimal point	3,333.
# (for x or o format)	Adds 0x or 0 prefix	0xcaffe
\$	Specifies the index of the argument to be formatted; for example, %1\$d %1\$x prints the first argument in decimal and hexadecimal	159 9F
<	Formats the same value as the previous specification; for example, %d %<x prints the same number in decimal and hexadecimal	159 9F

You can use the static `String.format` method to create a formatted string without printing it:

```
String message = String.format("Hello, %s. Next year, you'll be %d", name, age);
```

Although we do not describe the `Date` type in detail until Chapter 4, we do, in the interest of completeness, briefly discuss the date and time formatting options of the `printf` method. You use a two-letter format, starting with `t` and ending in one of the letters of Table 3-7. For example,

```
System.out.printf("%tc", new Date());
```

prints the current date and time in the format

```
Mon Feb 09 18:05:19 PST 2004
```

**Table 3-7** Date and Time Conversion Characters

Conversion Character	Type	Example
c	Complete date and time	Mon Feb 09 18:05:19 PST 2004
F	ISO 8601 date	2004-02-09
D	U.S. formatted date (month/day/year)	02/09/2004
T	24-hour time	18:05:19

**Table 3-7 Date and Time Conversion Characters (continued)**

<b>Conversion Character</b>	<b>Type</b>	<b>Example</b>
r	12-hour time	06:05:19 pm
R	24-hour time, no seconds	18:05
Y	Four-digit year (with leading zeroes)	2004
y	Last two digits of the year (with leading zeroes)	04
C	First two digits of the year (with leading zeroes)	20
B	Full month name	February
b or h	Abbreviated month name	Feb
m	Two-digit month (with leading zeroes)	02
d	Two-digit day (with leading zeroes)	09
e	Two-digit day (without leading zeroes)	9
A	Full weekday name	Monday
a	Abbreviated weekday name	Mon
j	Three-digit day of year (with leading zeroes), between 001 and 366	069
H	Two-digit hour (with leading zeroes), between 00 and 23	18
k	Two-digit hour (without leading zeroes), between 0 and 23	18
I	Two-digit hour (with leading zeroes), between 01 and 12	06
l	Two-digit hour (without leading zeroes), between 1 and 12	6
M	Two-digit minutes (with leading zeroes)	05
S	Two-digit seconds (with leading zeroes)	19
L	Three-digit milliseconds (with leading zeroes)	047
N	Nine-digit nanoseconds (with leading zeroes)	047000000
P	Uppercase morning or afternoon marker	PM
p	Lowercase morning or afternoon marker	pm
z	RFC 822 numeric offset from GMT	-0800
Z	Time zone	PST
s	Seconds since 1970-01-01 00:00:00 GMT	1078884319
Q	Milliseconds since 1970-01-01 00:00:00 GMT	1078884319047

As you can see in Table 3–7, some of the formats yield only a part of a given date, for example, just the day or just the month. It would be a bit silly if you had to supply the date multiple times to format each part. For that reason, a format string can indicate the *index* of the argument to be formatted. The index must immediately follow the %, and it must be terminated by a \$. For example,

```
System.out.printf("%1$s %2$tB %2$te, %2$tY", "Due date:", new Date());
```

prints

```
Due date: February 9, 2004
```

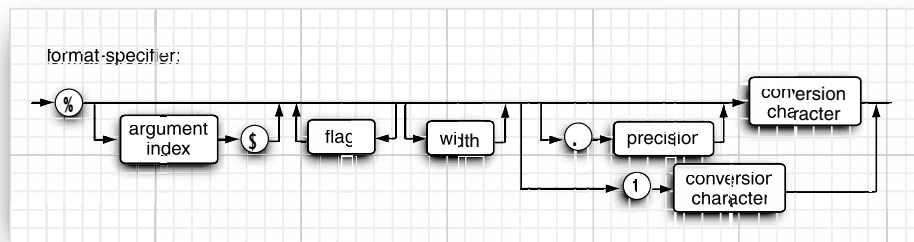
Alternatively, you can use the < flag. It indicates that the same argument as in the preceding format specification should be used again. That is, the statement

```
System.out.printf("%s %tB %<te, %<tY", "Due date:", new Date());
```

yields the same output as the preceding statement.

**X** CAUTION: Argument index values start with 1, not with 0: %1\$. . . formats the first argument. This avoids confusion with the 0 flag.

You have now seen all features of the printf method. Figure 3–6 shows a syntax diagram for format specifiers.



**Figure 3–6** Format specifier syntax

**✓** NOTE: A number of the formatting rules are *locale specific*. For example, in Germany, the decimal separator is a period, not a comma, and Monday is formatted as Montag. You will see in Volume II how to control the international behavior of your applications.

### File Input and Output

To read from a file, construct a Scanner object from a File object, like this:

```
Scanner in = new Scanner(new File("myfile.txt"));
```

If the file name contains backslashes, remember to escape each of them with an additional backslash: "c:\mydirectory\myfile.txt".

Now you can read from the file, using any of the `Scanner` methods that we already described.

To write to a file, construct a `PrintWriter` object. In the constructor, simply supply the file name:

```
PrintWriter out = new PrintWriter("myfile.txt");
```

If the file does not exist, you can simply use the `print`, `println`, and `printf` commands as you did when printing to `System.out`.



**CAUTION:** You can construct a `Scanner` with a string parameter, but the scanner interprets the string as data, not a file name. For example, if you call

```
Scanner in = new Scanner("myfile.txt"); // ERROR?
```

then the scanner will see ten characters of data: 'm', 'y', 'f', and so on. That is probably not what was intended in this case.



**NOTE:** When you specify a relative file name, such as `"myfile.txt"`, `"mydirectory/myfile.txt"`, or `"../myfile.txt"`, the file is located relative to the directory in which the Java virtual machine was started. If you launched your program from a command shell, by executing

```
java MyProg
```

then the starting directory is the current directory of the command shell. However, if you use an integrated development environment, the starting directory is controlled by the IDE. You can find the directory location with this call:

```
String dir = System.getProperty("user.dir");
```

If you run into grief with locating files, consider using absolute path names such as `"c:\\mydirectory\\myfile.txt"` or `"/home/me/mydirectory/myfile.txt"`.

As you just saw, you can access files just as easily as you can use `System.in` and `System.out`. There is just one catch: If you construct a `Scanner` with a file that does not exist or a `PrintWriter` with a file name that cannot be created, an exception occurs. The Java compiler considers these exceptions to be more serious than a “divide by zero” exception, for example. In Chapter 11, you will learn various ways for handling exceptions. For now, you should simply tell the compiler that you are aware of the possibility of a “file not found” exception. You do this by tagging the `main` method with a `throws` clause, like this:

```
public static void main(String[] args) throws FileNotFoundException
{
    Scanner in = new Scanner(new File("myfile.txt"));
    . . .
}
```

You have now seen how to read and write files that contain textual data. For more advanced topics, such as dealing with different character encodings, processing binary data, reading directories, and writing zip files, please turn to Chapter 1 of Volume II.



NOTE: When you launch a program from a command shell, you can use the redirection syntax of your shell and attach any file to `System.in` and `System.out`:

```
java MyProg < myfile.txt > output.txt
```

Then, you need not worry about handling the `FileNotFoundException`.

#### API `java.util.Scanner` 5.0

- `Scanner(File f)`  
constructs a `Scanner` that reads data from the given file.
- `Scanner(String data)`  
constructs a `Scanner` that reads data from the given string.

#### API `java.io.PrintWriter` 1.1

- `PrintWriter(File f)`  
constructs a `PrintWriter` that writes data to the given file.
- `PrintWriter(String fileName)`  
constructs a `PrintWriter` that writes data to the file with the given file name.

#### API `java.io.File` 1.0

- `File(String fileName)`  
constructs a `File` object that describes a file with the given name. Note that the file need not currently exist.

### Control Flow

Java, like any programming language, supports both conditional statements and loops to determine control flow. We start with the conditional statements and then move on to loops. We end with the somewhat cumbersome `switch` statement that you can use when you have to test for many values of a single expression.



C++ NOTE: The Java control flow constructs are identical to those in C and C++, with a few exceptions. There is no `goto`, but there is a “labeled” version of `break` that you can use to break out of a nested loop (where you perhaps would have used a `goto` in C). Finally! Java SE 5.0 added a variant of the `for` loop that has no analog in C or C++. It is similar to the `foreach` loop in C#.

### Block Scope

Before we get into the actual control structures, you need to know more about *blocks*.

A block or compound statement is any number of simple Java statements that are surrounded by a pair of braces. Blocks define the scope of your variables. Blocks can be *nested* inside another block. Here is a block that is nested inside the block of the `main` method.

```

public static void main(String[] args)
{
    int n;
    . . .
    {
        int k;
        . . .
    } // k is only defined up to here
}

```

However, you may not declare identically named variables in two nested blocks. For example, the following is an error and will not compile:

```

public static void main(String[] args)
{
    int n;
    . . .
    {
        int k;
        int n; // error--can't redefine n in inner block
        . . .
    }
}

```



**C++ NOTE:** In C++, it is possible to redefine a variable inside a nested block. The inner definition then shadows the outer one. This can be a source of programming errors; hence, Java does not allow it.

### Conditional Statements

The conditional statement in Java has the form

```
if (condition) statement
```

The condition must be surrounded by parentheses.

In Java, as in most programming languages, you will often want to execute multiple statements when a single condition is true. In this case, you use a *block statement* that takes the form

```

{
    statement1
    statement2
    . . .
}

```

For example:

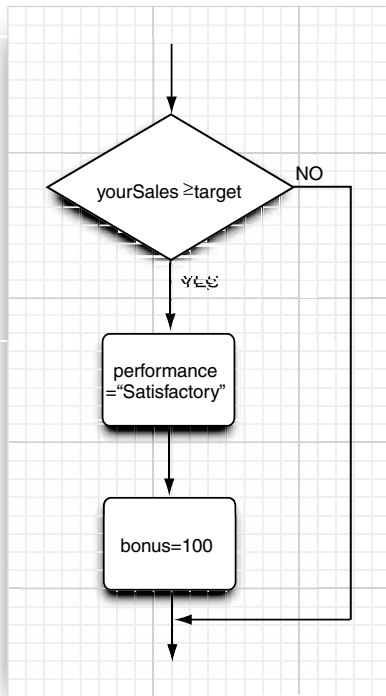
```

if (yourSales >= target)
{
    performance = "Satisfactory";
    bonus = 100;
}

```

In this code all the statements surrounded by the braces will be executed when `yourSales` is greater than or equal to `target` (see Figure 3–7).





**Figure 3-7** Flowchart for the if statement

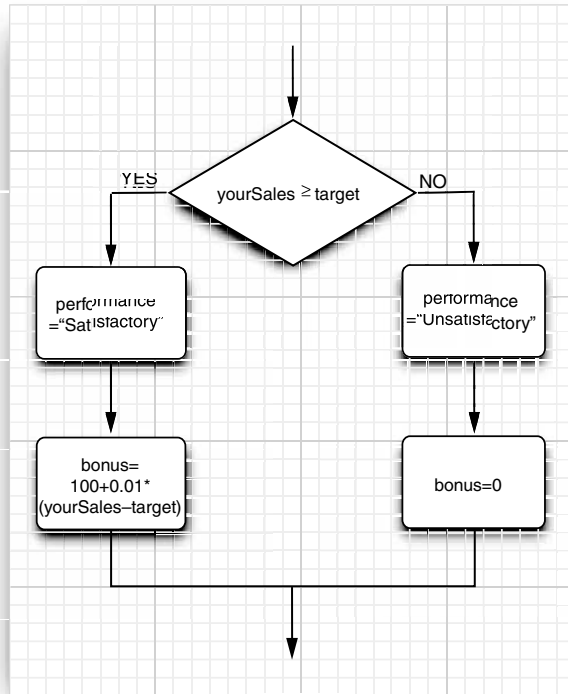
NOTE: A block (sometimes called a *compound statement*) allows you to have more than one (simple) statement in any Java programming structure that might otherwise have a single (simple) statement.

The more general conditional in Java looks like this (see Figure 3-8):

```
if (condition) statement1 else statement2
```

For example:

```
if (yourSales >= target)
{
    performance = "Satisfactory";
    bonus = 100 + 0.01 * (yourSales - target);
}
else
{
    performance = "Unsatisfactory";
    bonus = 0;
}
```



**Figure 3-8** Flowchart for the if/else statement

The else part is always optional. An else groups with the closest if. Thus, in the statement

```
if (x <= 0) if (x == 0) sign = 0; else sign = -1;
```

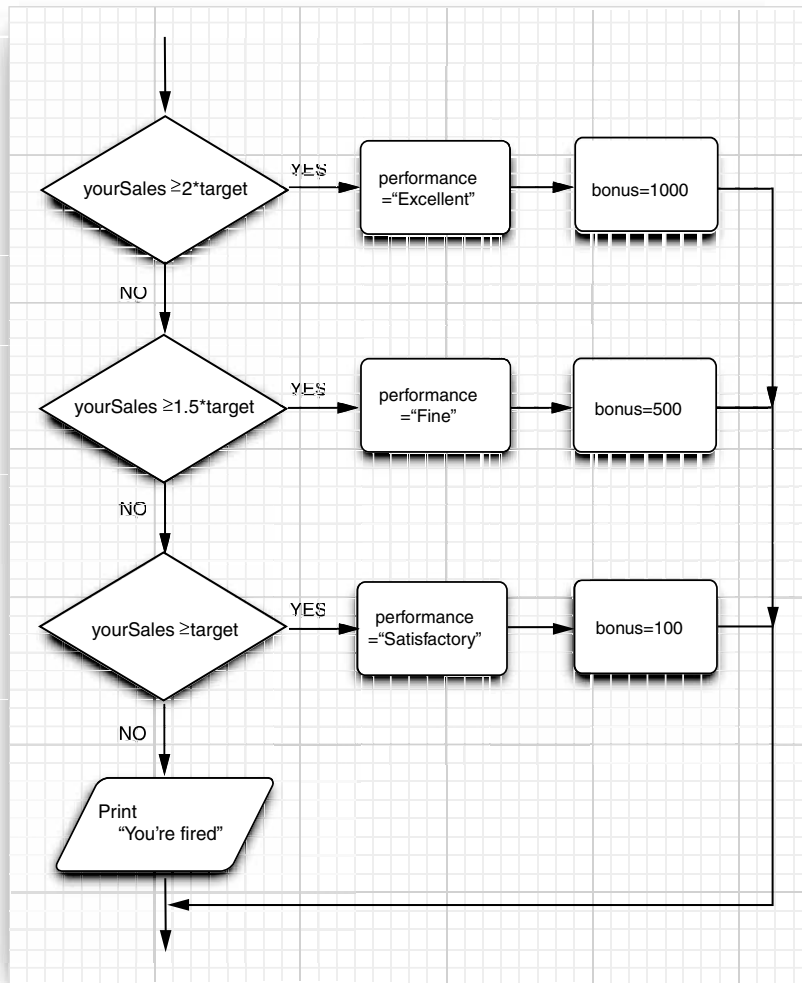
the else belongs to the second if. Of course, it is a good idea to use braces to clarify this code:

```
if (x <= 0) { if (x == 0) sign = 0; else sign = -1; }
```

Repeated if . . . else if . . . alternatives are common (see Figure 3-9). For example:

```
if (yourSales >= 2 * target)
{
    performance = "Excellent";
    bonus = 1000;
}
else if (yourSales >= 1.5 * target)
{
    performance = "Fine";
    bonus = 500;
}
else if (yourSales >= target)
```

```
{
  performance = "Satisfactory";
  bonus = 100;
}
else
{
  System.out.println("You're fired");
}
```



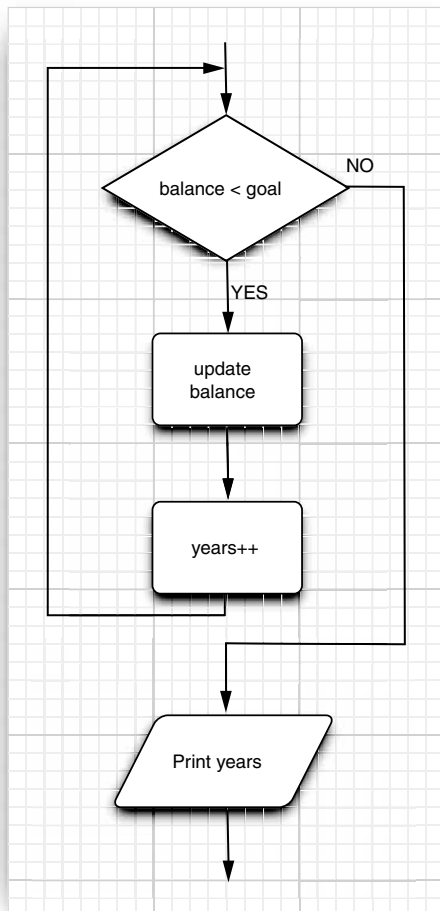
**Figure 3-9** Flowchart for the if/else if (multiple branches)

**Loops**

The `while` loop executes a statement (which may be a block statement) while a condition is true. The general form is

```
while (condition) statement
```

The `while` loop will never execute if the condition is false at the outset (see Figure 3–10).



**Figure 3–10** Flowchart for the `while` statement

The program in Listing 3–3 determines how long it will take to save a specific amount of money for your well-earned retirement, assuming that you deposit the same amount of money per year and that the money earns a specified interest rate.

In the example, we are incrementing a counter and updating the amount currently accumulated in the body of the loop until the total exceeds the targeted amount.

```
while (balance < goal)
{
    balance += payment;
    double interest = balance * interestRate / 100;
    balance += interest;
    years++;
}
System.out.println(years + " years.");
```

(Don't rely on this program to plan for your retirement. We left out a few niceties such as inflation and your life expectancy.)

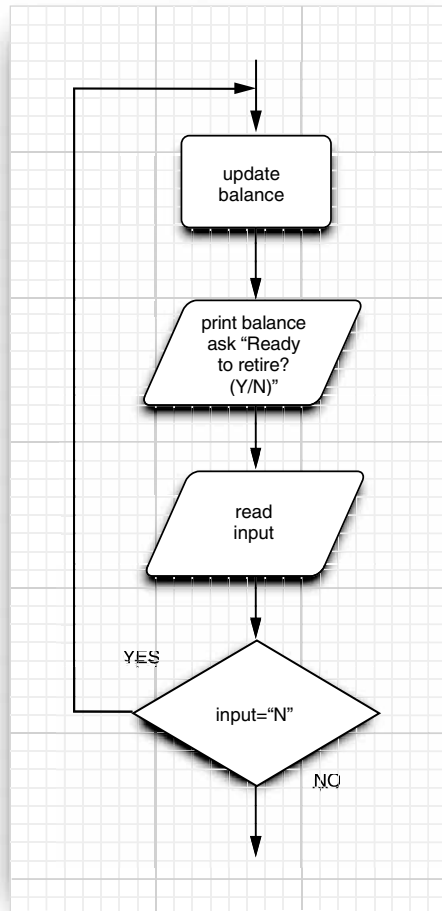
A `while` loop tests at the top. Therefore, the code in the block may never be executed. If you want to make sure a block is executed at least once, you will need to move the test to the bottom. You do that with the `do/while` loop. Its syntax looks like this:

```
do statement while (condition);
```

This loop executes the statement (which is typically a block) and only then tests the condition. It then repeats the statement and retests the condition, and so on. The code in Listing 3–4 computes the new balance in your retirement account and then asks if you are ready to retire:

```
do
{
    balance += payment;
    double interest = balance * interestRate / 100;
    balance += interest;
    year++;
    // print current balance
    . . .
    // ask if ready to retire and get input
    . . .
}
while (input.equals("N"));
```

As long as the user answers "N", the loop is repeated (see Figure 3–11). This program is a good example of a loop that needs to be entered at least once, because the user needs to see the balance before deciding whether it is sufficient for retirement.

**Figure 3-11** Flowchart for the do/while statement**Listing 3-3** Retirement.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates a <code>while</code> loop.
5.  * @version 1.20 2004-02-10
6.  * @author Cay Horstmann
7.  */
```

**Listing 3-3** Retirement.java (continued)

```
8. public class Retirement
9. {
10.     public static void main(String[] args)
11.     {
12.         // read inputs
13.         Scanner in = new Scanner(System.in);
14.
15.         System.out.print("How much money do you need to retire? ");
16.         double goal = in.nextDouble();
17.
18.         System.out.print("How much money will you contribute every year? ");
19.         double payment = in.nextDouble();
20.
21.         System.out.print("Interest rate in %: ");
22.         double interestRate = in.nextDouble();
23.
24.         double balance = 0;
25.         int years = 0;
26.
27.         // update account balance while goal isn't reached
28.         while (balance < goal)
29.         {
30.             // add this year's payment and interest
31.             balance += payment;
32.             double interest = balance * interestRate / 100;
33.             balance += interest;
34.             years++;
35.         }
36.
37.         System.out.println("You can retire in " + years + " years.");
38.     }
39. }
```

**Listing 3-4** Retirement2.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates a <code>do/while</code> loop.
5.  * @version 1.20 2004-02-10
6.  * @author Cay Horstmann
7.  */
8. public class Retirement2
9. {
```

**Listing 3-4** Retirement2.java (continued)

```
10. public static void main(String[] args)
11. {
12.     Scanner in = new Scanner(System.in);
13.
14.     System.out.print("How much money will you contribute every year? ");
15.     double payment = in.nextDouble();
16.
17.     System.out.print("Interest rate in %: ");
18.     double interestRate = in.nextDouble();
19.
20.     double balance = 0;
21.     int year = 0;
22.
23.     String input;
24.
25.     // update account balance while user isn't ready to retire
26.     do
27.     {
28.         // add this year's payment and interest
29.         balance += payment;
30.         double interest = balance * interestRate / 100;
31.         balance += interest;
32.
33.         year++;
34.
35.         // print current balance
36.         System.out.printf("After year %d, your balance is %, .2f%n", year, balance);
37.
38.         // ask if ready to retire and get input
39.         System.out.print("Ready to retire? (Y/N) ");
40.         input = in.next();
41.     }
42.     while (input.equals("N"));
43. }
44. }
```

### ***Determinate Loops***

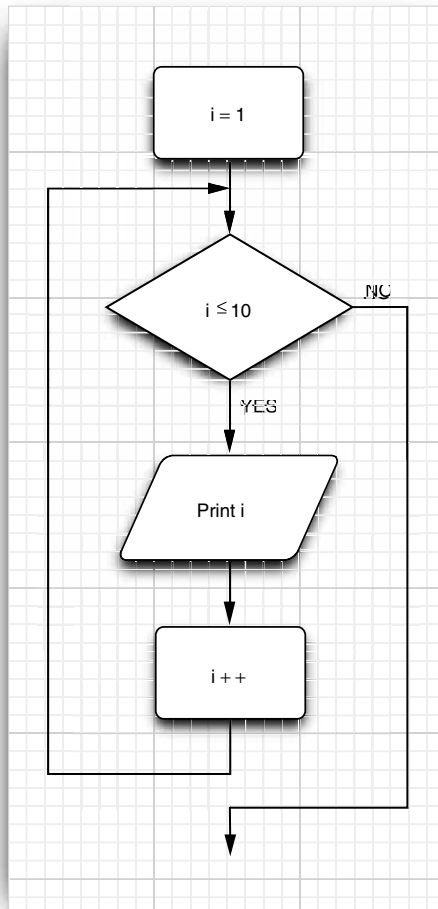
The for loop is a general construct to support iteration that is controlled by a counter or similar variable that is updated after every iteration. As Figure 3-12 shows, the following loop prints the numbers from 1 to 10 on the screen.

```
for (int i = 1; i <= 10; i++)
    System.out.println(i);
```

The first slot of the for statement usually holds the counter initialization. The second slot gives the condition that will be tested before each new pass through the loop, and the third slot explains how to update the counter.



Although Java, like C++, allows almost any expression in the various slots of a `for` loop, it is an unwritten rule of good taste that the three slots of a `for` statement should only initialize, test, and update the same counter variable. One can write very obscure loops by disregarding this rule.



**Figure 3-12** Flowchart for the `for` statement

Even within the bounds of good taste, much is possible. For example, you can have loops that count down:

```
for (int i = 10; i > 0; i--)  
    System.out.println("Counting down . . . " + i);  
System.out.println("Blastoff!");
```



**CAUTION:** Be careful about testing for equality of floating-point numbers in loops. A for loop that looks like

```
for (double x = 0; x != 10; x += 0.1) . . .
```

may never end. Because of roundoff errors, the final value may not be reached exactly. For example, in the loop above, x jumps from 9.99999999999998 to 10.099999999999998 because there is no exact binary representation for 0.1.

When you declare a variable in the first slot of the for statement, the scope of that variable extends until the end of the body of the for loop.

```
for (int i = 1; i <= 10; i++)
{
    . . .
}
// i no longer defined here
```

In particular, if you define a variable inside a for statement, you cannot use the value of that variable outside the loop. Therefore, if you wish to use the final value of a loop counter outside the for loop, be sure to declare it outside the loop header!

```
int i;
for (i = 1; i <= 10; i++)
{
    . . .
}
// i still defined here
```

On the other hand, you can define variables with the same name in separate for loops:

```
for (int i = 1; i <= 10; i++)
{
    . . .
}
. . .
for (int i = 11; i <= 20; i++) // ok to define another variable named i
{
    . . .
}
```

A for loop is merely a convenient shortcut for a while loop. For example,

```
for (int i = 10; i > 0; i--)
    System.out.println("Counting down . . . " + i);
```

can be rewritten as

```
int i = 10;
while (i > 0)
{
    System.out.println("Counting down . . . " + i);
    i--;
}
```

Listing 3-5 shows a typical example of a for loop.


The program computes the odds on winning a lottery. For example, if you must pick 6 numbers from the numbers 1 to 50 to win, then there are  $(50 \times 49 \times 48 \times 47 \times 46 \times 45) / (1 \times 2 \times 3 \times 4 \times 5 \times 6)$  possible outcomes, so your chance is 1 in 15,890,700. Good luck!

In general, if you pick  $k$  numbers out of  $n$ , there are

$$\frac{n \times (n-1) \times (n-2) \times \dots \times (n-k+1)}{1 \times 2 \times 3 \times \dots \times k}$$

possible outcomes. The following `for` loop computes this value:

```
int lotteryOdds = 1;
for (int i = 1; i <= k; i++)
    lotteryOdds = lotteryOdds * (n - i + 1) / i;
```

 NOTE: See “The ‘for each’ Loop” on page 91 for a description of the “generalized for loop” (also called “for each” loop) that was added to the Java language in Java SE 5.0.

#### Listing 3-5 LotteryOdds.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates a <code>for</code> loop.
5.  * @version 1.20 2004-02-10
6.  * @author Cay Horstmann
7.  */
8. public class LotteryOdds
9. {
10.     public static void main(String[] args)
11.     {
12.         Scanner in = new Scanner(System.in);
13.
14.         System.out.print("How many numbers do you need to draw? ");
15.         int k = in.nextInt();
16.
17.         System.out.print("What is the highest number you can draw? ");
18.         int n = in.nextInt();
19.
20.         /*
21.          * compute binomial coefficient n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k)
22.          */
23.
24.         int lotteryOdds = 1;
25.         for (int i = 1; i <= k; i++)
26.             lotteryOdds = lotteryOdds * (n - i + 1) / i;
27.
28.         System.out.println("Your odds are 1 in " + lotteryOdds + ". Good luck!");
29.     }
30. }
```

**Multiple Selections—The switch Statement**

The if/else construct can be cumbersome when you have to deal with multiple selections with many alternatives. Java has a switch statement that is exactly like the switch statement in C and C++, warts and all.

For example, if you set up a menuing system with four alternatives like that in Figure 3–13, you could use code that looks like this:

```
Scanner in = new Scanner(System.in);
System.out.print("Select an option (1, 2, 3, 4) ");
int choice = in.nextInt();
switch (choice)
{
    case 1:
        . . .
        break;
    case 2:
        . . .
        break;
    case 3:
        . . .
        break;
    case 4:
        . . .
        break;
    default:
        // bad input
        . . .
        break;
}
```

Execution starts at the case label that matches the value on which the selection is performed and continues until the next break or the end of the switch. If none of the case labels match, then the default clause is executed, if it is present.



**CAUTION:** It is possible for multiple alternatives to be triggered. If you forget to add a break at the end of an alternative, then execution falls through to the next alternative! This behavior is plainly dangerous and a common cause for errors. For that reason, we never use the switch statement in our programs.

The case labels must be integers or enumerated constants. You cannot test strings. For example, the following is an error:

```
String input = . . . ;
switch (input) // ERROR
{
    case "A": // ERROR
        . . .
        break;
    . . .
}
```

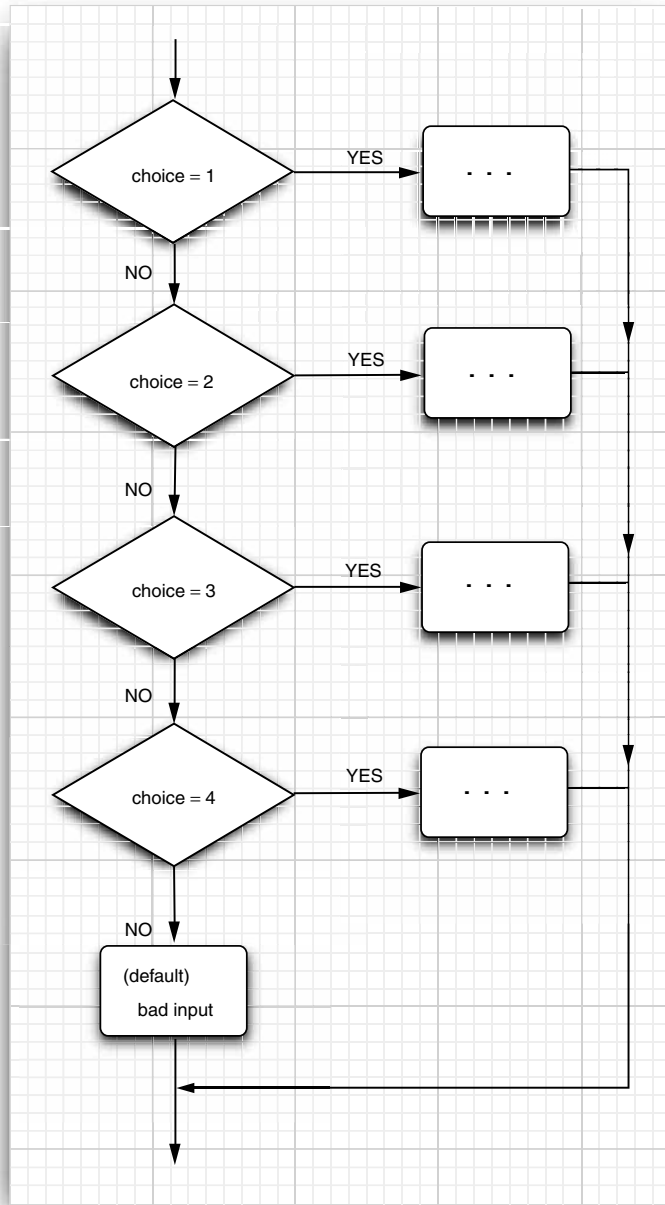


Figure 3-13 Flowchart for the switch statement

When you use the `switch` statement with enumerated constants, you need not supply the name of the enumeration in each label—it is deduced from the `switch` value. For example:

```
Size sz = . . . ;
switch (sz)
{
    case SMALL: // no need to use Size.SMALL
        . . .
        break;
    . . .
}
```

### Statements That Break Control Flow

Although the designers of Java kept the `goto` as a reserved word, they decided not to include it in the language. In general, `goto` statements are considered poor style. Some programmers feel the anti-`goto` forces have gone too far (see, for example, the famous article of Donald Knuth called “Structured Programming with `goto` statements”). They argue that unrestricted use of `goto` is error prone but that an occasional jump *out of a loop* is beneficial. The Java designers agreed and even added a new statement, the labeled `break`, to support this programming style.

Let us first look at the unlabeled `break` statement. The same `break` statement that you use to exit a `switch` can also be used to break out of a loop. For example:

```
while (years <= 100)
{
    balance += payment;
    double interest = balance * interestRate / 100;
    balance += interest;
    if (balance >= goal) break;
    years++;
}
```

Now the loop is exited if either `years > 100` occurs at the top of the loop or `balance >= goal` occurs in the middle of the loop. Of course, you could have computed the same value for `years` without a `break`, like this:

```
while (years <= 100 && balance < goal)
{
    balance += payment;
    double interest = balance * interestRate / 100;
    balance += interest;
    if (balance < goal)
        years++;
}
```

But note that the test `balance < goal` is repeated twice in this version. To avoid this repeated test, some programmers prefer the `break` statement.

Unlike C++, Java also offers a *labeled break* statement that lets you break out of multiple nested loops. Occasionally something weird happens inside a deeply nested loop. In that case, you may want to break completely out of all the nested loops. It is inconvenient to program that simply by adding extra conditions to the various loop tests.

Here's an example that shows the `break` statement at work. Notice that the label must precede the outermost loop out of which you want to break. It also must be followed by a colon.

```
Scanner in = new Scanner(System.in);
int n;
read_data:
while (. . .) // this loop statement is tagged with the label
{
    . . .
    for (. . .) // this inner loop is not labeled
    {
        System.out.print("Enter a number >= 0: ");
        n = in.nextInt();
        if (n < 0) // should never happen—can't go on
            break read_data;
            // break out of read_data loop
        . . .
    }
}
// this statement is executed immediately after the labeled break
if (n < 0) // check for bad situation
{
    // deal with bad situation
}
else
{
    // carry out normal processing
}
```

If there was a bad input, the labeled `break` moves past the end of the labeled block. As with any use of the `break` statement, you then need to test whether the loop exited normally or as a result of a `break`.



**NOTE:** Curiously, you can apply a label to any statement, even an `if` statement or a block statement, like this:

```
label:
{
    . . .
    if (condition) break label; // exits block
    . . .
}
// jumps here when the break statement executes
```

Thus, if you are lusting after a `goto` and if you can place a block that ends just before the place to which you want to jump, you can use a `break` statement! Naturally, we don't recommend this approach. Note, however, that you can only jump *out of* a block, never *into* a block.

Finally, there is a `continue` statement that, like the `break` statement, breaks the regular flow of control. The `continue` statement transfers control to the header of the innermost enclosing loop. Here is an example:

```

Scanner in = new Scanner(System.in);
while (sum < goal)
{
    System.out.print("Enter a number: ");
    n = in.nextInt();
    if (n < 0) continue;
    sum += n; // not executed if n < 0
}

```

If  $n < 0$ , then the `continue` statement jumps immediately to the loop header, skipping the remainder of the current iteration.

If the `continue` statement is used in a `for` loop, it jumps to the “update” part of the `for` loop. For example, consider this loop:

```

for (count = 1; count <= 100; count++)
{
    System.out.print("Enter a number, -1 to quit: ");
    n = in.nextInt();
    if (n < 0) continue;
    sum += n; // not executed if n < 0
}

```

If  $n < 0$ , then the `continue` statement jumps to the `count++` statement.

There is also a labeled form of the `continue` statement that jumps to the header of the loop with the matching label.



**TIP:** Many programmers find the `break` and `continue` statements confusing. These statements are entirely optional—you can always express the same logic without them. In this book, we never use `break` or `continue`.

## Big Numbers

If the precision of the basic integer and floating-point types is not sufficient, you can turn to a couple of handy classes in the `java.math` package: `BigInteger` and `BigDecimal`. These are classes for manipulating numbers with an arbitrarily long sequence of digits. The `BigInteger` class implements arbitrary precision integer arithmetic, and `BigDecimal` does the same for floating-point numbers.

Use the static `valueOf` method to turn an ordinary number into a big number:

```
BigInteger a = BigInteger.valueOf(100);
```

Unfortunately, you cannot use the familiar mathematical operators such as `+` and `*` to combine big numbers. Instead, you must use methods such as `add` and `multiply` in the big number classes.

```

BigInteger c = a.add(b); // c = a + b
BigInteger d = c.multiply(b.add(BigInteger.valueOf(2))); // d = c * (b + 2)

```



**C++ NOTE:** Unlike C++, Java has no programmable operator overloading. There was no way for the programmer of the `BigInteger` class to redefine the `+` and `*` operators to give the `add` and `multiply` operations of the `BigInteger` classes. The language designers did overload the `+` operator to denote concatenation of strings. They chose not to overload other operators, and they did not give Java programmers the opportunity to overload operators in their own classes.



Listing 3–6 shows a modification of the lottery odds program of Listing 3–5, updated to work with big numbers. For example, if you are invited to participate in a lottery in which you need to pick 60 numbers out of a possible 490 numbers, then this program will tell you that your odds are 1 in 716395843461995557415116222540092933411717612789263493493351 013459481104668848. Good luck!

The program in Listing 3–5 computed the statement

```
lotteryOdds = lotteryOdds * (n - i + 1) / i;
```

When big numbers are used, the equivalent statement becomes

```
lotteryOdds = lotteryOdds.multiply(BigInteger.valueOf(n - i + 1)).divide(BigInteger.valueOf(i));
```

### Listing 3–6 BigIntegerTest.java

```

1. import java.math.*;
2. import java.util.*;
3.
4. /**
5.  * This program uses big numbers to compute the odds of winning the grand prize in a lottery.
6.  * @version 1.20 2004-02-10
7.  * @author Cay Horstmann
8.  */
9. public class BigIntegerTest
10. {
11.     public static void main(String[] args)
12.     {
13.         Scanner in = new Scanner(System.in);
14.
15.         System.out.print("How many numbers do you need to draw? ");
16.         int k = in.nextInt();
17.
18.         System.out.print("What is the highest number you can draw? ");
19.         int n = in.nextInt();
20.
21.         /*
22.          * compute binomial coefficient n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k)
23.          */
24.
25.         BigInteger lotteryOdds = BigInteger.valueOf(1);
26.
27.         for (int i = 1; i <= k; i++)
28.             lotteryOdds = lotteryOdds.multiply(BigInteger.valueOf(n - i + 1)).divide(
29.                 BigInteger.valueOf(i));
30.
31.         System.out.println("Your odds are 1 in " + lotteryOdds + ". Good luck!");
32.     }
33. }
```

**API** `java.math.BigInteger` 1.1

- `BigInteger add(BigInteger other)`
- `BigInteger subtract(BigInteger other)`
- `BigInteger multiply(BigInteger other)`
- `BigInteger divide(BigInteger other)`
- `BigInteger mod(BigInteger other)`  
returns the sum, difference, product, quotient, and remainder of this big integer and other.
- `int compareTo(BigInteger other)`  
returns 0 if this big integer equals other, a negative result if this big integer is less than other, and a positive result otherwise.
- `static BigInteger valueOf(long x)`  
returns a big integer whose value equals  $x$ .

**API** `java.math.BigDecimal` 1.1

- `BigDecimal add(BigDecimal other)`
- `BigDecimal subtract(BigDecimal other)`
- `BigDecimal multiply(BigDecimal other)`
- `BigDecimal divide(BigDecimal other, RoundingMode mode)` 5.0  
returns the sum, difference, product, or quotient of this big decimal and other. To compute the quotient, you must supply a *rounding mode*. The mode `RoundingMode.HALF_UP` is the rounding mode that you learned in school (i.e., round down digits 0 . . . 4, round up digits 5 . . . 9). It is appropriate for routine calculations. See the API documentation for other rounding modes.
- `int compareTo(BigDecimal other)`  
returns 0 if this big decimal equals other, a negative result if this big decimal is less than other, and a positive result otherwise.
- `static BigDecimal valueOf(long x)`
- `static BigDecimal valueOf(long x, int scale)`  
returns a big decimal whose value equals  $x$  or  $x/10^{\text{scale}}$ .

**Arrays**

An array is a data structure that stores a collection of values of the same type. You access each individual value through an integer *index*. For example, if `a` is an array of integers, then `a[i]` is the  $i$ th integer in the array.

You declare an array variable by specifying the array type—which is the element type followed by `[]`—and the array variable name. For example, here is the declaration of an array of integers:

```
int[] a;
```

However, this statement only declares the variable `a`. It does not yet initialize `a` with an actual array. You use the `new` operator to create the array.

```
int[] a = new int[100];
```

This statement sets up an array that can hold 100 integers.



NOTE: You can define an array variable either as

```
int[] a;
```

or as

```
int a[];
```

Most Java programmers prefer the former style because it neatly separates the type `int[]` (integer array) from the variable name.

The array entries are *numbered from 0 to 99* (and not 1 to 100). Once the array is created, you can fill the entries in an array, for example, by using a loop:

```
int[] a = new int[100];
for (int i = 0; i < 100; i++)
    a[i] = i; // fills the array with 0 to 99
```



CAUTION: If you construct an array with 100 elements and then try to access the element `a[100]` (or any other index outside the range 0 . . . 99), then your program will terminate with an “array index out of bounds” exception.

To find the number of elements of an array, use `array.length`. For example:

```
for (int i = 0; i < a.length; i++)
    System.out.println(a[i]);
```

Once you create an array, you cannot change its size (although you can, of course, change an individual array element). If you frequently need to expand the size of an array while a program is running, you should use a different data structure called an *array list*. (See Chapter 5 for more on array lists.)

### The “for each” Loop

Java SE 5.0 introduced a powerful looping construct that allows you to loop through each element in an array (as well as other collections of elements) without having to fuss with index values.

The *enhanced* for loop

```
for (variable : collection) statement
```

sets the given variable to each element of the collection and then executes the statement (which, of course, may be a block). The *collection* expression must be an array or an object of a class that implements the `Iterable` interface, such as `ArrayList`. We discuss array lists in Chapter 5 and the `Iterable` interface in Chapter 2 of Volume II.

For example,

```
for (int element : a)
    System.out.println(element);
```

prints each element of the array `a` on a separate line.

You should read this loop as “for each element in `a`”. The designers of the Java language considered using keywords such as `foreach` and `in`. But this loop was a late addition to the Java language, and in the end nobody wanted to break old code that already contains methods or variables with the same names (such as `System.in`).

Of course, you could achieve the same effect with a traditional `for` loop:

```
for (int i = 0; i < a.length; i++)
    System.out.println(a[i]);
```

However, the “for each” loop is more concise and less error prone. (You don’t have to worry about those pesky start and end index values.)



**NOTE:** The loop variable of the “for each” loop traverses the *elements* of the array, not the index values.

The “for each” loop is a pleasant improvement over the traditional loop if you need to process all elements in a collection. However, there are still plenty of opportunities to use the traditional `for` loop. For example, you may not want to traverse the entire collection, or you may need the index value inside the loop.



**TIP:** There is an even easier way to print all values of an array, using the `toString` method of the `Arrays` class. The call `Arrays.toString(a)` returns a string containing the array elements, enclosed in brackets and separated by commas, such as “[2, 3, 5, 7, 11, 13]”. To print the array, simply call

```
System.out.println(Arrays.toString(a));
```

### **Array Initializers and Anonymous Arrays**

Java has a shorthand to create an array object and supply initial values at the same time. Here’s an example of the syntax at work:

```
int[] smallPrimes = { 2, 3, 5, 7, 11, 13 };
```

Notice that you do not call `new` when you use this syntax.

You can even initialize an *anonymous array*:

```
new int[] { 17, 19, 23, 29, 31, 37 }
```

This expression allocates a new array and fills it with the values inside the braces. It counts the number of initial values and sets the array size accordingly. You can use this syntax to reinitialize an array without creating a new variable. For example,

```
smallPrimes = new int[] { 17, 19, 23, 29, 31, 37 };
```

is shorthand for

```
int[] anonymous = { 17, 19, 23, 29, 31, 37 };
smallPrimes = anonymous;
```



**NOTE:** It is legal to have arrays of length 0. Such an array can be useful if you write a method that computes an array result and the result happens to be empty. You construct an array of length 0 as

```
new elementType[0]
```

Note that an array of length 0 is not the same as `null`. (See Chapter 4 for more information about `null`.)

### Array Copying

You can copy one array variable into another, but then *both variables refer to the same array*:

```
int[] luckyNumbers = smallPrimes;
luckyNumbers[5] = 12; // now smallPrimes[5] is also 12
```

Figure 3–14 shows the result. If you actually want to copy all values of one array into a new array, you use the `copyOf` method in the `Arrays` class:

```
int[] copiedLuckyNumbers = Arrays.copyOf(luckyNumbers, luckyNumbers.length);
```

The second parameter is the length of the new array. A common use of this method is to increase the size of an array:

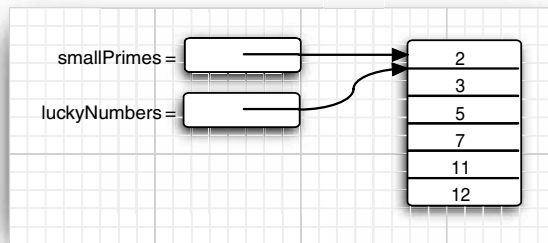
```
luckyNumbers = Arrays.copyOf(luckyNumbers, 2 * luckyNumbers.length);
```

The additional elements are filled with 0 if the array contains numbers, `false` if the array contains `boolean` values. Conversely, if the length is less than the length of the original array, only the initial values are copied.

NOTE: Prior to Java SE 6, the `arraycopy` method in the `System` class was used to copy elements from one array to another. The syntax for this call is

```
System.arraycopy(from, fromIndex, to, toIndex, count);
```

The `to` array must have sufficient space to hold the copied elements.



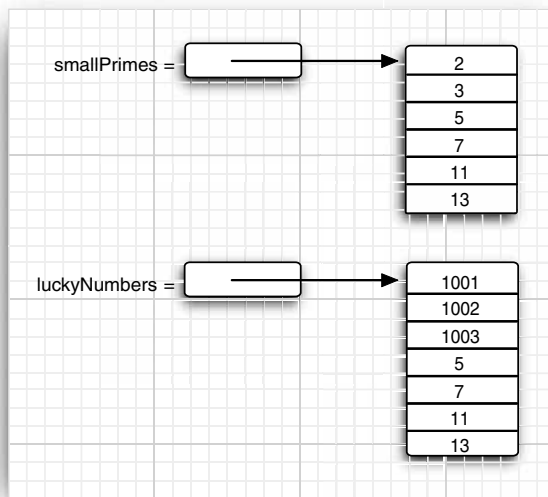
**Figure 3–14 Copying an array variable**

For example, the following statements, whose result is illustrated in Figure 3–15, set up two arrays and then copy the last four entries of the first array to the second array. The copy starts at position 2 in the source array and copies four entries, starting at position 3 of the target.

```
int[] smallPrimes = {2, 3, 5, 7, 11, 13};
int[] luckyNumbers = {1001, 1002, 1003, 1004, 1005, 1006, 1007};
System.arraycopy(smallPrimes, 2, luckyNumbers, 3, 4);
for (int i = 0; i < luckyNumbers.length; i++)
    System.out.println(i + ": " + luckyNumbers[i]);
```

The output is

```
0: 1001
1: 1002
2: 1003
3: 5
4: 7
5: 11
6: 13
```



**Figure 3–15 Copying values between arrays**

**C++** NOTE: A Java array is quite different from a C++ array on the stack. It is, however, essentially the same as a pointer to an array allocated on the *heap*. That is,

```
int[] a = new int[100]; // Java
```

is not the same as

```
int a[100]; // C++
```

but rather

```
int* a = new int[100]; // C++
```

In Java, the `[]` operator is predefined to perform *bounds checking*. Furthermore, there is no pointer arithmetic—you can't increment `a` to point to the next element in the array.

### Command-Line Parameters

You have already seen one example of Java arrays repeated quite a few times. Every Java program has a `main` method with a `String[] args` parameter. This parameter indicates that the `main` method receives an array of strings, namely, the arguments specified on the command line.

For example, consider this program:

```
public class Message
{
    public static void main(String[] args)
    {
        if (args[0].equals("-h"))
            System.out.print("Hello,");
        else if (args[0].equals("-g"))
            System.out.print("Goodbye,");
        // print the other command-line arguments
        for (int i = 1; i < args.length; i++)
            System.out.print(" " + args[i]);
        System.out.println("!");
    }
}
```

If the program is called as

```
java Message -g cruel world
```

then the `args` array has the following contents:

```
args[0]: "-g"
args[1]: "cruel"
args[2]: "world"
```

The program prints the message

```
Goodbye, cruel world!
```



**C++ NOTE:** In the `main` method of a Java program, the name of the program is not stored in the `args` array. For example, when you start up a program as

```
java Message -h world
```

from the command line, then `args[0]` will be `"-h"` and not `"Message"` or `"java"`.

### Array Sorting

To sort an array of numbers, you can use one of the sort methods in the `Arrays` class:

```
int[] a = new int[10000];
. . .
Arrays.sort(a)
```

This method uses a tuned version of the QuickSort algorithm that is claimed to be very efficient on most data sets. The `Arrays` class provides several other convenience methods for arrays that are included in the API notes at the end of this section.

The program in Listing 3-7 puts arrays to work. This program draws a random combination of numbers for a lottery game. For example, if you play a “choose 6 numbers from 49” lottery, then the program might print this:

```

Bet the following combination. It'll make you rich!
 4
 7
 8
19
30
44

```

To select such a random set of numbers, we first fill an array `numbers` with the values 1, 2, . . . , `n`:

```

int[] numbers = new int[n];
for (int i = 0; i < numbers.length; i++)
    numbers[i] = i + 1;

```

A second array holds the numbers to be drawn:

```
int[] result = new int[k];
```

Now we draw `k` numbers. The `Math.random` method returns a random floating-point number that is between 0 (inclusive) and 1 (exclusive). By multiplying the result with `n`, we obtain a random number between 0 and `n - 1`.

```
int r = (int) (Math.random() * n);
```

We set the `i`th result to be the number at that index. Initially, that is just `r + 1`, but as you'll see presently, the contents of the `numbers` array are changed after each draw.

```
result[i] = numbers[r];
```

Now we must be sure never to draw that number again—all lottery numbers must be distinct. Therefore, we overwrite `numbers[r]` with the *last* number in the array and reduce `n` by 1.

```

numbers[r] = numbers[n - 1];
n--;

```

The point is that in each draw we pick an *index*, not the actual value. The index points into an array that contains the values that have not yet been drawn.

After drawing `k` lottery numbers, we sort the `result` array for a more pleasing output:

```

Arrays.sort(result);
for (int r : result)
    System.out.println(r);

```

#### Listing 3-7 LotteryDrawing.java

```

1. import java.util.*;
2.
3. /**
4.  * This program demonstrates array manipulation.
5.  * @version 1.20 2004-02-10
6.  * @author Cay Horstmann
7.  */

```



**Listing 3-7** LotteryDrawing.java (continued)

```
8. public class LotteryDrawing
9. {
10.     public static void main(String[] args)
11.     {
12.         Scanner in = new Scanner(System.in);
13.
14.         System.out.print("How many numbers do you need to draw? ");
15.         int k = in.nextInt();
16.
17.         System.out.print("What is the highest number you can draw? ");
18.         int n = in.nextInt();
19.
20.         // fill an array with numbers 1 2 3 . . . n
21.         int[] numbers = new int[n];
22.         for (int i = 0; i < numbers.length; i++)
23.             numbers[i] = i + 1;
24.
25.         // draw k numbers and put them into a second array
26.         int[] result = new int[k];
27.         for (int i = 0; i < result.length; i++)
28.         {
29.             // make a random index between 0 and n - 1
30.             int r = (int) (Math.random() * n);
31.
32.             // pick the element at the random location
33.             result[i] = numbers[r];
34.
35.             // move the last element into the random location
36.             numbers[r] = numbers[n - 1];
37.             n--;
38.         }
39.
40.         // print the sorted array
41.         Arrays.sort(result);
42.         System.out.println("Bet the following combination. It'll make you rich!");
43.         for (int r : result)
44.             System.out.println(r);
45.     }
46. }
```

**API** java.util.Arrays 1.2

- static String toString(*type[]* a) **5.0**  
returns a string with the elements of a, enclosed in brackets and delimited by commas.  
*Parameters:* a an array of type int, long, short, char, byte, boolean, float, or double.

- static *type* copyOf(*type*[] a, int length) 6
- static *type* copyOf(*type*[] a, int start, int end) 6  
returns an array of the same type as a, of length either length or end - start, filled with the values of a.  
*Parameters:*
  - a an array of type int, long, short, char, byte, boolean, float, or double.
  - start the starting index (inclusive).
  - end the ending index (exclusive). May be larger than a.length, in which case the result is padded with 0 or false values.
  - length the length of the copy. If length is larger than a.length, the result is padded with 0 or false values. Otherwise, only the initial length values are copied.
- static void sort(*type*[] a)  
sorts the array, using a tuned QuickSort algorithm.  
*Parameters:*
  - a an array of type int, long, short, char, byte, float, or double.
- static int binarySearch(*type*[] a, *type* v)
- static int binarySearch(*type*[] a, int start, int end *type* v) 6  
uses the binary search algorithm to search for the value v. If it is found, its index is returned. Otherwise, a negative value r is returned; -r - 1 is the spot at which v should be inserted to keep a sorted.  
*Parameters:*
  - a a sorted array of type int, long, short, char, byte, float, or double.
  - start the starting index (inclusive).
  - end the ending index (exclusive).
  - v a value of the same type as the elements of a.
- static void fill(*type*[] a, *type* v)  
sets all elements of the array to v.  
*Parameters:*
  - a an array of type int, long, short, char, byte, boolean, float, or double.
  - v a value of the same type as the elements of a.
- static boolean equals(*type*[] a, *type*[] b)  
returns true if the arrays have the same length, and if the elements in corresponding indexes match.  
*Parameters:*
  - a, b arrays of type int, long, short, char, byte, boolean, float, or double.

**API** `java.lang.System` 1.1

- static void `arraycopy(Object from, int fromIndex, Object to, int toIndex, int count)` copies elements from the first array to the second array.

*Parameters:*

<code>from</code>	an array of any type (Chapter 5 explains why this is a parameter of type <code>Object</code> ).
<code>fromIndex</code>	the starting index from which to copy elements.
<code>to</code>	an array of the same type as <code>from</code> .
<code>toIndex</code>	the starting index to which to copy elements.
<code>count</code>	the number of elements to copy.

**Multidimensional Arrays**

Multidimensional arrays use more than one index to access array elements. They are used for tables and other more complex arrangements. You can safely skip this section until you have a need for this storage mechanism.

Suppose you want to make a table of numbers that shows how much an investment of \$10,000 will grow under different interest rate scenarios in which interest is paid annually and reinvested. Table 3–8 illustrates this scenario.

**Table 3–8 Growth of an Investment at Different Interest Rates**

10%	11%	12%	13%	14%	15%
10,000.00	10,000.00	10,000.00	10,000.00	10,000.00	10,000.00
11,000.00	11,100.00	11,200.00	11,300.00	11,400.00	11,500.00
12,100.00	12,321.00	12,544.00	12,769.00	12,996.00	13,225.00
13,310.00	13,676.31	14,049.28	14,428.97	14,815.44	15,208.75
14,641.00	15,180.70	15,735.19	16,304.74	16,889.60	17,490.06
16,105.10	16,850.58	17,623.42	18,424.35	19,254.15	20,113.57
17,715.61	18,704.15	19,738.23	20,819.52	21,949.73	23,130.61
19,487.17	20,761.60	22,106.81	23,526.05	25,022.69	26,600.20
21,435.89	23,045.38	24,759.63	26,584.44	28,525.86	30,590.23
23,579.48	25,580.37	27,730.79	30,040.42	32,519.49	35,178.76

You can store this information in a two-dimensional array (or matrix), which we call `balances`.

Declaring a two-dimensional array in Java is simple enough. For example:

```
double[][] balances;
```

As always, you cannot use the array until you initialize it with a call to `new`. In this case, you can do the initialization as follows:

```
balances = new double[NYEARS][NRATES];
```

In other cases, if you know the array elements, you can use a shorthand notion for initializing multidimensional arrays without needing a call to `new`. For example:

```
int[][] magicSquare =
{
    {16, 3, 2, 13},
    {5, 10, 11, 8},
    {9, 6, 7, 12},
    {4, 15, 14, 1}
};
```

Once the array is initialized, you can access individual elements by supplying two brackets, for example, `balances[i][j]`.

The example program stores a one-dimensional array `interest` of interest rates and a two-dimensional array `balance` of account balances, one for each year and interest rate. We initialize the first row of the array with the initial balance:

```
for (int j = 0; j < balance[0].length; j++)
    balances[0][j] = 10000;
```

Then we compute the other rows, as follows:

```
for (int i = 1; i < balances.length; i++)
{
    for (int j = 0; j < balances[i].length; j++)
    {
        double oldBalance = balances[i - 1][j];
        double interest = . . . ;
        balances[i][j] = oldBalance + interest;
    }
}
```

Listing 3-8 shows the full program.



**NOTE:** A “for each” loop does not automatically loop through all entries in a two-dimensional array. Instead, it loops through the rows, which are themselves one-dimensional arrays. To visit all elements of a two-dimensional array `a`, nest two loops, like this:

```
for (double[] row : a)
    for (double value : row)
        do something with value
```



**TIP:** To print out a quick and dirty list of the elements of a two-dimensional array, call `System.out.println(Arrays.deepToString(a));`

The output is formatted like this:

```
[[16, 3, 2, 13], [5, 10, 11, 8], [9, 6, 7, 12], [4, 15, 14, 1]]
```

**Listing 3-8** CompoundInterest.java

```
1. /**
2.  * This program shows how to store tabular data in a 2D array.
3.  * @version 1.40 2004-02-10
4.  * @author Cay Horstmann
5.  */
6. public class CompoundInterest
7. {
8.     public static void main(String[] args)
9.     {
10.         final double STARTRATE = 10;
11.         final int NRATES = 6;
12.         final int NYEARS = 10;
13.
14.         // set interest rates to 10 . . . 15%
15.         double[] interestRate = new double[NRATES];
16.         for (int j = 0; j < interestRate.length; j++)
17.             interestRate[j] = (STARTRATE + j) / 100.0;
18.
19.         double[][] balances = new double[NYEARS][NRATES];
20.
21.         // set initial balances to 10000
22.         for (int j = 0; j < balances[0].length; j++)
23.             balances[0][j] = 10000;
24.
25.         // compute interest for future years
26.         for (int i = 1; i < balances.length; i++)
27.         {
28.             for (int j = 0; j < balances[i].length; j++)
29.             {
30.                 // get last year's balances from previous row
31.                 double oldBalance = balances[i - 1][j];
32.
33.                 // compute interest
34.                 double interest = oldBalance * interestRate[j];
35.
36.                 // compute this year's balances
37.                 balances[i][j] = oldBalance + interest;
38.             }
39.         }
40.
41.         // print one row of interest rates
42.         for (int j = 0; j < interestRate.length; j++)
43.             System.out.printf("%9.0f%%", 100 * interestRate[j]);
44.
45.         System.out.println();
46.
47.         // print balance table
48.         for (double[] row : balances)
49.         {
```

**Listing 3-8** CompoundInterest.java (continued)

```

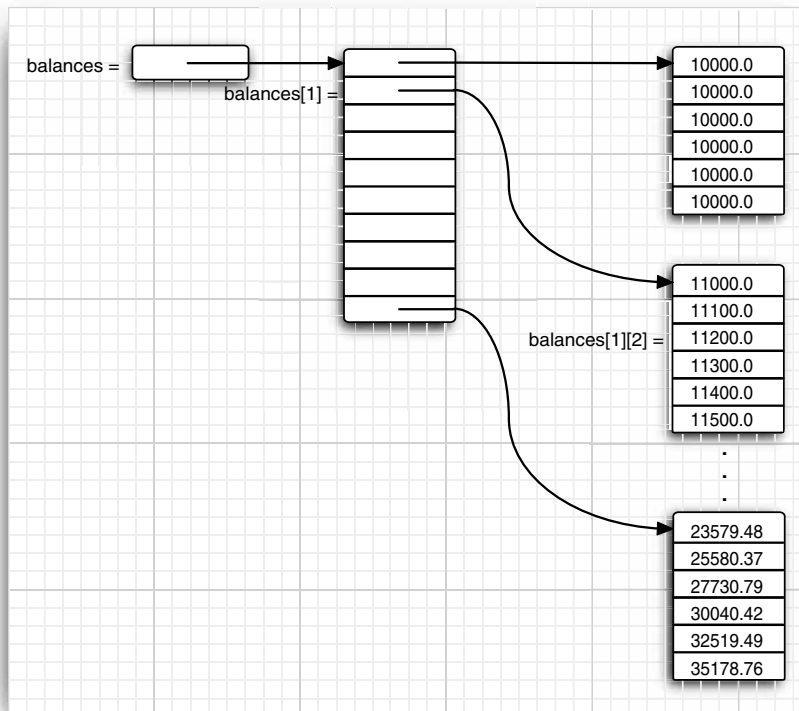
50.     // print table row
51.     for (double b : row)
52.         System.out.printf("%10.2f", b);
53.
54.     System.out.println();
55. }
56. }
57. }

```

**Ragged Arrays**

So far, what you have seen is not too different from other programming languages. But there is actually something subtle going on behind the scenes that you can sometimes turn to your advantage: Java has *no* multidimensional arrays at all, only one-dimensional arrays. Multidimensional arrays are faked as “arrays of arrays.”

For example, the `balances` array in the preceding example is actually an array that contains ten elements, each of which is an array of six floating-point numbers (see Figure 3-16).



**Figure 3-16** A two-dimensional array

The expression `balances[i]` refers to the *i*th subarray, that is, the *i*th row of the table. It is itself an array, and `balances[i][j]` refers to the *j*th entry of that array.

Because rows of arrays are individually accessible, you can actually swap them!

```
double[] temp = balances[i];
balances[i] = balances[i + 1];
balances[i + 1] = temp;
```

It is also easy to make “ragged” arrays, that is, arrays in which different rows have different lengths. Here is the standard example. Let us make an array in which the entry at row *i* and column *j* equals the number of possible outcomes of a “choose *j* numbers from *i* numbers” lottery.

```
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
```

Because *j* can never be larger than *i*, the matrix is triangular. The *i*th row has *i* + 1 elements. (We allow choosing 0 elements; there is one way to make such a choice.) To build this ragged array, first allocate the array holding the rows.

```
int[][] odds = new int[NMAX + 1][];
```

Next, allocate the rows.

```
for (int n = 0; n <= NMAX; n++)
    odds[n] = new int[n + 1];
```

Now that the array is allocated, we can access the elements in the normal way, provided we do not overstep the bounds.

```
for (int n = 0; n < odds.length; n++)
    for (int k = 0; k < odds[n].length; k++)
    {
        // compute lotteryOdds
        . . .
        odds[n][k] = lotteryOdds;
    }
```

Listing 3–9 gives the complete program.



**C++ NOTE:** In C++, the Java declaration

```
double[][] balances = new double[10][6]; // Java
```

is not the same as

```
double balances[10][6]; // C++
```

or even

```
double (*balances)[6] = new double[10][6]; // C++
```

Instead, an array of 10 pointers is allocated:

```
double** balances = new double*[10]; // C++
```

Then, each element in the pointer array is filled with an array of 6 numbers:

```
for (i = 0; i < 10; i++)
    balances[i] = new double[6];
```

Mercifully, this loop is automatic when you ask for a new `double[10][6]`. When you want ragged arrays, you allocate the row arrays separately.

#### Listing 3-9 LotteryArray.java

```
1. /**
2.  * This program demonstrates a triangular array.
3.  * @version 1.20 2004-02-10
4.  * @author Cay Horstmann
5.  */
6. public class LotteryArray
7. {
8.     public static void main(String[] args)
9.     {
10.        final int NMAX = 10;
11.
12.        // allocate triangular array
13.        int[][] odds = new int[NMAX + 1][];
14.        for (int n = 0; n <= NMAX; n++)
15.            odds[n] = new int[n + 1];
16.
17.        // fill triangular array
18.        for (int n = 0; n < odds.length; n++)
19.            for (int k = 0; k < odds[n].length; k++)
20.            {
21.                /*
22.                 * compute binomial coefficient n*(n-1)*(n-2)*...*(n-k+1)/(1*2*3*...*k)
23.                 */
24.                int lotteryOdds = 1;
25.                for (int i = 1; i <= k; i++)
26.                    lotteryOdds = lotteryOdds * (n - i + 1) / i;
27.
28.                odds[n][k] = lotteryOdds;
29.            }
30.
31.        // print triangular array
32.        for (int[] row : odds)
33.        {
34.            for (int odd : row)
35.                System.out.printf("%4d", odd);
36.            System.out.println();
37.        }
38.    }
39. }
```

You have now seen the fundamental programming structures of the Java language. The next chapter covers object-oriented programming in Java.



# *Chapter*

# 4

## OBJECTS AND CLASSES

- ▼ INTRODUCTION TO OBJECT-ORIENTED PROGRAMMING
- ▼ USING PREDEFINED CLASSES
- ▼ DEFINING YOUR OWN CLASSES
- ▼ STATIC FIELDS AND METHODS
- ▼ METHOD PARAMETERS
- ▼ OBJECT CONSTRUCTION
- ▼ PACKAGES
- ▼ THE CLASS PATH
- ▼ DOCUMENTATION COMMENTS
- ▼ CLASS DESIGN HINTS

In this chapter, we

- Introduce you to object-oriented programming;
- Show you how you can create objects that belong to classes in the standard Java library; and
- Show you how to write your own classes.

If you do not have a background in object-oriented programming, you will want to read this chapter carefully. Thinking about object-oriented programming requires a different way of thinking than for procedural languages. The transition is not always easy, but you do need some familiarity with object concepts to go further with Java.

For experienced C++ programmers, this chapter, like the previous chapter, presents familiar information; however, there are enough differences between the two languages that you should read the later sections of this chapter carefully. You'll find the C++ notes helpful for making the transition.

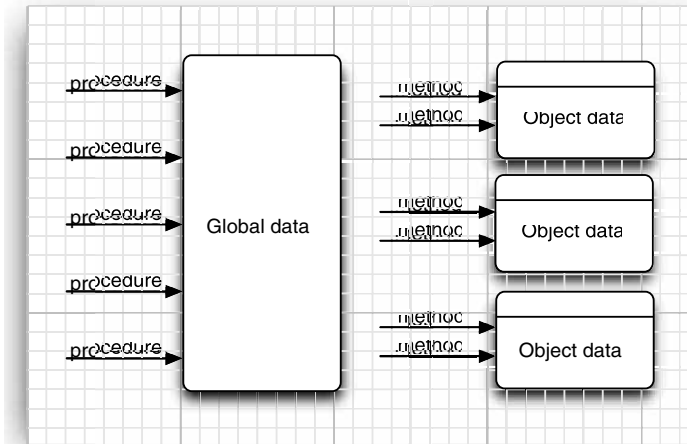
### Introduction to Object-Oriented Programming

Object-oriented programming (or OOP for short) is the dominant programming paradigm these days, having replaced the “structured,” procedural programming techniques that were developed in the 1970s. Java is totally object oriented, and you have to be familiar with OOP to become productive with Java.

An object-oriented program is made of objects. Each object has a specific functionality that is exposed to its users, and a hidden implementation. Many objects in your programs will be taken “off-the-shelf” from a library; others are custom designed. Whether you build an object or buy it might depend on your budget or on time. But, basically, as long as objects satisfy your specifications, you don't care how the functionality was implemented. In OOP, you don't care how an object is implemented as long as it does what you want.

Traditional structured programming consists of designing a set of procedures (or *algorithms*) to solve a problem. After the procedures were determined, the traditional next step was to find appropriate ways to store the data. This is why the designer of the Pascal language, Niklaus Wirth, called his famous book on programming *Algorithms + Data Structures = Programs* (Prentice Hall, 1975). Notice that in Wirth's title, algorithms come first, and data structures come second. This mimics the way programmers worked at that time. First, they decided the procedures for manipulating the data; then, they decided what structure to impose on the data to make the manipulations easier. OOP reverses the order and puts data first, then looks at the algorithms that operate on the data.

For small problems, the breakdown into procedures works very well. But objects are more appropriate for larger problems. Consider a simple web browser. It might require 2,000 procedures for its implementation, all of which manipulate a set of global data. In the object-oriented style, there might be 100 classes with an average of 20 methods per class (see Figure 4-1). The latter structure is much easier for a programmer to grasp. It is also much easier to find bugs. Suppose the data of a particular object is in an incorrect state. It is far easier to search for the culprit among the 20 methods that had access to that data item than among 2,000 procedures.



**Figure 4-1** Procedural vs. OO programming

### Classes

A *class* is the template or blueprint from which objects are made. Thinking about classes as cookie cutters. Objects are the cookies themselves. When you *construct* an object from a class, you are said to have created an *instance* of the class.

As you have seen, all code that you write in Java is inside a class. The standard Java library supplies several thousand classes for such diverse purposes as user interface design, dates and calendars, and network programming. Nonetheless, you still have to create your own classes in Java to describe the objects of the problem domains of your applications.

*Encapsulation* (sometimes called information hiding) is a key concept in working with objects. Formally, encapsulation is nothing more than combining data and behavior in one package and hiding the implementation details from the user of the object. The data in an object are called its *instance fields*, and the procedures that operate on the data are called its *methods*. A specific object that is an instance of a class will have specific values for its instance fields. The set of those values is the current *state* of the object. Whenever you invoke a method on an object, its state may change.

The key to making encapsulation work is to have methods *never* directly access instance fields in a class other than their own. Programs should interact with object data *only* through the object's methods. Encapsulation is the way to give the object its "black box" behavior, which is the key to reuse and reliability. This means a class may totally change how it stores its data, but as long as it continues to use the same methods to manipulate the data, no other object will know or care.

When you do start writing your own classes in Java, another tenet of OOP makes this easier: classes can be built by *extending* other classes. Java, in fact, comes with a "cosmic

superclass” called `Object`. All other classes extend this class. You will see more about the `Object` class in the next chapter.

When you extend an existing class, the new class has all the properties and methods of the class that you extend. You supply new methods and data fields that apply to your new class only. The concept of extending a class to obtain another class is called *inheritance*. See the next chapter for details on inheritance.

### **Objects**

To work with OOP, you should be able to identify three key characteristics of objects:

- The object’s *behavior*—What can you do with this object, or what methods can you apply to it?
- The object’s *state*—How does the object react when you apply those methods?
- The object’s *identity*—How is the object distinguished from others that may have the same behavior and state?

All objects that are instances of the same class share a family resemblance by supporting the same *behavior*. The behavior of an object is defined by the methods that you can call.

Next, each object stores information about what it currently looks like. This is the object’s *state*. An object’s state may change over time, but not spontaneously. A change in the state of an object must be a consequence of method calls. (If the object state changed without a method call on that object, someone broke encapsulation.)

However, the state of an object does not completely describe it, because each object has a distinct *identity*. For example, in an order-processing system, two orders are distinct even if they request identical items. Notice that the individual objects that are instances of a class *always* differ in their identity and *usually* differ in their state.

These key characteristics can influence each other. For example, the state of an object can influence its behavior. (If an order is “shipped” or “paid,” it may reject a method call that asks it to add or remove items. Conversely, if an order is “empty,” that is, no items have yet been ordered, it should not allow itself to be shipped.)

### **Identifying Classes**

In a traditional procedural program, you start the process at the top, with the `main` function. When designing an object-oriented system, there is no “top,” and newcomers to OOP often wonder where to begin. The answer is, you first find classes and then you add methods to each class.

A simple rule of thumb in identifying classes is to look for nouns in the problem analysis. Methods, on the other hand, correspond to verbs.

For example, in an order-processing system, some of these nouns are

- Item
- Order
- Shipping address
- Payment
- Account

These nouns may lead to the classes `Item`, `Order`, and so on.

Next, look for verbs. Items are *added* to orders. Orders are *shipped* or *canceled*. Payments are *applied* to orders. With each verb, such as “add,” “ship,” “cancel,” and “apply,” you identify the one object that has the major responsibility for carrying it out. For example, when a new item is added to an order, the order object should be the one in charge because it knows how it stores and sorts items. That is, *add* should be a method of the `Order` class that takes an `Item` object as a parameter.

Of course, the “noun and verb” rule is only a rule of thumb, and only experience can help you decide which nouns and verbs are the important ones when building your classes.

### **Relationships between Classes**

The most common relationships between classes are

- *Dependence* (“uses-a”)
- *Aggregation* (“has-a”)
- *Inheritance* (“is-a”)

The *dependence*, or “uses-a” relationship, is the most obvious and also the most general. For example, the `Order` class uses the `Account` class because `Order` objects need to access `Account` objects to check for credit status. But the `Item` class does not depend on the `Account` class, because `Item` objects never need to worry about customer accounts. Thus, a class depends on another class if its methods use or manipulate objects of that class.

Try to minimize the number of classes that depend on each other. The point is, if a class A is unaware of the existence of a class B, it is also unconcerned about any changes to B! (And this means that changes to B do not introduce bugs into A.) In software engineering terminology, you want to minimize the *coupling* between classes.

The *aggregation*, or “has-a” relationship, is easy to understand because it is concrete; for example, an `Order` object contains `Item` objects. Containment means that objects of class A contain objects of class B.



NOTE: Some methodologists view the concept of aggregation with disdain and prefer to use a more general “association” relationship. From the point of view of modeling, that is understandable. But for programmers, the “has-a” relationship makes a lot of sense. We like to use aggregation for a second reason—the standard notation for associations is less clear. See Table 4–1.

The *inheritance*, or “is-a” relationship, expresses a relationship between a more special and a more general class. For example, a `RushOrder` class inherits from an `Order` class. The specialized `RushOrder` class has special methods for priority handling and a different method for computing shipping charges, but its other methods, such as adding items and billing, are inherited from the `Order` class. In general, if class A extends class B, class A inherits methods from class B but has more capabilities. (We describe inheritance more fully in the next chapter, in which we discuss this important notion at some length.)

Many programmers use the UML (Unified Modeling Language) notation to draw *class diagrams* that describe the relationships between classes. You can see an example of such a diagram in Figure 4–2. You draw classes as rectangles, and relationships as arrows with various adornments. Table 4–1 shows the most common UML arrow styles.

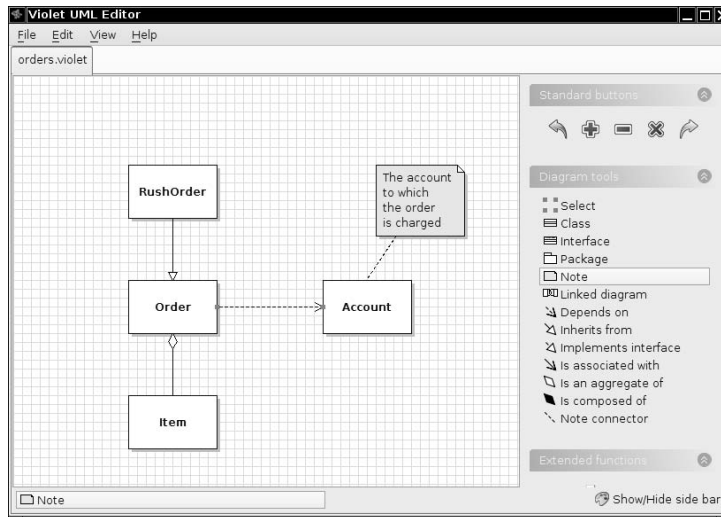


Figure 4-2 A class diagram

✓ NOTE: A number of tools are available for drawing UML diagrams. Several vendors offer high-powered (and high-priced) tools that aim to be the focal point of your development process. Among them are Rational Rose (<http://www.ibm.com/software/awdtools/developer/rose>) and Together (<http://www.borland.com/us/products/together>). Another choice is the open source program ArgoUML (<http://argouml.tigris.org>). A commercially supported version is available from GentleWare (<http://gentleware.com>). If you just want to draw a simple diagrams with a minimum of fuss, try out Violet (<http://violet.sourceforge.net>).

Table 4-1 UML Notation for Class Relationships

Relationship	UML Connector
Inheritance	—>▷
Interface inheritance	- - - - ->▷
Dependency	- - - - ->
Aggregation	◊—
Association	—
Directed association	—>

## Using Predefined Classes

Because you can't do anything in Java without classes, you have already seen several classes at work. However, not all of these show off the typical features of object orientation. Take, for example, the `Math` class. You have seen that you can use methods of the `Math` class, such as `Math.random`, without needing to know how they are implemented—all you need to know is the name and parameters (if any). That is the point of encapsulation and will certainly be true of all classes. But the `Math` class *only* encapsulates functionality; it neither needs nor hides data. Because there is no data, you do not need to worry about making objects and initializing their instance fields—there aren't any!

In the next section, we look at a more typical class, the `Date` class. You will see how to construct objects and call methods of this class.

### Objects and Object Variables

To work with objects, you first construct them and specify their initial state. Then you apply methods to the objects.

In the Java programming language, you use *constructors* to construct new instances. A constructor is a special method whose purpose is to construct and initialize objects. Let us look at an example. The standard Java library contains a `Date` class. Its objects describe points in time, such as “December 31, 1999, 23:59:59 GMT”.



**NOTE:** You may be wondering: Why use classes to represent dates rather than (as in some languages) a built-in type? For example, Visual Basic has a built-in date type and programmers can specify dates in the format `#6/1/1995#`. On the surface, this sounds convenient—programmers can simply use the built-in date type rather than worrying about classes. But actually, how suitable is the Visual Basic design? In some locales, dates are specified as month/day/year, in others as day/month/year. Are the language designers really equipped to foresee these kinds of issues? If they do a poor job, the language becomes an unpleasant muddle, but unhappy programmers are powerless to do anything about it. With classes, the design task is offloaded to a library designer. If the class is not perfect, other programmers can easily write their own classes to enhance or replace the system classes. (To prove the point: The Java date library is a bit muddled, and a major redesign is underway; see <http://jcp.org/en/jsr/detail?id=310>.)

Constructors always have the same name as the class name. Thus, the constructor for the `Date` class is called `Date`. To construct a `Date` object, you combine the constructor with the `new` operator, as follows:

```
new Date()
```

This expression constructs a new object. The object is initialized to the current date and time.

If you like, you can pass the object to a method:

```
System.out.println(new Date());
```

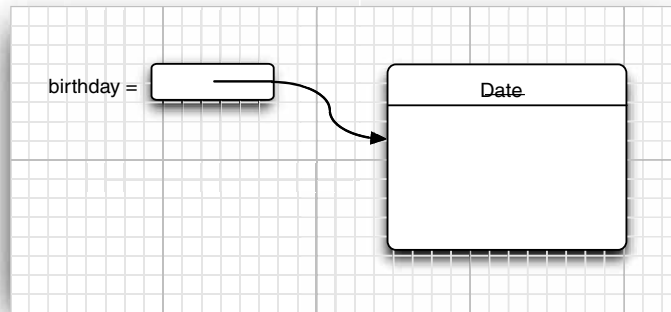
Alternatively, you can apply a method to the object that you just constructed. One of the methods of the `Date` class is the `toString` method. That method yields a string representation of the date. Here is how you would apply the `toString` method to a newly constructed `Date` object:

```
String s = new Date().toString();
```

In these two examples, the constructed object is used only once. Usually, you will want to hang on to the objects that you construct so that you can keep using them. Simply store the object in a variable:

```
Date birthday = new Date();
```

Figure 4-3 shows the object variable `birthday` that refers to the newly constructed object.



**Figure 4-3** Creating a new object

There is an important difference between objects and object variables. For example, the statement

```
Date deadline; // deadline doesn't refer to any object
```

defines an object variable, `deadline`, that can refer to objects of type `Date`. It is important to realize that the variable `deadline` is *not an object* and, in fact, does not yet even refer to an object. You cannot use any `Date` methods on this variable at this time. The statement

```
s = deadline.toString(); // not yet
```

would cause a compile-time error.

You must first initialize the `deadline` variable. You have two choices. Of course, you can initialize the variable with a newly constructed object:

```
deadline = new Date();
```

Or you can set the variable to refer to an existing object:

```
deadline = birthday;
```

Now both variables refer to the *same* object (see Figure 4-4).

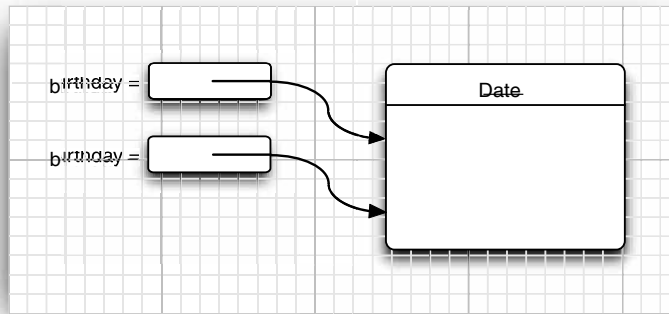
It is important to realize that an object variable doesn't actually contain an object. It only *refers* to an object.

In Java, the value of any object variable is a reference to an object that is stored elsewhere. The return value of the `new` operator is also a reference. A statement such as

```
Date deadline = new Date();
```

has two parts. The expression `new Date()` makes an object of type `Date`, and its value is a reference to that newly created object. That reference is then stored in the `deadline` variable.





**Figure 4-4** Object variables that refer to the same object

You can explicitly set an object variable to `null` to indicate that it currently refers to no object.

```
deadline = null;
. . .
if (deadline != null)
    System.out.println(deadline);
```

If you apply a method to a variable that holds `null`, then a runtime error occurs.

```
birthday = null;
String s = birthday.toString(); // runtime error!
```

Variables are not automatically initialized to `null`. You must initialize them, either by calling `new` or by setting them to `null`.



**C++ NOTE:** Many people mistakenly believe that Java object variables behave like C++ references. But in C++ there are no null references, and references cannot be assigned. You should think of Java object variables as analogous to *object pointers* in C++. For example,

```
Date birthday; // Java
```

is really the same as

```
Date* birthday; // C++
```

Once you make this association, everything falls into place. Of course, a `Date*` pointer isn't initialized until you initialize it with a call to `new`. The syntax is almost the same in C++ and Java.

```
Date* birthday = new Date(); // C++
```

If you copy one variable to another, then both variables refer to the same date—they are pointers to the same object. The equivalent of the Java `null` reference is the C++ `NULL` pointer.

All Java objects live on the heap. When an object contains another object variable, that variable still contains just a pointer to yet another heap object.

In C++, pointers make you nervous because they are so error prone. It is easy to create bad pointers or to mess up memory management. In Java, these problems simply go away. If you use an uninitialized pointer, the runtime system will reliably generate a runtime error instead of producing random results. You don't worry about memory management, because the garbage collector takes care of it.

C++ makes quite an effort, with its support for copy constructors and assignment operators, to allow the implementation of objects that copy themselves automatically. For example, a copy of a linked list is a new linked list with the same contents but with an independent set of links. This makes it possible to design classes with the same copy behavior as the built-in types. In Java, you must use the `clone` method to get a complete copy of an object.

### **The `GregorianCalendar` Class of the Java Library**

In the preceding examples, we used the `Date` class that is a part of the standard Java library. An instance of the `Date` class has a state, namely *a particular point in time*.

Although you don't need to know this when you use the `Date` class, the time is represented by the number of milliseconds (positive or negative) from a fixed point, the so-called *epoch*, which is 00:00:00 UTC, January 1, 1970. UTC is the Coordinated Universal Time, the scientific time standard that is, for practical purposes, the same as the more familiar GMT or Greenwich Mean Time.

But as it turns out, the `Date` class is not very useful for manipulating dates. The designers of the Java library take the point of view that a date description such as "December 31, 1999, 23:59:59" is an arbitrary convention, governed by a *calendar*. This particular description follows the Gregorian calendar, which is the calendar used in most places of the world. The same point in time would be described quite differently in the Chinese or Hebrew lunar calendars, not to mention the calendar used by your customers from Mars.



**NOTE:** Throughout human history, civilizations grappled with the design of calendars that attached names to dates and brought order to the solar and lunar cycles. For a fascinating explanation of calendars around the world, from the French Revolutionary calendar to the Mayan long count, see *Calendrical Calculations, Second Edition* by Nachum Dershowitz and Edward M. Reingold (Cambridge University Press, 2001).

The library designers decided to separate the concerns of keeping time and attaching names to points in time. Therefore, the standard Java library contains two separate classes: the `Date` class, which represents a point in time, and the `GregorianCalendar` class, which expresses dates in the familiar calendar notation. In fact, the `GregorianCalendar` class extends a more generic `Calendar` class that describes the properties of calendars in general. In theory, you can extend the `Calendar` class and implement the Chinese lunar calendar or a Martian calendar. However, the standard library does not contain any calendar implementations besides the Gregorian calendar.

Separating time measurement from calendars is good object-oriented design. In general, it is a good idea to use separate classes to express different concepts.

The `Date` class has only a small number of methods that allow you to compare two points in time. For example, the `before` and `after` methods tell you if one point in time comes before or after another:

```
if (today.before(birthday))
    System.out.println("Still time to shop for a gift.");
```



NOTE: Actually, the `Date` class has methods such as `getDay`, `getMonth`, and `getYear`, but these methods are *deprecated*. A method is deprecated when a library designer realizes that the method should have never been introduced in the first place.

These methods were a part of the `Date` class before the library designers realized that it makes more sense to supply separate calendar classes. When the calendar classes were introduced, the `Date` methods were tagged as deprecated. You can still use them in your programs, but you will get unsightly compiler warnings if you do. It is a good idea to stay away from using deprecated methods because they may be removed in a future version of the library.

The `GregorianCalendar` class has many more methods than the `Date` class. In particular, it has several useful constructors. The expression

```
new GregorianCalendar()
```

constructs a new object that represents the date and time at which the object was constructed.

You can construct a calendar object for midnight on a specific date by supplying year, month, and day:

```
new GregorianCalendar(1999, 11, 31)
```

Somewhat curiously, the months are counted from 0. Therefore, 11 is December. For greater clarity, there are constants like `Calendar.DECEMBER`:

```
new GregorianCalendar(1999, Calendar.DECEMBER, 31)
```

You can also set the time:

```
new GregorianCalendar(1999, Calendar.DECEMBER, 31, 23, 59, 59)
```

Of course, you will usually want to store the constructed object in an object variable:

```
GregorianCalendar deadline = new GregorianCalendar(. . .);
```

The `GregorianCalendar` has encapsulated instance fields to maintain the date to which it is set. Without looking at the source code, it is impossible to know the representation that the class uses internally. But, of course, the point of encapsulation is that this doesn't matter. What matters are the methods that a class exposes.

### ***Mutator and Accessor Methods***

At this point, you are probably asking yourself: How do I get at the current day or month or year for the date encapsulated in a specific `GregorianCalendar` object? And how do I change the values if I am unhappy with them? You can find out how to carry out these tasks by looking at the on-line documentation or the API notes at the end of this section. We go over the most important methods in this section.

The job of a calendar is to compute attributes, such as the date, weekday, month, or year, of a certain point in time. To query one of these settings, you use the `get` method of the `GregorianCalendar` class. To select the item that you want to get, you pass a constant defined in the `Calendar` class, such as `Calendar.MONTH` or `Calendar.DAY_OF_WEEK`:

```
GregorianCalendar now = new GregorianCalendar();
int month = now.get(Calendar.MONTH);
int weekday = now.get(Calendar.DAY_OF_WEEK);
```

The API notes list all the constants that you can use.

You change the state with a call to the set method:

```
deadline.set(Calendar.YEAR, 2001);
deadline.set(Calendar.MONTH, Calendar.APRIL);
deadline.set(Calendar.DAY_OF_MONTH, 15);
```

There is also a convenience method to set the year, month, and day with a single call:

```
deadline.set(2001, Calendar.APRIL, 15);
```

Finally, you can add a number of days, weeks, months, and so on, to a given calendar object:

```
deadline.add(Calendar.MONTH, 3); // move deadline by 3 months
```

If you add a negative number, then the calendar is moved backwards.

There is a conceptual difference between the get method on the one hand and the set and add methods on the other hand. The get method only looks up the state of the object and reports on it. The set and add methods modify the state of the object. Methods that change instance fields are called *mutator methods*, and those that only access instance fields without modifying them are called *accessor methods*.



**C++** NOTE: In C++, the const suffix denotes accessor methods. A method that is not declared as const is assumed to be a mutator. However, in the Java programming language, no special syntax distinguishes between accessors and mutators.

A common convention is to prefix accessor methods with the prefix `get` and mutator methods with the prefix `set`. For example, the `GregorianCalendar` class has methods `getTime` and `setTime` that get and set the point in time that a calendar object represents:

```
Date time = calendar.getTime();
calendar.setTime(time);
```

These methods are particularly useful for converting between the `GregorianCalendar` and `Date` classes. Here is an example. Suppose you know the year, month, and day and you want to make a `Date` object with those settings. Because the `Date` class knows nothing about calendars, first construct a `GregorianCalendar` object and then call the `getTime` method to obtain a date:

```
GregorianCalendar calendar = new GregorianCalendar(year, month, day);
Date hireDay = calendar.getTime();
```

Conversely, if you want to find the year, month, or day of a `Date` object, you construct a `GregorianCalendar` object, set the time, and then call the `get` method:

```
GregorianCalendar calendar = new GregorianCalendar();
calendar.setTime(hireDay);
int year = calendar.get(Calendar.YEAR);
```

We finish this section with a program that puts the `GregorianCalendar` class to work. The program displays a calendar for the current month, like this:

```

Sun Mon Tue Wed Thu Fri Sat
          1
 2  3  4  5  6  7  8
 9 10 11 12 13 14 15
16 17 18 19* 20 21 22
23 24 25 26 27 28 29
30 31

```

The current day is marked with an asterisk (\*). As you can see, the program needs to know how to compute the length of a month and the weekday of a given day.

Let us go through the key steps of the program. First, we construct a calendar object that is initialized with the current date.

```
GregorianCalendar d = new GregorianCalendar();
```

We capture the current day and month by calling the `get` method twice.

```
int today = d.get(Calendar.DAY_OF_MONTH);
int month = d.get(Calendar.MONTH);
```

Then we set `d` to the first of the month and get the weekday of that date.

```
d.set(Calendar.DAY_OF_MONTH, 1);
int weekday = d.get(Calendar.DAY_OF_WEEK);
```

The variable `weekday` is set to `Calendar.SUNDAY` if the first day of the month is a Sunday, to `Calendar.MONDAY` if it is a Monday, and so on. (These values are actually the integers 1, 2, . . . , 7, but it is best not to write code that depends on that knowledge.)

Note that the first line of the calendar is indented, so that the first day of the month falls on the appropriate weekday. This is a bit tricky since there are varying conventions about the starting day of the week. In the United States, the week starts with Sunday and ends with Saturday, whereas in Europe, the week starts with Monday and ends with Sunday.

The Java virtual machine is aware of the *locale* of the current user. The locale describes local formatting conventions, including the start of the week and the names of the weekdays.



**TIP:** If you want to see the program output in a different locale, add a line such as the following as the first line of the `main` method:

```
Locale.setDefault(Locale.ITALY);
```

The `getFirstDayOfWeek` method gets the starting weekday in the current locale. To determine the required indentation, we subtract 1 from the day of the calendar object until we reach the first day of the week.

```
int firstDayOfWeek = d.getFirstDayOfWeek();
int indent = 0;
while (weekday != firstDayOfWeek)
{
    indent++;
    d.add(Calendar.DAY_OF_MONTH, -1);
    weekday = d.get(Calendar.DAY_OF_WEEK);
}
```

Next, we print the header with the weekday names. These are available from the class `DateFormatSymbols`.

```
String [] weekdayNames = new DateFormatSymbols().getShortWeekdays();
```

The `getShortWeekdays` method returns a string with short weekday names in the user's language (such as "Sun", "Mon", and so on in English). The array is indexed by weekday values. Here is the loop to print the header:

```
do
{
    System.out.printf("%4s", weekdayNames[weekday]);
    d.add(Calendar.DAY_OF_MONTH, 1);
    weekday = d.get(Calendar.DAY_OF_WEEK);
}
while (weekday != firstDayOfWeek);
System.out.println();
```

Now, we are ready to print the body of the calendar. We indent the first line and set the date object back to the start of the month. We enter a loop in which `d` traverses the days of the month.

In each iteration, we print the date value. If `d` is today, the date is marked with an `*`. If we reach the beginning of each new week, we print a new line. Then, we advance `d` to the next day:

```
    d.add(Calendar.DAY_OF_MONTH, 1);
```

When do we stop? We don't know whether the month has 31, 30, 29, or 28 days. Instead, we keep iterating while `d` is still in the current month.

```
do
{
    . . .
}
while (d.get(Calendar.MONTH) == month);
```

Once `d` has moved into the next month, the program terminates.

Listing 4-1 shows the complete program.

As you can see, the `GregorianCalendar` class makes it possible to write a calendar program that takes care of complexities such as weekdays and the varying month lengths. You don't need to know *how* the `GregorianCalendar` class computes months and weekdays. You just use the *interface* of the class—the `get`, `set`, and `add` methods.

The point of this example program is to show you how you can use the interface of a class to carry out fairly sophisticated tasks without having to know the implementation details.

#### Listing 4-1 CalendarTest.java

```
1. import java.text.DateFormatSymbols;
2. import java.util.*;
3.
4. /**
5.  * @version 1.4 2007-04-07
6.  * @author Cay Horstmann
7.  */
```

**Listing 4-1** CalendarTest.java (continued)

```
8.
9. public class CalendarTest
10. {
11.     public static void main(String[] args)
12.     {
13.         // construct d as current date
14.         GregorianCalendar d = new GregorianCalendar();
15.
16.         int today = d.get(Calendar.DAY_OF_MONTH);
17.         int month = d.get(Calendar.MONTH);
18.
19.         // set d to start date of the month
20.         d.set(Calendar.DAY_OF_MONTH, 1);
21.
22.         int weekday = d.get(Calendar.DAY_OF_WEEK);
23.
24.         // get first day of week (Sunday in the U.S.)
25.         int firstDayOfWeek = d.getFirstDayOfWeek();
26.
27.         // determine the required indentation for the first line
28.         int indent = 0;
29.         while (weekday != firstDayOfWeek)
30.         {
31.             indent++;
32.             d.add(Calendar.DAY_OF_MONTH, -1);
33.             weekday = d.get(Calendar.DAY_OF_WEEK);
34.         }
35.
36.         // print weekday names
37.         String[] weekdayNames = new DateFormatSymbols().getShortWeekdays();
38.         do
39.         {
40.             System.out.printf("%4s", weekdayNames[weekday]);
41.             d.add(Calendar.DAY_OF_MONTH, 1);
42.             weekday = d.get(Calendar.DAY_OF_WEEK);
43.         }
44.         while (weekday != firstDayOfWeek);
45.         System.out.println();
46.
47.         for (int i = 1; i <= indent; i++)
48.             System.out.print(" ");
49.
50.         d.set(Calendar.DAY_OF_MONTH, 1);
51.         do
52.         {
53.             // print day
54.             int day = d.get(Calendar.DAY_OF_MONTH);
55.             System.out.printf("%3d", day);
```

**Listing 4-1** CalendarTest.java (continued)

```

56.
57.     // mark current day with *
58.     if (day == today) System.out.print("*");
59.     else System.out.print(" ");
60.
61.     // advance d to the next day
62.     d.add(Calendar.DAY_OF_MONTH, 1);
63.     weekday = d.get(Calendar.DAY_OF_WEEK);
64.
65.     // start a new line at the start of the week
66.     if (weekday == firstDayOfWeek) System.out.println();
67. }
68. while (d.get(Calendar.MONTH) == month);
69. // the loop exits when d is day 1 of the next month
70.
71. // print final end of line if necessary
72. if (weekday != firstDayOfWeek) System.out.println();
73. }
74. }

```

**API** java.util.GregorianCalendar 1.1

- `GregorianCalendar()`  
constructs a calendar object that represents the current time in the default time zone with the default locale.
- `GregorianCalendar(int year, int month, int day)`
- `GregorianCalendar(int year, int month, int day, int hour, int minutes, int seconds)`  
constructs a Gregorian calendar with the given date and time.

*Parameters:*

year	the year of the date
month	the month of the date. This value is 0-based; for example, 0 for January
day	the day of the month
hour	the hour (between 0 and 23)
minutes	the minutes (between 0 and 59)
seconds	the seconds (between 0 and 59)

- `int get(int field)`  
gets the value of a particular field.

*Parameters:*

field	one of <code>Calendar.ERA</code> , <code>Calendar.YEAR</code> , <code>Calendar.MONTH</code> , <code>Calendar.WEEK_OF_YEAR</code> , <code>Calendar.WEEK_OF_MONTH</code> , <code>Calendar.DAY_OF_MONTH</code> , <code>Calendar.DAY_OF_YEAR</code> , <code>Calendar.DAY_OF_WEEK</code> , <code>Calendar.DAY_OF_WEEK_IN_MONTH</code> , <code>Calendar.AM_PM</code> , <code>Calendar.HOUR</code> , <code>Calendar.HOUR_OF_DAY</code> , <code>Calendar.MINUTE</code> , <code>Calendar.SECOND</code> , <code>Calendar.MILLISECOND</code> , <code>Calendar.ZONE_OFFSET</code> , <code>Calendar.DST_OFFSET</code>
-------	--



- `void set(int field, int value)`  
sets the value of a particular field.  
*Parameters:*

<code>field</code>	one of the constants accepted by <code>get</code>
<code>value</code>	the new value
- `void set(int year, int month, int day)`
- `void set(int year, int month, int day, int hour, int minutes, int seconds)`  
sets the fields to new values.  
*Parameters:*

<code>year</code>	the year of the date
<code>month</code>	the month of the date. This value is 0-based; for example, 0 for January
<code>day</code>	the day of the month
<code>hour</code>	the hour (between 0 and 23)
<code>minutes</code>	the minutes (between 0 and 59)
<code>seconds</code>	the seconds (between 0 and 59)
- `void add(int field, int amount)`  
is a date arithmetic method. Adds the specified amount of time to the given time field. For example, to add 7 days to the current calendar date, call `c.add(Calendar.DAY_OF_MONTH, 7)`.  
*Parameters:*

<code>field</code>	the field to modify (using one of the constants documented in the <code>get</code> method)
<code>amount</code>	the amount by which the field should be changed (can be negative)
- `int getFirstDayOfWeek()`  
gets the first day of the week in the locale of the current user, for example, `Calendar.SUNDAY` in the United States.
- `void setTime(Date time)`  
sets this calendar to the given point in time.  
*Parameters:*

<code>time</code>	a point in time
-------------------	-----------------
- `Date getTime()`  
gets the point in time that is represented by the current value of this calendar object.

**API** `java.text.DateFormatSymbols` 1.1

- `String[] getShortWeekdays()`
- `String[] getShortMonths()`
- `String[] getWeekdays()`
- `String[] getMonths()`  
gets the names of the weekdays or months in the current locale. Uses `Calendar` weekday and month constants as array index values.

## Defining Your Own Classes

In Chapter 3, you started writing simple classes. However, all those classes had just a single `main` method. Now the time has come to show you how to write the kind of “work-horse classes” that are needed for more sophisticated applications. These classes typically do not have a `main` method. Instead, they have their own instance fields and methods. To build a complete program, you combine several classes, one of which has a `main` method.

### An Employee Class

The simplest form for a class definition in Java is

```
class ClassName
{
    constructor1
    constructor2
    . . .
    method1
    method2
    . . .
    field1
    field2
    . . .
}
```



**NOTE:** We adopt the style that the methods for the class come first and the fields come at the end. Perhaps this, in a small way, encourages the notion of looking at the interface first and paying less attention to the implementation.

Consider the following, very simplified, version of an `Employee` class that might be used by a business in writing a payroll system.

```
class Employee
{
    // constructor
    public Employee(String n, double s, int year, int month, int day)
    {
        name = n;
        salary = s;
        GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
        hireDay = calendar.getTime();
    }

    // a method
    public String getName()
    {
        return name;
    }

    // more methods
    . . .
}
```

```
// instance fields
private String name;
private double salary;
private Date hireDay;
}
```

We break down the implementation of this class in some detail in the sections that follow. First, though, Listing 4–2 shows a program that shows the `Employee` class in action.

In the program, we construct an `Employee` array and fill it with three employee objects:

```
Employee[] staff = new Employee[3];

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

Next, we use the `raiseSalary` method of the `Employee` class to raise each employee's salary by 5%:

```
for (Employee e : staff)
    e.raiseSalary(5);
```

Finally, we print out information about each employee, by calling the `getName`, `getSalary`, and `getHireDay` methods:

```
for (Employee e : staff)
    System.out.println("name=" + e.getName()
        + ", salary=" + e.getSalary()
        + ", hireDay=" + e.getHireDay());
```

Note that the example program consists of *two* classes: the `Employee` class and a class `EmployeeTest` with the `public` access specifier. The `main` method with the instructions that we just described is contained in the `EmployeeTest` class.

The name of the source file is `EmployeeTest.java` because the name of the file must match the name of the `public` class. You can have only one `public` class in a source file, but you can have any number of nonpublic classes.

Next, when you compile this source code, the compiler creates two class files in the directory: `EmployeeTest.class` and `Employee.class`.

You start the program by giving the bytecode interpreter the name of the class that contains the `main` method of your program:

```
java EmployeeTest
```

The bytecode interpreter starts running the code in the `main` method in the `EmployeeTest` class. This code in turn constructs three new `Employee` objects and shows you their state.

**Listing 4-2** EmployeeTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program tests the Employee class.
5.  * @version 1.11 2004-02-19
6.  * @author Cay Horstmann
7.  */
8. public class EmployeeTest
9. {
10.     public static void main(String[] args)
11.     {
12.         // fill the staff array with three Employee objects
13.         Employee[] staff = new Employee[3];
14.
15.         staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
16.         staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
17.         staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
18.
19.         // raise everyone's salary by 5%
20.         for (Employee e : staff)
21.             e.raiseSalary(5);
22.
23.         // print out information about all Employee objects
24.         for (Employee e : staff)
25.             System.out.println("name=" + e.getName() + ",salary=" + e.getSalary() + ",hireDay="
26.                 + e.getHireDay());
27.     }
28. }
29.
30. class Employee
31. {
32.     public Employee(String n, double s, int year, int month, int day)
33.     {
34.         name = n;
35.         salary = s;
36.         GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
37.         // GregorianCalendar uses 0 for January
38.         hireDay = calendar.getTime();
39.     }
40.
41.     public String getName()
42.     {
43.         return name;
44.     }
45.
46.     public double getSalary()
47.     {
48.         return salary;
49.     }
}
```

**Listing 4-2** EmployeeTest.java (continued)

```
50.
51. public Date getHireDay()
52. {
53.     return hireDay;
54. }
55.
56. public void raiseSalary(double byPercent)
57. {
58.     double raise = salary * byPercent / 100;
59.     salary += raise;
60. }
61.
62. private String name;
63. private double salary;
64. private Date hireDay;
65. }
```

### Use of Multiple Source Files

The program in Listing 4-2 has two classes in a single source file. Many programmers prefer to put each class into its own source file. For example, you can place the `Employee` class into a file `Employee.java` and the `EmployeeTest` class into `EmployeeTest.java`.

If you like this arrangement, then you have two choices for compiling the program. You can invoke the Java compiler with a wildcard:

```
javac Employee*.java
```

Then, all source files matching the wildcard will be compiled into class files. Or, you can simply type

```
javac EmployeeTest.java
```

You may find it surprising that the second choice works even though the `Employee.java` file is never explicitly compiled. However, when the Java compiler sees the `Employee` class being used inside `EmployeeTest.java`, it will look for a file named `Employee.class`. If it does not find that file, it automatically searches for `Employee.java` and then compiles it. Even more is true: if the time stamp of the version of `Employee.java` that it finds is newer than that of the existing `Employee.class` file, the Java compiler will *automatically* recompile the file.



NOTE: If you are familiar with the “make” facility of UNIX (or one of its Windows cousins such as “nmake”), then you can think of the Java compiler as having the “make” functionality already built in.

### Dissecting the Employee Class

In the sections that follow, we want to dissect the `Employee` class. Let’s start with the methods in this class. As you can see by examining the source code, this class has one constructor and four methods:

```

public Employee(String n, double s, int year, int month, int day)
public String getName()
public double getSalary()
public Date getHireDay()
public void raiseSalary(double byPercent)

```

All methods of this class are tagged as `public`. The keyword `public` means that any method in any class can call the method. (The four possible access levels are covered in this and the next chapter.)

Next, notice that three instance fields will hold the data we will manipulate inside an instance of the `Employee` class.

```

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

```

The `private` keyword makes sure that the *only* methods that can access these instance fields are the methods of the `Employee` class itself. No outside method can read or write to these fields.



**NOTE:** You could use the `public` keyword with your instance fields, but it would be a very bad idea. Having `public` data fields would allow any part of the program to read and modify the instance fields. That completely ruins encapsulation. Any method of any class can modify `public` fields—and, in our experience, some code usually will take advantage of that access privilege when you least expect it. We strongly recommend that you always make your instance fields `private`.

Finally, notice that two of the instance fields are themselves objects: the `name` and `hireDay` fields are references to `String` and `Date` objects. This is quite usual: classes will often contain instance fields of class type.

### **First Steps with Constructors**

Let's look at the constructor listed in our `Employee` class.

```

public Employee(String n, double s, int year, int month, int day)
{
    name = n;
    salary = s;
    GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
    hireDay = calendar.getTime();
}

```

As you can see, the name of the constructor is the same as the name of the class. This constructor runs when you construct objects of the `Employee` class—giving the instance fields the initial state you want them to have.

For example, when you create an instance of the `Employee` class with code like this:

```

new Employee("James Bond", 100000, 1950, 1, 1);

```

you have set the instance fields as follows:

```

name = "James Bond";
salary = 100000;
hireDay = January 1, 1950;

```

There is an important difference between constructors and other methods. A constructor can only be called in conjunction with the `new` operator. You can't apply a constructor to an existing object to reset the instance fields. For example,

```
james.Employee("James Bond", 250000, 1950, 1, 1); // ERROR
```

is a compile-time error.

We have more to say about constructors later in this chapter. For now, keep the following in mind:

- A constructor has the same name as the class.
- A class can have more than one constructor.
- A constructor can take zero, one, or more parameters.
- A constructor has no return value.
- A constructor is always called with the `new` operator.



**C++ NOTE:** Constructors work the same way in Java as they do in C++. But keep in mind that all Java objects are constructed on the heap and that a constructor must be combined with `new`. It is a common C++ programmer error to forget the `new` operator:

```
Employee number007("James Bond", 100000, 1950, 1, 1);
// C++, not Java
```

That works in C++ but does not work in Java.



**CAUTION:** Be careful not to introduce local variables with the same names as the instance fields. For example, the following constructor will not set the salary:

```
public Employee(String n, double s, . . .)
{
    String name = n; // ERROR
    double salary = s; // ERROR
    . . .
}
```

The constructor declares *local* variables `name` and `salary`. These variables are only accessible inside the constructor. They *shadow* the instance fields with the same name. Some programmers—such as the authors of this book—write this kind of code when they type faster than they think, because their fingers are used to adding the data type. This is a nasty error that can be hard to track down. You just have to be careful in all of your methods that you don't use variable names that equal the names of instance fields.

### **Implicit and Explicit Parameters**

Methods operate on objects and access their instance fields. For example, the method

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

sets a new value for the salary instance field in the object on which this method is invoked. Consider the call

```
number007.raiseSalary(5);
```

The effect is to increase the value of the `number007.salary` field by 5%. More specifically, the call executes the following instructions:

```
double raise = number007.salary * 5 / 100;
number007.salary += raise;
```

The `raiseSalary` method has two parameters. The first parameter, called the *implicit* parameter, is the object of type `Employee` that appears before the method name. The second parameter, the number inside the parentheses after the method name, is an *explicit* parameter.

As you can see, the explicit parameters are explicitly listed in the method declaration, for example, `double byPercent`. The implicit parameter does not appear in the method declaration.

In every method, the keyword `this` refers to the implicit parameter. If you like, you can write the `raiseSalary` method as follows:

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

Some programmers prefer that style because it clearly distinguishes between instance fields and local variables.



**C++ NOTE:** In C++, you generally define methods outside the class:

```
void Employee::raiseSalary(double byPercent) // C++, not Java
{
    . . .
}
```

If you define a method inside a class, then it is automatically an inline method.

```
class Employee
{
    . . .
    int getName() { return name; } // inline in C++
}
```

In the Java programming language, all methods are defined inside the class itself. This does not make them inline. Finding opportunities for inline replacement is the job of the Java virtual machine. The just-in-time compiler watches for calls to methods that are short, commonly called, and not overridden, and optimizes them away.

---



### **Benefits of Encapsulation**

Finally, let's look more closely at the rather simple `getName`, `getSalary`, and `getHireDay` methods.

```
public String getName()
{
    return name;
}

public double getSalary()
{
    return salary;
}

public Date getHireDay()
{
    return hireDay;
}
```

These are obvious examples of accessor methods. Because they simply return the values of instance fields, they are sometimes called *field accessors*.

Wouldn't it be easier to simply make the `name`, `salary`, and `hireDay` fields public, instead of having separate accessor methods?

The point is that the `name` field is a read-only field. Once you set it in the constructor, there is no method to change it. Thus, we have a guarantee that the `name` field will never be corrupted.

The `salary` field is not read-only, but it can only be changed by the `raiseSalary` method. In particular, should the value ever be wrong, only that method needs to be debugged. Had the `salary` field been public, the culprit for messing up the value could have been anywhere.

Sometimes, it happens that you want to get and set the value of an instance field. Then you need to supply *three* items:

- A private data field;
- A public field accessor method; and
- A public field mutator method.

This is a lot more tedious than supplying a single public data field, but there are considerable benefits.

First, you can change the internal implementation without affecting any code other than the methods of the class.

For example, if the storage of the name is changed to

```
String firstName;
String lastName;
```

then the `getName` method can be changed to return

```
firstName + " " + lastName
```

This change is completely invisible to the remainder of the program.

Of course, the accessor and mutator methods may need to do a lot of work and convert between the old and the new data representation. But that leads us to our second benefit: Mutator methods can perform error-checking, whereas code that simply assigns to a field may not go to the trouble. For example, a `setSalary` method might check that the salary is never less than 0.



**CAUTION:** Be careful not to write accessor methods that return references to mutable objects. We violated that rule in our `Employee` class in which the `getHireDay` method returns an object of class `Date`:

```
class Employee
{
    . . .
    public Date getHireDay()
    {
        return hireDay;
    }
    . . .
    private Date hireDay;
}
```

This breaks the encapsulation! Consider the following rogue code:

```
Employee harry = . . . ;
Date d = harry.getHireDay();
double tenYearsInMilliseconds = 10 * 365.25 * 24 * 60 * 60 * 1000;
d.setTime(d.getTime() - (long) tenYearsInMilliseconds);
// let's give Harry ten years added seniority
```

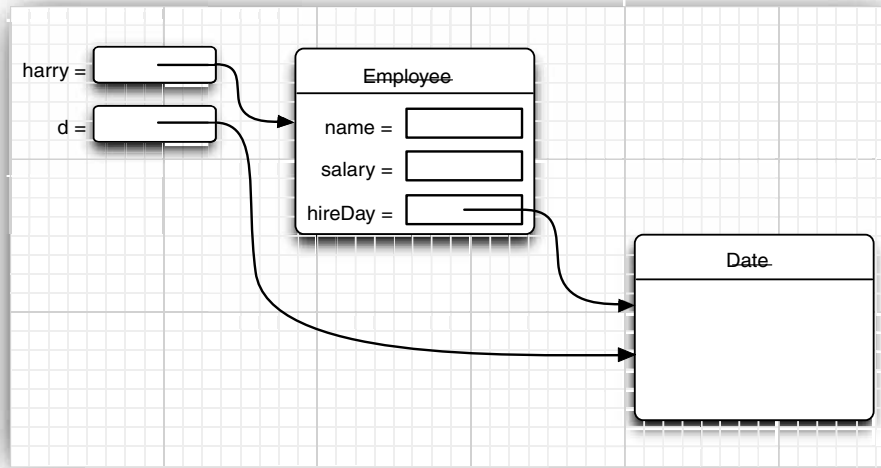
The reason is subtle. Both `d` and `harry.hireDay` refer to the same object (see Figure 4–5).

Applying mutator methods to `d` automatically changes the private state of the employee object!

If you need to return a reference to a mutable object, you should *clone* it first. A clone is an exact copy of an object that is stored in a new location. We discuss cloning in detail in Chapter 6. Here is the corrected code:

```
class Employee
{
    . . .
    public Date getHireDay()
    {
        return (Date) hireDay.clone();
    }
    . . .
}
```

As a rule of thumb, always use `clone` whenever you need to return a copy of a mutable data field.



**Figure 4-5** Returning a reference to a mutable data field

### ***Class-Based Access Privileges***

You know that a method can access the private data of the object on which it is invoked. What many people find surprising is that a method can access the private data of *all* objects of its class. For example, consider a method `equals` that compares two employees.

```
class Employee
{
    . . .
    boolean equals(Employee other)
    {
        return name.equals(other.name);
    }
}
```

A typical call is

```
if (harry.equals(boss)) . . .
```

This method accesses the private fields of `harry`, which is not surprising. It also accesses the private fields of `boss`. This is legal because `boss` is an object of type `Employee`, and a method of the `Employee` class is permitted to access the private fields of *any* object of type `Employee`.



**C++ NOTE:** C++ has the same rule. A method can access the private features of any object of its class, not just of the implicit parameter.

### Private Methods

When implementing a class, we make all data fields private because public data are dangerous. But what about the methods? While most methods are public, private methods are used in certain circumstances. Sometimes, you may wish to break up the code for a computation into separate helper methods. Typically, these helper methods should not become part of the public interface—they may be too close to the current implementation or require a special protocol or calling order. Such methods are best implemented as private.

To implement a private method in Java, simply change the `public` keyword to `private`.

By making a method private, you are under no obligation to keep it available if you change to another implementation. The method may well be *harder* to implement or *unnecessary* if the data representation changes: this is irrelevant. The point is that as long as the method is private, the designers of the class can be assured that it is never used outside the other class operations and can simply drop it. If a method is public, you cannot simply drop it because other code might rely on it.

### Final Instance Fields

You can define an instance field as `final`. Such a field must be initialized when the object is constructed. That is, it must be guaranteed that the field value has been set after the end of every constructor. Afterwards, the field may not be modified again. For example, the `name` field of the `Employee` class may be declared as `final` because it never changes after the object is constructed—there is no `setName` method.

```
class Employee
{
    . . .
    private final String name;
}
```

The `final` modifier is particularly useful for fields whose type is primitive or an *immutable class*. (A class is immutable if none of its methods ever mutate its objects. For example, the `String` class is immutable.) For mutable classes, the `final` modifier is likely to confuse the reader. For example,

```
private final Date hiredate;
```

merely means that the object reference stored in the `hiredate` variable doesn't get changed after the object is constructed. That does not mean that the `hiredate` object is constant. Any method is free to invoke the `setTime` mutator on the object to which `hiredate` refers.

### Static Fields and Methods

In all sample programs that you have seen, the `main` method is tagged with the `static` modifier. We are now ready to discuss the meaning of this modifier.

#### Static Fields

If you define a field as `static`, then there is only one such field per class. In contrast, each object has its own copy of all instance fields. For example, let's suppose we want to assign a unique identification number to each employee. We add an instance field `id` and a static field `nextId` to the `Employee` class:

```
class Employee
{
```

```

    . . .
    private int id;
    private static int nextId = 1;
}

```

Every employee object now has its own `id` field, but there is only one `nextId` field that is shared among all instances of the class. Let's put it another way. If there are 1,000 objects of the `Employee` class, then there are 1,000 instance fields `id`, one for each object. But there is a single static field `nextId`. Even if there are no employee objects, the static field `nextId` is present. It belongs to the class, not to any individual object.



NOTE: In most object-oriented programming languages, static fields are called *class fields*. The term "static" is a meaningless holdover from C++.

Let's implement a simple method:

```

public void setId()
{
    id = nextId;
    nextId++;
}

```

Suppose you set the employee identification number for `harry`:

```
harry.setId();
```

Then, the `id` field of `harry` is set to the current value of the static field `nextId`, and the value of the static field is incremented:

```

harry.id = Employee.nextId;
Employee.nextId++;

```

### Static Constants

Static variables are quite rare. However, static constants are more common. For example, the `Math` class defines a static constant:

```

public class Math
{
    . . .
    public static final double PI = 3.14159265358979323846;
    . . .
}

```

You can access this constant in your programs as `Math.PI`.

If the keyword `static` had been omitted, then `PI` would have been an instance field of the `Math` class. That is, you would need an object of the `Math` class to access `PI`, and every `Math` object would have its own copy of `PI`.

Another static constant that you have used many times is `System.out`. It is declared in the `System` class as follows:

```

public class System
{
    . . .
    public static final PrintStream out = . . . ;
    . . .
}

```

As we mentioned several times, it is never a good idea to have public fields, because everyone can modify them. However, public constants (that is, `final` fields) are fine. Because `out` has been declared as `final`, you cannot reassign another print stream to it:

```
System.out = new PrintStream(. . .); // ERROR--out is final
```



**NOTE:** If you look at the `System` class, you will notice a method `setOut` that lets you set `System.out` to a different stream. You may wonder how that method can change the value of a `final` variable. However, the `setOut` method is a *native* method, not implemented in the Java programming language. Native methods can bypass the access control mechanisms of the Java language. This is a very unusual workaround that you should not emulate in your own programs.

### Static Methods

Static methods are methods that do not operate on objects. For example, the `pow` method of the `Math` class is a static method. The expression

```
Math.pow(x, a)
```

computes the power  $x^a$ . It does not use any `Math` object to carry out its task. In other words, it has no implicit parameter.

You can think of static methods as methods that don't have a `this` parameter. (In a non-static method, the `this` parameter refers to the implicit parameter of the method—see the section “Implicit and Explicit Parameters” on page 127.)

Because static methods don't operate on objects, you cannot access instance fields from a static method. But static methods can access the static fields in their class. Here is an example of such a static method:

```
public static int getNextId()
{
    return nextId; // returns static field
}
```

To call this method, you supply the name of the class:

```
int n = Employee.getNextId();
```

Could you have omitted the keyword `static` for this method? Yes, but then you would need to have an object reference of type `Employee` to invoke the method.



**NOTE:** It is legal to use an object to call a static method. For example, if `harry` is an `Employee` object, then you can call `harry.getNextId()` instead of `Employee.getNextId()`. However, we find that notation confusing. The `getNextId` method doesn't look at `harry` at all to compute the result. We recommend that you use class names, not objects, to invoke static methods.

You use static methods in two situations:

- When a method doesn't need to access the object state because all needed parameters are supplied as explicit parameters (example: `Math.pow`)
- When a method only needs to access static fields of the class (example: `Employee.getNextId`)

**C++** NOTE: Static fields and methods have the same functionality in Java and C++. However, the syntax is slightly different. In C++, you use the `::` operator to access a static field or method outside its scope, such as `Math::PI`.

The term “static” has a curious history. At first, the keyword `static` was introduced in C to denote local variables that don’t go away when a block is exited. In that context, the term “static” makes sense: the variable stays around and is still there when the block is entered again. Then `static` got a second meaning in C, to denote global variables and functions that cannot be accessed from other files. The keyword `static` was simply reused, to avoid introducing a new keyword. Finally, C++ reused the keyword for a third, unrelated, interpretation—to denote variables and functions that belong to a class but not to any particular object of the class. That is the same meaning that the keyword has in Java.

### Factory Methods

Here is another common use for static methods. The `NumberFormat` class uses *factory methods* that yield formatter objects for various styles.

```
NumberFormat currencyFormatter = NumberFormat.getCurrencyInstance();
NumberFormat percentFormatter = NumberFormat.getPercentInstance();
double x = 0.1;
System.out.println(currencyFormatter.format(x)); // prints $0.10
System.out.println(percentFormatter.format(x)); // prints 10%
```

Why doesn’t the `NumberFormat` class use a constructor instead? There are two reasons:

- You can’t give names to constructors. The constructor name is always the same as the class name. But we want two different names to get the currency instance and the percent instance.
- When you use a constructor, you can’t vary the type of the constructed object. But the factory methods actually return objects of the class `DecimalFormat`, a subclass that inherits from `NumberFormat`. (See Chapter 5 for more on inheritance.)

### The main Method

Note that you can call static methods without having any objects. For example, you never construct any objects of the `Math` class to call `Math.pow`.

For the same reason, the `main` method is a static method.

```
public class Application
{
    public static void main(String[] args)
    {
        // construct objects here
        . . .
    }
}
```

The `main` method does not operate on any objects. In fact, when a program starts, there aren’t any objects yet. The static `main` method executes, and constructs the objects that the program needs.



TIP: Every class can have a main method. That is a handy trick for unit testing of classes. For example, you can add a main method to the Employee class:

```
class Employee
{
    public Employee(String n, double s, int year, int month, int day)
    {
        name = n;
        salary = s;
        GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
        hireDay = calendar.getTime();
    }
    . . .
    public static void main(String[] args) // unit test
    {
        Employee e = new Employee("Romeo", 50000, 2003, 3, 31);
        e.raiseSalary(10);
        System.out.println(e.getName() + " " + e.getSalary());
    }
    . . .
}
```

If you want to test the Employee class in isolation, you simply execute

```
java Employee
```

If the employee class is a part of a larger application, then you start the application with

```
java Application
```

and the main method of the Employee class is never executed.

The program in Listing 4-3 contains a simple version of the Employee class with a static field nextId and a static method getNextId. We fill an array with three Employee objects and then print the employee information. Finally, we print the next available identification number, to demonstrate the static method.

Note that the Employee class also has a static main method for unit testing. Try running both

```
java Employee
```

and

```
java StaticTest
```

to execute both main methods.

#### Listing 4-3 StaticTest.java

```
1. /**
2.  * This program demonstrates static methods.
3.  * @version 1.01 2004-02-19
4.  * @author Cay Horstmann
5.  */
6. public class StaticTest
7. {
8.     public static void main(String[] args)
9.     {
```



**Listing 4-3** StaticTest.java (continued)

```
10. // fill the staff array with three Employee objects
11. Employee[] staff = new Employee[3];
12.
13. staff[0] = new Employee("Tom", 40000);
14. staff[1] = new Employee("Dick", 60000);
15. staff[2] = new Employee("Harry", 65000);
16.
17. // print out information about all Employee objects
18. for (Employee e : staff)
19. {
20.     e.setId();
21.     System.out.println("name=" + e.getName() + ",id=" + e.getId() + ",salary="
22.         + e.getSalary());
23. }
24.
25. int n = Employee.getNextId(); // calls static method
26. System.out.println("Next available id=" + n);
27. }
28. }
29.
30. class Employee
31. {
32.     public Employee(String n, double s)
33.     {
34.         name = n;
35.         salary = s;
36.         id = 0;
37.     }
38.
39.     public String getName()
40.     {
41.         return name;
42.     }
43.
44.     public double getSalary()
45.     {
46.         return salary;
47.     }
48.
49.     public int getId()
50.     {
51.         return id;
52.     }
53.
54.     public void setId()
55.     {
56.         id = nextId; // set id to next available id
57.         nextId++;
58.     }
```

**Listing 4-3** StaticTest.java (continued)

```
59.
60. public static int getNextId()
61. {
62.     return nextId; // returns static field
63. }
64.
65. public static void main(String[] args) // unit test
66. {
67.     Employee e = new Employee("Harry", 50000);
68.     System.out.println(e.getName() + " " + e.getSalary());
69. }
70.
71. private String name;
72. private double salary;
73. private int id;
74. private static int nextId = 1;
75. }
```

### Method Parameters

Let us review the computer science terms that describe how parameters can be passed to a method (or a function) in a programming language. The term *call by value* means that the method gets just the value that the caller provides. In contrast, *call by reference* means that the method gets the *location* of the variable that the caller provides. Thus, a method can *modify* the value stored in a variable that is passed by reference but not in one that is passed by value. These “call by . . .” terms are standard computer science terminology that describe the behavior of method parameters in various programming languages, not just Java. (In fact, there is also a *call by name* that is mainly of historical interest, being employed in the Algol programming language, one of the oldest high-level languages.)

The Java programming language *always* uses call by value. That means that the method gets a copy of all parameter values. In particular, the method cannot modify the contents of any parameter variables that are passed to it.

For example, consider the following call:

```
double percent = 10;
harry.raiseSalary(percent);
```

No matter how the method is implemented, we know that after the method call, the value of `percent` is still 10.

Let us look a little more closely at this situation. Suppose a method tried to triple the value of a method parameter:

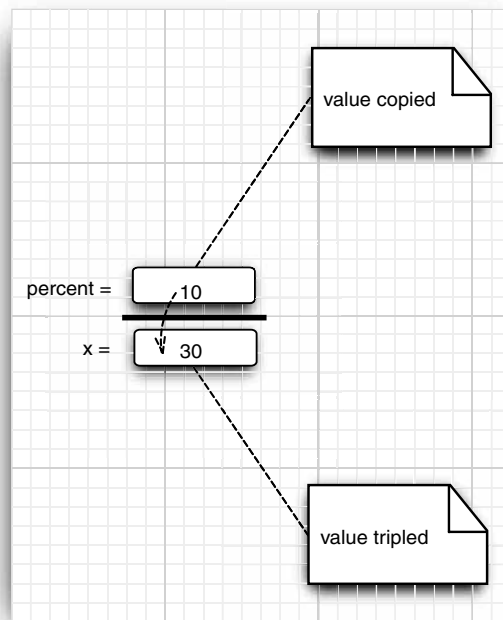
```
public static void tripleValue(double x) // doesn't work
{
    x = 3 * x;
}
```

Let's call this method:

```
double percent = 10;
tripleValue(percent);
```

However, this does not work. After the method call, the value of `percent` is still 10. Here is what happens:

1. `x` is initialized with a copy of the value of `percent` (that is, 10).
2. `x` is tripled—it is now 30. But `percent` is still 10 (see Figure 4–6).
3. The method ends, and the parameter variable `x` is no longer in use.



**Figure 4–6** Modifying a numeric parameter has no lasting effect

There are, however, two kinds of method parameters:

- Primitive types (numbers, boolean values)
- Object references

You have seen that it is impossible for a method to change a primitive type parameter. The situation is different for object parameters. You can easily implement a method that triples the salary of an employee:

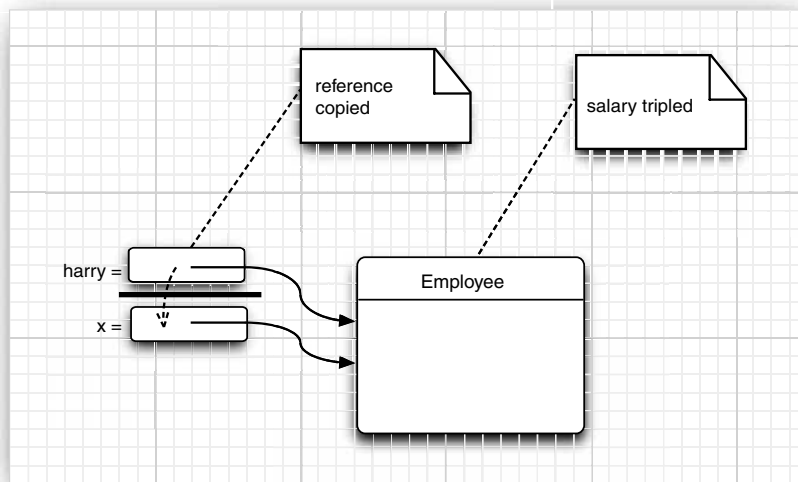
```
public static void tripleSalary(Employee x) // works
{
    x.raiseSalary(200);
}
```

When you call

```
harry = new Employee(. . .);
tripleSalary(harry);
```

then the following happens:

1. `x` is initialized with a copy of the value of `harry`, that is, an object reference.
2. The `raiseSalary` method is applied to that object reference. The `Employee` object to which both `x` and `harry` refer gets its salary raised by 200 percent.
3. The method ends, and the parameter variable `x` is no longer in use. Of course, the object variable `harry` continues to refer to the object whose salary was tripled (see Figure 4-7).



**Figure 4-7** Modifying an object parameter has a lasting effect

As you have seen, it is easily possible—and in fact very common—to implement methods that change the state of an object parameter. The reason is simple. The method gets a copy of the object reference, and both the original and the copy refer to the same object.

Many programming languages (in particular, C++ and Pascal) have two methods for parameter passing: call by value and call by reference. Some programmers (and unfortunately even some book authors) claim that the Java programming language uses call by reference for objects. However, that is false. Because this is such a common misunderstanding, it is worth examining a counterexample in detail.

Let's try to write a method that swaps two employee objects:

```
public static void swap(Employee x, Employee y) // doesn't work
{
    Employee temp = x;
    x = y;
```

```

    y = temp;
}

```

If the Java programming language used call by reference for objects, this method would work:

```

Employee a = new Employee("Alice", . . .);
Employee b = new Employee("Bob", . . .);
swap(a, b);
// does a now refer to Bob, b to Alice?

```

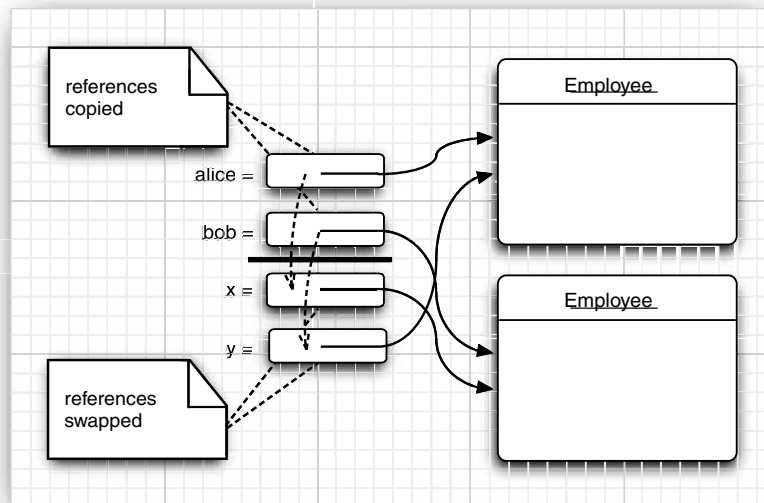
However, the method does not actually change the object references that are stored in the variables `a` and `b`. The `x` and `y` parameters of the `swap` method are initialized with *copies* of these references. The method then proceeds to swap these copies.

```

// x refers to Alice, y to Bob
Employee temp = x;
x = y;
y = temp;
// now x refers to Bob, y to Alice

```

But ultimately, this is a wasted effort. When the method ends, the parameter variables `x` and `y` are abandoned. The original variables `a` and `b` still refer to the same objects as they did before the method call (see Figure 4–8).



**Figure 4–8 Swapping object parameters has no lasting effect**

This discussion demonstrates that the Java programming language does not use call by reference for objects. Instead, *object references are passed by value*.

Here is a summary of what you can and cannot do with method parameters in the Java programming language:

- A method cannot modify a parameter of primitive type (that is, numbers or boolean values).
- A method can change the *state* of an object parameter.
- A method cannot make an object parameter refer to a new object.

The program in Listing 4–4 demonstrates these facts. The program first tries to triple the value of a number parameter and does not succeed:

```
Testing tripleValue:
Before: percent=10.0
End of method: x=30.0
After: percent=10.0
```

It then successfully triples the salary of an employee:

```
Testing tripleSalary:
Before: salary=50000.0
End of method: salary=150000.0
After: salary=150000.0
```

After the method, the state of the object to which `harry` refers has changed. This is possible because the method modified the state through a copy of the object reference.

Finally, the program demonstrates the failure of the `swap` method:

```
Testing swap:
Before: a=Alice
Before: b=Bob
End of method: x=Bob
End of method: y=Alice
After: a=Alice
After: b=Bob
```

As you can see, the parameter variables `x` and `y` are swapped, but the variables `a` and `b` are not affected.



**C++ NOTE:** C++ has both call by value and call by reference. You tag reference parameters with `&`. For example, you can easily implement methods `void tripleValue(double& x)` or `void swap(Employee& x, Employee& y)` that modify their reference parameters.

#### Listing 4–4 ParamTest.java

```
1. /**
2.  * This program demonstrates parameter passing in Java.
3.  * @version 1.00 2000-01-27
4.  * @author Cay Horstmann
5.  */
6. public class ParamTest
7. {
8.     public static void main(String[] args)
9.     {
10.         /*
11.          * Test 1: Methods can't modify numeric parameters
12.          */
```

**Listing 4-4** ParamTest.java (continued)

```
13. System.out.println("Testing tripleValue:");
14. double percent = 10;
15. System.out.println("Before: percent=" + percent);
16. tripleValue(percent);
17. System.out.println("After: percent=" + percent);
18.
19. /*
20.  * Test 2: Methods can change the state of object parameters
21.  */
22. System.out.println("\nTesting tripleSalary:");
23. Employee harry = new Employee("Harry", 50000);
24. System.out.println("Before: salary=" + harry.getSalary());
25. tripleSalary(harry);
26. System.out.println("After: salary=" + harry.getSalary());
27.
28. /*
29.  * Test 3: Methods can't attach new objects to object parameters
30.  */
31. System.out.println("\nTesting swap:");
32. Employee a = new Employee("Alice", 70000);
33. Employee b = new Employee("Bob", 60000);
34. System.out.println("Before: a=" + a.getName());
35. System.out.println("Before: b=" + b.getName());
36. swap(a, b);
37. System.out.println("After: a=" + a.getName());
38. System.out.println("After: b=" + b.getName());
39. }
40.
41. public static void tripleValue(double x) // doesn't work
42. {
43.     x = 3 * x;
44.     System.out.println("End of method: x=" + x);
45. }
46.
47. public static void tripleSalary(Employee x) // works
48. {
49.     x.raiseSalary(200);
50.     System.out.println("End of method: salary=" + x.getSalary());
51. }
52.
53. public static void swap(Employee x, Employee y)
54. {
55.     Employee temp = x;
56.     x = y;
57.     y = temp;
58.     System.out.println("End of method: x=" + x.getName());
59.     System.out.println("End of method: y=" + y.getName());
60. }
61. }
```

**Listing 4-4** ParamTest.java (continued)

```
62.
63. class Employee // simplified Employee class
64. {
65.     public Employee(String n, double s)
66.     {
67.         name = n;
68.         salary = s;
69.     }
70.
71.     public String getName()
72.     {
73.         return name;
74.     }
75.
76.     public double getSalary()
77.     {
78.         return salary;
79.     }
80.
81.     public void raiseSalary(double byPercent)
82.     {
83.         double raise = salary * byPercent / 100;
84.         salary += raise;
85.     }
86.
87.     private String name;
88.     private double salary;
89. }
```

---

## Object Construction

You have seen how to write simple constructors that define the initial state of your objects. However, because object construction is so important, Java offers quite a variety of mechanisms for writing constructors. We go over these mechanisms in the sections that follow.

### Overloading

Recall that the `GregorianCalendar` class had more than one constructor. We could use

```
GregorianCalendar today = new GregorianCalendar();
```

or

```
GregorianCalendar deadline = new GregorianCalendar(2099, Calendar.DECEMBER, 31);
```

This capability is called *overloading*. Overloading occurs if several methods have the same name (in this case, the `GregorianCalendar` constructor method) but different parameters. The compiler must sort out which method to call. It picks the correct method by matching the parameter types in the headers of the various methods with the types of the values used in the specific method call. A compile-time error occurs if the compiler cannot match the parameters or if more than one match is possible. (This process is called *overloading resolution*.)





NOTE: Java allows you to overload any method—not just constructor methods. Thus, to completely describe a method, you need to specify the name of the method together with its parameter types. This is called the *signature* of the method. For example, the `String` class has four public methods called `indexOf`. They have signatures

```
indexOf(int)
indexOf(int, int)
indexOf(String)
indexOf(String, int)
```

The return type is not part of the method signature. That is, you cannot have two methods with the same names and parameter types but different return types.

### Default Field Initialization

If you don't set a field explicitly in a constructor, it is automatically set to a default value: numbers to `0`, boolean values to `false`, and object references to `null`. But it is considered poor programming practice to rely on this. Certainly, it makes it harder for someone to understand your code if fields are being initialized invisibly.



NOTE: This is an important difference between fields and local variables. You must always explicitly initialize local variables in a method. But if you don't initialize a field in a class, it is automatically initialized to a default (`0`, `false`, or `null`).

For example, consider the `Employee` class. Suppose you don't specify how to initialize some of the fields in a constructor. By default, the `salary` field would be initialized with `0` and the `name` and `hireDay` fields would be initialized with `null`.

However, that would not be a good idea. If anyone called the `getName` or `getHireDay` method, then they would get a `null` reference that they probably don't expect:

```
Date h = harry.getHireDay();
calendar.setTime(h); // throws exception if h is null
```

### Default Constructors

A *default constructor* is a constructor with no parameters. For example, here is a default constructor for the `Employee` class:

```
public Employee()
{
    name = "";
    salary = 0;
    hireDay = new Date();
}
```

If you write a class with no constructors whatsoever, then a default constructor is provided for you. This default constructor sets *all* the instance fields to their default values. So, all numeric data contained in the instance fields would be `0`, all boolean values would be `false`, and all object variables would be set to `null`.

If a class supplies at least one constructor but does not supply a default constructor, it is illegal to construct objects without construction parameters. For example, our original `Employee` class in Listing 4-2 provided a single constructor:

```
Employee(String name, double salary, int y, int m, int d)
```

With that class, it was not legal to construct default employees. That is, the call

```
e = new Employee();
```

would have been an error.



CAUTION: Please keep in mind that you get a free default constructor *only* when your class has no other constructors. If you write your class with even a single constructor of your own and you want the users of your class to have the ability to create an instance by a call to

```
new ClassName()
```

then you must provide a default constructor (with no parameters). Of course, if you are happy with the default values for all fields, you can simply supply

```
public ClassName()
{
}
```

### **Explicit Field Initialization**

Because you can overload the constructor methods in a class, you can obviously build in many ways to set the initial state of the instance fields of your classes. It is always a good idea to make sure that, regardless of the constructor call, every instance field is set to something meaningful.

You can simply assign a value to any field in the class definition. For example:

```
class Employee
{
    . . .
    private String name = "";
}
```

This assignment is carried out before the constructor executes. This syntax is particularly useful if all constructors of a class need to set a particular instance field to the same value.

The initialization value doesn't have to be a constant value. Here is an example in which a field is initialized with a method call. Consider an `Employee` class where each employee has an `id` field. You can initialize it as follows:

```
class Employee
{
    . . .
    static int assignId()
    {
        int r = nextId;
        nextId++;
        return r;
    }
    . . .
    private int id = assignId();
}
```

**C++** NOTE: In C++, you cannot directly initialize instance fields of a class. All fields must be set in a constructor. However, C++ has a special initializer list syntax, such as

```
Employee::Employee(String n, double s, int y, int m, int d) // C++
: name(n),
  salary(s),
  hireDay(y, m, d)
{
}
```

C++ uses this special syntax to call field constructors. In Java, there is no need for it because objects have no subobjects, only pointers to other objects.

### Parameter Names

When you write very trivial constructors (and you'll write a lot of them), then it can be somewhat frustrating to come up with parameter names.

We have generally opted for single-letter parameter names:

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

However, the drawback is that you need to read the code to tell what `n` and `s` parameters mean.

Some programmers prefix each parameter with an "a":

```
public Employee(String aName, double aSalary)
{
    name = aName;
    salary = aSalary;
}
```

That is quite neat. Any reader can immediately figure out the meaning of the parameters.

Another commonly used trick relies on the fact that parameter variables *shadow* instance fields with the same name. For example, if you call a parameter `salary`, then `salary` refers to the parameter, not the instance field. But you can still access the instance field as `this.salary`. Recall that `this` denotes the implicit parameter, that is, the object that is being constructed. Here is an example:

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

**C++** NOTE: In C++, it is common to prefix instance fields with an underscore or a fixed letter. (The letters `m` and `x` are common choices.) For example, the `salary` field might be called `_salary`, `mSalary`, or `xSalary`. Java programmers don't usually do that.

### Calling Another Constructor

The keyword `this` refers to the implicit parameter of a method. However, the keyword has a second meaning.

If the *first statement of a constructor* has the form `this(. . .)`, then the constructor calls another constructor of the same class. Here is a typical example:

```
public Employee(double s)
{
    // calls Employee(String, double)
    this("Employee #" + nextId, s);
    nextId++;
}
```

When you call `new Employee(60000)`, then the `Employee(double)` constructor calls the `Employee(String, double)` constructor.

Using the `this` keyword in this manner is useful—you only need to write common construction code once.



**C++ NOTE:** The `this` reference in Java is identical to the `this` pointer in C++. However, in C++ it is not possible for one constructor to call another. If you want to factor out common initialization code in C++, you must write a separate method.

### Initialization Blocks

You have already seen two ways to initialize a data field:

- By setting a value in a constructor
- By assigning a value in the declaration

There is actually a *third* mechanism in Java; it's called an *initialization block*. Class declarations can contain arbitrary blocks of code. These blocks are executed whenever an object of that class is constructed. For example:

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

    public Employee()
    {
        name = "";
        salary = 0;
    }
    . . .
    private static int nextId;

    private int id;
    private String name;
    private double salary;
    . . .
}
```

```
// object initialization block
{
    id = nextId;
    nextId++;
}
```

In this example, the `id` field is initialized in the object initialization block, no matter which constructor is used to construct an object. The initialization block runs first, and then the body of the constructor is executed.

This mechanism is never necessary and is not common. It usually is more straightforward to place the initialization code inside a constructor.



**NOTE:** It is legal to set fields in initialization blocks even though they are only defined later in the class. Some versions of Sun's Java compiler handled this case incorrectly (bug # 4459133). This bug has been fixed in Java SE 1.4.1. However, to avoid circular definitions, it is not legal to read from fields that are only initialized later. The exact rules are spelled out in section 8.3.2.3 of the Java Language Specification (<http://java.sun.com/docs/books/jls>). Because the rules are complex enough to baffle the compiler implementors, we suggest that you place initialization blocks after the field definitions.

With so many ways of initializing data fields, it can be quite confusing to give all possible pathways for the construction process. Here is what happens in detail when a constructor is called:

1. All data fields are initialized to their default value (0, false, or null).
2. All field initializers and initialization blocks are executed, in the order in which they occur in the class declaration.
3. If the first line of the constructor calls a second constructor, then the body of the second constructor is executed.
4. The body of the constructor is executed.

Naturally, it is always a good idea to organize your initialization code so that another programmer could easily understand it without having to be a language lawyer. For example, it would be quite strange and somewhat error prone to have a class whose constructors depend on the order in which the data fields are declared.

You initialize a static field either by supplying an initial value or by using a static initialization block. You have already seen the first mechanism:

```
static int nextId = 1;
```

If the static fields of your class require complex initialization code, use a static initialization block.

Place the code inside a block and tag it with the keyword `static`. Here is an example. We want the employee ID numbers to start at a random integer less than 10,000.

```
// static initialization block
static
{
    Random generator = new Random();
    nextId = generator.nextInt(10000);
}
```

Static initialization occurs when the class is first loaded. Like instance fields, static fields are 0, false, or null unless you explicitly set them to another value. All static field initializers and static initialization blocks are executed in the order in which they occur in the class declaration.



NOTE: Here is a Java trivia fact to amaze your fellow Java coders: You can write a “Hello, World” program in Java without ever writing a main method.

```
public class Hello
{
    static
    {
        System.out.println("Hello, World");
    }
}
```

When you invoke the class with `java Hello`, the class is loaded, the static initialization block prints “Hello, World,” and only then do you get an ugly error message that main is not defined. You can avoid that blemish by calling `System.exit(0)` at the end of the static initialization block.

The program in Listing 4–5 shows many of the features that we discussed in this section:

- Overloaded constructors
- A call to another constructor with `this(...)`
- A default constructor
- An object initialization block
- A static initialization block
- An instance field initialization

**Listing 4–5** ConstructorTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates object construction.
5.  * @version 1.01 2004-02-19
6.  * @author Cay Horstmann
7.  */
8. public class ConstructorTest
9. {
10.     public static void main(String[] args)
11.     {
12.         // fill the staff array with three Employee objects
13.         Employee[] staff = new Employee[3];
14.
15.         staff[0] = new Employee("Harry", 40000);
16.         staff[1] = new Employee(60000);
17.         staff[2] = new Employee();
18.
```

**Listing 4-5** ConstructorTest.java (continued)

```
19. // print out information about all Employee objects
20. for (Employee e : staff)
21.     System.out.println("name=" + e.getName() + ",id=" + e.getId() + ",salary="
22.         + e.getSalary());
23. }
24. }
25.
26. class Employee
27. {
28.     // three overloaded constructors
29.     public Employee(String n, double s)
30.     {
31.         name = n;
32.         salary = s;
33.     }
34.
35.     public Employee(double s)
36.     {
37.         // calls the Employee(String, double) constructor
38.         this("Employee #" + nextId, s);
39.     }
40.
41.     // the default constructor
42.     public Employee()
43.     {
44.         // name initialized to ""--see below
45.         // salary not explicitly set--initialized to 0
46.         // id initialized in initialization block
47.     }
48.
49.     public String getName()
50.     {
51.         return name;
52.     }
53.
54.     public double getSalary()
55.     {
56.         return salary;
57.     }
58.
59.     public int getId()
60.     {
61.         return id;
62.     }
63.
64.     private static int nextId;
65.
66.     private int id;
67.     private String name = ""; // instance field initialization
68.     private double salary;
```

**Listing 4-5** ConstructorTest.java (continued)

```
69.
70. // static initialization block
71. static
72. {
73.     Random generator = new Random();
74.     // set nextId to a random number between 0 and 9999
75.     nextId = generator.nextInt(10000);
76. }
77.
78. // object initialization block
79. {
80.     id = nextId;
81.     nextId++;
82. }
83. }
```

**API** java.util.Random 1.0

- Random()  
constructs a new random number generator.
- int nextInt(int n) 1.2  
returns a random number between 0 and  $n - 1$ .

**Object Destruction and the finalize Method**

Some object-oriented programming languages, notably C++, have explicit destructor methods for any cleanup code that may be needed when an object is no longer used. The most common activity in a destructor is reclaiming the memory set aside for objects. Because Java does automatic garbage collection, manual memory reclamation is not needed and so Java does not support destructors.

Of course, some objects utilize a resource other than memory, such as a file or a handle to another object that uses system resources. In this case, it is important that the resource be reclaimed and recycled when it is no longer needed.

You can add a `finalize` method to any class. The `finalize` method will be called before the garbage collector sweeps away the object. In practice, *do not rely on the `finalize` method* for recycling any resources that are in short supply—you simply cannot know when this method will be called.



**NOTE:** The method call `System.runFinalizersOnExit(true)` guarantees that finalizer methods are called before Java shuts down. However, this method is inherently unsafe and has been deprecated. An alternative is to add “shutdown hooks” with the method `Runtime.addShutdownHook`—see the API documentation for details.

If a resource needs to be closed as soon as you have finished using it, you need to manage it manually. Supply a method such as `dispose` or `close` that *you* call to clean up what



needs cleaning. Just as importantly, if a class you use has such a method, you need to call it when you are done with the object.

### Packages

Java allows you to group classes in a collection called a *package*. Packages are convenient for organizing your work and for separating your work from code libraries provided by others.

The standard Java library is distributed over a number of packages, including `java.lang`, `java.util`, `java.net`, and so on. The standard Java packages are examples of hierarchical packages. Just as you have nested subdirectories on your hard disk, you can organize packages by using levels of nesting. All standard Java packages are inside the `java` and `javax` package hierarchies.

The main reason for using packages is to guarantee the uniqueness of class names. Suppose two programmers come up with the bright idea of supplying an `Employee` class. As long as both of them place their class into different packages, there is no conflict. In fact, to absolutely guarantee a unique package name, Sun recommends that you use your company's Internet domain name (which is known to be unique) written in reverse. You then use subpackages for different projects. For example, `horstmann.com` is a domain that one of the authors registered. Written in reverse order, it turns into the package `com.horstmann`. That package can then be further subdivided into subpackages such as `com.horstmann.corejava`.

From the point of view of the compiler, there is absolutely no relationship between nested packages. For example, the packages `java.util` and `java.util.jar` have nothing to do with each other. Each is its own independent collection of classes.

### Class Importation

A class can use all classes from its own package and all *public* classes from other packages.

You can access the public classes in another package in two ways. The first is simply to add the full package name in front of *every* class name. For example:

```
java.util.Date today = new java.util.Date();
```

That is obviously tedious. The simpler, and more common, approach is to use the `import` statement. The point of the `import` statement is simply to give you a shorthand to refer to the classes in the package. Once you use `import`, you no longer have to give the classes their full names.

You can import a specific class or the whole package. You place `import` statements at the top of your source files (but below any `package` statements). For example, you can import all classes in the `java.util` package with the statement

```
import java.util.*;
```

Then you can use

```
Date today = new Date();
```

without a package prefix. You can also import a specific class inside a package:

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

The `java.util.*` syntax is less tedious. It has no negative effect on code size. However, if you import classes explicitly, the reader of your code knows exactly which classes you use.



TIP: In Eclipse, you can select the menu option Source -> Organize Imports. Package statements such as `import java.util.*;` are automatically expanded into a list of specific imports such as

```
import java.util.ArrayList;
import java.util.Date;
```

This is an extremely convenient feature.

However, note that you can only use the `*` notation to import a single package. You cannot use `import java.*` or `import java.*.*` to import all packages with the `java` prefix.

Most of the time, you just import the packages that you need, without worrying too much about them. The only time that you need to pay attention to packages is when you have a name conflict. For example, both the `java.util` and `java.sql` packages have a `Date` class. Suppose you write a program that imports both packages.

```
import java.util.*;
import java.sql.*;
```

If you now use the `Date` class, then you get a compile-time error:

```
Date today; // ERROR--java.util.Date or java.sql.Date?
```

The compiler cannot figure out which `Date` class you want. You can solve this problem by adding a specific `import` statement:

```
import java.util.*;
import java.sql.*;
import java.util.Date;
```

What if you really need both `Date` classes? Then you need to use the full package name with every class name.

```
java.util.Date deadline = new java.util.Date();
java.sql.Date today = new java.sql.Date(...);
```

Locating classes in packages is an activity of the *compiler*. The bytecodes in class files always use full package names to refer to other classes.



C++ NOTE: C++ programmers usually confuse `import` with `#include`. The two have nothing in common. In C++, you must use `#include` to include the declarations of external features because the C++ compiler does not look inside any files except the one that it is compiling and explicitly included header files. The Java compiler will happily look inside other files provided you tell it where to look.

In Java, you can entirely avoid the `import` mechanism by explicitly naming all classes, such as `java.util.Date`. In C++, you cannot avoid the `#include` directives.

The only benefit of the `import` statement is convenience. You can refer to a class by a name shorter than the full package name. For example, after an `import java.util.*` (or `import java.util.Date`) statement, you can refer to the `java.util.Date` class simply as `Date`.

The analogous construction to the package mechanism in C++ is the namespace feature. Think of the package and `import` statements in Java as the analogs of the namespace and `using` directives in C++.

### Static Imports

Starting with Java SE 5.0, the `import` statement has been enhanced to permit the importing of static methods and fields, not just classes.

For example, if you add the directive

```
import static java.lang.System.*;
```

to the top of your source file, then you can use static methods and fields of the `System` class without the class name prefix:

```
out.println("Goodbye, World!"); // i.e., System.out
exit(0); // i.e., System.exit
```

You can also import a specific method or field:

```
import static java.lang.System.out;
```

In practice, it seems doubtful that many programmers will want to abbreviate `System.out` or `System.exit`. The resulting code seems less clear. But there are two practical uses for static imports.

- **Mathematical functions:** If you use a static import for the `Math` class, you can use mathematical functions in a more natural way. For example,

```
sqrt(pow(x, 2) + pow(y, 2))
```

seems much clearer than

```
Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2))
```

- **Cumbersome constants:** If you use lots of constants with tedious names, you will welcome static import. For example,

```
if (d.get(DAY_OF_WEEK) == MONDAY)
```

is easier on the eye than

```
if (d.get(Calendar.DAY_OF_WEEK) == Calendar.MONDAY)
```

### Addition of a Class into a Package

To place classes inside a package, you must put the name of the package at the top of your source file, *before* the code that defines the classes in the package. For example, the file `Employee.java` in Listing 4–7 starts out like this:

```
package com.horstmann.corejava;

public class Employee
{
    . . .
}
```

If you don't put a package statement in the source file, then the classes in that source file belong to the *default package*. The default package has no package name. Up to now, all our example classes were located in the default package.

You place source files into a subdirectory that matches the full package name. For example, all source files in the package `com.horstmann.corejava` package should be in a subdirectory `com/horstmann/corejava` (`com\horstmann\corejava` on Windows). The compiler places the class files into the same directory structure.

The program in Listings 4–6 and 4–7 is distributed over two packages: the `PackageTest` class belongs to the default package and the `Employee` class belongs to the `com.horstmann.corejava` package. Therefore, the `Employee.java` file must be contained in a subdirectory `com/horstmann/corejava`. In other words, the directory structure is as follows:

```

. (base directory)
├── PackageTest.java
├── PackageTest.class
└── com/
    └── horstmann/
        └── corejava/
            ├── Employee.java
            └── Employee.class

```

To compile this program, simply change to the base directory and run the command  
`javac PackageTest.java`

The compiler automatically finds the file `com/horstmann/corejava/Employee.java` and compiles it.

Let's look at a more realistic example, in which we don't use the default package but have classes distributed over several packages (`com.horstmann.corejava` and `com.mycompany`).

```

. (base directory)
└── com/
    ├── horstmann/
    │   ├── corejava/
    │   │   ├── Employee.java
    │   │   └── Employee.class
    └── mycompany/
        ├── PayrollApp.java
        └── PayrollApp.class

```

In this situation, you still must compile and run classes from the *base* directory, that is, the directory containing the `com` directory:

```

javac com/mycompany/PayrollApp.java
java com.mycompany.PayrollApp

```

Note again that the compiler operates on *files* (with file separators and an extension `.java`), whereas the Java interpreter loads a *class* (with dot separators).



**CAUTION:** The compiler does *not* check the directory structure when it compiles source files. For example, suppose you have a source file that starts with the directive

```
package com.mycompany;
```

You can compile the file even if it is not contained in a subdirectory `com/mycompany`. The source file will compile without errors *if it doesn't depend on other packages*. However, the resulting program will not run. The *virtual machine* won't find the resulting classes when you try to run the program.

**Listing 4-6** PackageTest.java

```
1. import com.horstmann.corejava.*;
2. // the Employee class is defined in that package
3.
4. import static java.lang.System.*;
5.
6. /**
7.  * This program demonstrates the use of packages.
8.  * @author cay
9.  * @version 1.11 2004-02-19
10. * @author Cay Horstmann
11. */
12. public class PackageTest
13. {
14.     public static void main(String[] args)
15.     {
16.         // because of the import statement, we don't have to use com.horstmann.corejava.Employee here
17.         Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
18.
19.         harry.raiseSalary(5);
20.
21.         // because of the static import statement, we don't have to use System.out here
22.         out.println("name=" + harry.getName() + ",salary=" + harry.getSalary());
23.     }
24. }
```

**Listing 4-7** Employee.java

```
1. package com.horstmann.corejava;
2.
3. // the classes in this file are part of this package
4.
5. import java.util.*;
6.
7. // import statements come after the package statement
8.
9. /**
10. * @version 1.10 1999-12-18
11. * @author Cay Horstmann
12. */
13. public class Employee
14. {
15.     public Employee(String n, double s, int year, int month, int day)
16.     {
17.         name = n;
18.         salary = s;
19.         GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
```

**Listing 4-7** Employee.java (continued)

```
20.    // GregorianCalendar uses 0 for January
21.    hireDay = calendar.getTime();
22.    }
23.
24.    public String getName()
25.    {
26.        return name;
27.    }
28.
29.    public double getSalary()
30.    {
31.        return salary;
32.    }
33.
34.    public Date getHireDay()
35.    {
36.        return hireDay;
37.    }
38.
39.    public void raiseSalary(double byPercent)
40.    {
41.        double raise = salary * byPercent / 100;
42.        salary += raise;
43.    }
44.
45.    private String name;
46.    private double salary;
47.    private Date hireDay;
48. }
```

**Package Scope**

You have already encountered the access modifiers `public` and `private`. Features tagged as `public` can be used by any class. Private features can be used only by the class that defines them. If you don't specify either `public` or `private`, the feature (that is, the class, method, or variable) can be accessed by all methods in the same *package*.

Consider the program in Listing 4-2 on page 124. The `Employee` class was not defined as a `public` class. Therefore, only other classes in the same package—the default package in this case—such as `EmployeeTest` can access it. For classes, this is a reasonable default. However, for variables, this default was an unfortunate choice. Variables must explicitly be marked

`private` or they will default to being package visible. This, of course, breaks encapsulation. The problem is that it is awfully easy to forget to type the `private` keyword. Here is an example from the `Window` class in the `java.awt` package, which is part of the source code supplied with the JDK:

```
public class Window extends Container
{
    String warningString;
    . . .
}
```

Note that the `warningString` variable is not `private`! That means the methods of all classes in the `java.awt` package can access this variable and set it to whatever they like (such as "Trust me!"). Actually, the only methods that access this variable are in the `Window` class, so it would have been entirely appropriate to make the variable `private`. We suspect that the programmer typed the code in a hurry and simply forgot the `private` modifier. (We won't mention the programmer's name to protect the guilty—you can look into the source file yourself.)



**NOTE:** Amazingly enough, this problem has never been fixed, even though we have pointed it out in eight editions of this book—apparently the library implementors don't read *Core Java*. Not only that—new fields have been added to the class over time, and about half of them aren't `private` either.

Is this really a problem? It depends. By default, packages are not closed entities. That is, anyone can add more classes to a package. Of course, hostile or clueless programmers can then add code that modifies variables with package visibility. For example, in early versions of the Java programming language, it was an easy matter to smuggle another class into the `java.awt` package. Simply start out the class with

```
package java.awt;
```

Then, place the resulting class file inside a subdirectory `java/awt` somewhere on the class path, and you have gained access to the internals of the `java.awt` package. Through this subterfuge, it was possible to set the warning string (see Figure 4–9).



**Figure 4–9** Changing the warning string in an applet window

Starting with version 1.2, the JDK implementors rigged the class loader to explicitly disallow loading of user-defined classes whose package name starts with "java. ". Of course, your own classes won't benefit from that protection. Instead, you can use another mechanism, *package sealing*, to address the issue of promiscuous package access. If you seal a package, no further classes can be added to it. You will see in Chapter 10 how you can produce a JAR file that contains sealed packages.

## The Class Path

As you have seen, classes are stored in subdirectories of the file system. The path to the class must match the package name.

Class files can also be stored in a JAR (Java archive) file. A JAR file contains multiple class files and subdirectories in a compressed format, saving space and improving performance. When you use a third-party library in your programs, you will usually be given one or more JAR files to include. The JDK also supplies a number of JAR files, such as the file `jre/lib/rt.jar` that contains thousands of library classes. You will see in Chapter 10 how to create your own JAR files.



**TIP:** JAR files use the ZIP format to organize files and subdirectories. You can use any ZIP utility to peek inside `rt.jar` and other JAR files.

To share classes among programs, you need to do the following:

1. Place your class files inside a directory, for example, `/home/user/classdir`. Note that this directory is the *base* directory for the package tree. If you add the class `com.horstmann.corejava.Employee`, then the `Employee.class` file must be located in the subdirectory `/home/user/classdir/com/horstmann/corejava`.
2. Place any JAR files inside a directory, for example, `/home/user/archives`.
3. Set the *class path*. The class path is the collection of all locations that can contain class files.

On UNIX, the elements on the class path are separated by colons:

```
/home/user/classdir:./home/user/archives/archive.jar
```

On Windows, they are separated by semicolons:

```
c:\classdir;.;c:\archives\archive.jar
```

In both cases, the period denotes the current directory.

This class path contains

- The base directory `/home/user/classdir` or `c:\classdir`;
- The current directory (`.`); and
- The JAR file `/home/user/archives/archive.jar` or `c:\archives\archive.jar`.

Starting with Java SE 6, you can specify a wildcard for a JAR file directory, like this:

```
/home/user/classdir:./home/user/archives/*
```

or

```
c:\classdir;.;c:\archives\*
```

In UNIX, the `*` must be escaped to prevent shell expansion.

All JAR files (but not `.class` files) in the archives directory are included in this class path.

The runtime library files (`rt.jar` and the other JAR files in the `jre/lib` and `jre/lib/ext` directories) are always searched for classes; you don't include them explicitly in the class path.





**CAUTION:** The `javac` compiler always looks for files in the current directory, but the java virtual machine launcher only looks into the current directory if the `."` directory is on the class path. If you have no class path set, this is not a problem—the default class path consists of the `."` directory. But if you have set the class path and forgot to include the `."` directory, your programs will compile without error, but they won't run.

The class path lists all directories and archive files that are *starting points* for locating classes. Let's consider our sample class path:

```
/home/user/classdir:./home/user/archives/archive.jar
```

Suppose the virtual machine searches for the class file of the `com.horstmann.corejava.Employee` class. It first looks in the system class files that are stored in archives in the `jre/lib` and `jre/lib/ext` directories. It won't find the class file there, so it turns to the class path. It then looks for the following files:

- `/home/user/classdir/com/horstmann/corejava/Employee.class`
- `com/horstmann/corejava/Employee.class` starting from the current directory
- `com/horstmann/corejava/Employee.class` inside `/home/user/archives/archive.jar`

The compiler has a harder time locating files than does the virtual machine. If you refer to a class without specifying its package, the compiler first needs to find out the package that contains the class. It consults all `import` directives as possible sources for the class. For example, suppose the source file contains directives

```
import java.util.*;
import com.horstmann.corejava.*;
```

and the source code refers to a class `Employee`. The compiler then tries to find `java.lang.Employee` (because the `java.lang` package is always imported by default), `java.util.Employee`, `com.horstmann.corejava.Employee`, and `Employee` in the current package. It searches for *each* of these classes in all of the locations of the class path. It is a compile-time error if more than one class is found. (Because classes must be unique, the order of the `import` statements doesn't matter.)

The compiler goes one step further. It looks at the *source files* to see if the source is newer than the class file. If so, the source file is recompiled automatically. Recall that you can import only public classes from other packages. A source file can only contain one public class, and the names of the file and the public class must match. Therefore, the compiler can easily locate source files for public classes. However, you can import nonpublic classes from the current package. These classes may be defined in source files with different names. If you import a class from the current package, the compiler searches *all* source files of the current package to see which one defines the class.

### Setting the Class Path

It is best to specify the class path with the `-classpath` (or `-cp`) option:

```
java -classpath /home/user/classdir:./home/user/archives/archive.jar MyProg.java
```

or

```
java -classpath c:\classdir;.;c:\archives\archive.jar MyProg.java
```

The entire command must be typed onto a single line. It is a good idea to place such a long command line into a shell script or a batch file.

Using the `-classpath` option is the preferred approach for setting the class path. An alternate approach is the `CLASSPATH` environment variable. The details depend on your shell. With the Bourne Again shell (bash), use the command

```
export CLASSPATH=/home/user/classdir:./home/user/archives/archive.jar
```

With the C shell, use the command

```
setenv CLASSPATH /home/user/classdir:./home/user/archives/archive.jar
```

With the Windows shell, use

```
set CLASSPATH=c:\classdir;.;c:\archives\archive.jar
```

The class path is set until the shell exits.



**CAUTION:** Some people recommend to set the `CLASSPATH` environment variable permanently. This is generally a bad idea. People forget the global setting, and then they are surprised when their classes are not loaded properly. A particularly reprehensible example is Apple's QuickTime installer in Windows. It globally sets `CLASSPATH` to point to a JAR file that it needs, but it does not include the current directory in the classpath. As a result, countless Java programmers have been driven to distraction when their programs compiled but failed to run.



**CAUTION:** Some people recommend to bypass the class path altogether, by dropping all JAR files into the `jre/lib/ext` directory. That is truly bad advice, for two reasons. Archives that manually load other classes do not work correctly when they are placed in the extension directory. (See Volume II, Chapter 9 for more information on class loaders.) Moreover, programmers have a tendency to forget about the files they placed there months ago. Then, they scratch their heads when the class loader seems to ignore their carefully crafted class path, when it is actually loading long-forgotten classes from the extension directory.

## Documentation Comments

The JDK contains a very useful tool, called `javadoc`, that generates HTML documentation from your source files. In fact, the on-line API documentation that we described in Chapter 3 is simply the result of running `javadoc` on the source code of the standard Java library.

If you add comments that start with the special delimiter `/**` to your source code, you too can easily produce professional-looking documentation. This is a very nice scheme because it lets you keep your code and documentation in one place. If you put your documentation into a separate file, then you probably know that the code and comments tend to diverge over time. But because the documentation comments are in the same file as the source code, it is an easy matter to update both and run `javadoc` again.

### Comment Insertion

The `javadoc` utility extracts information for the following items:

- Packages
- Public classes and interfaces
- Public and protected methods
- Public and protected fields

Protected features are introduced in Chapter 5, interfaces in Chapter 6.

You can (and should) supply a comment for each of these features. Each comment is placed immediately *above* the feature it describes. A comment starts with a `/**` and ends with a `*/`.

Each `/** . . . */` documentation comment contains *free-form text* followed by *tags*. A tag starts with an `@`, such as `@author` or `@param`.

The *first sentence* of the free-form text should be a *summary statement*. The javadoc utility automatically generates summary pages that extract these sentences.

In the free-form text, you can use HTML modifiers such as `<em>...</em>` for emphasis, `<code>...</code>` for a monospaced “typewriter” font, `<strong>...</strong>` for strong emphasis, and even `<img ...>` to include an image. You should, however, stay away from headings `<h1>` or rules `<hr>` because they can interfere with the formatting of the document.



NOTE: If your comments contain links to other files such as images (for example, diagrams or images of user interface components), place those files into a subdirectory of the directory containing the source file named `doc-files`. The javadoc utility will copy the `doc-files` directories and their contents from the source directory to the documentation directory. You need to use the `doc-files` directory in your link, such as ``.

### Class Comments

The class comment must be placed *after* any `import` statements, directly before the class definition.

Here is an example of a class comment:

```
/**
 * A <code>Card</code> object represents a playing card, such
 * as "Queen of Hearts". A card has a suit (Diamond, Heart,
 * Spade or Club) and a value (1 = Ace, 2 . . . 10, 11 = Jack,
 * 12 = Queen, 13 = King)
 */
public class Card
{
    . . .
}
```



NOTE: There is no need to add an `*` in front of every line. For example, the following comment is equally valid:

```
/**
    A <code>Card</code> object represents a playing card, such
    as "Queen of Hearts". A card has a suit (Diamond, Heart,
    Spade or Club) and a value (1 = Ace, 2 . . . 10, 11 = Jack,
    12 = Queen, 13 = King).
*/
```

However, most IDEs supply the asterisks automatically and rearrange them when the line breaks change.

**Method Comments**

Each method comment must immediately precede the method that it describes. In addition to the general-purpose tags, you can use the following tags:

- `@param variable description`  
This tag adds an entry to the “parameters” section of the current method. The description can span multiple lines and can use HTML tags. All `@param` tags for one method must be kept together.
- `@return description`  
This tag adds a “returns” section to the current method. The description can span multiple lines and can use HTML tags.
- `@throws class description`  
This tag adds a note that this method may throw an exception. Exceptions are the topic of Chapter 11.

Here is an example of a method comment:

```
/**
 * Raises the salary of an employee.
 * @param byPercent the percentage by which to raise the salary (e.g. 10 = 10%)
 * @return the amount of the raise
 */
public double raiseSalary(double byPercent)
{
    double raise = salary * byPercent / 100;
    salary += raise;
    return raise;
}
```

**Field Comments**

You only need to document public fields—generally that means static constants. For example:

```
/**
 * The "Hearts" card suit
 */
public static final int HEARTS = 1;
```

**General Comments**

The following tags can be used in class documentation comments:

- `@author name`  
This tag makes an “author” entry. You can have multiple `@author` tags, one for each author.
- `@version text`  
This tag makes a “version” entry. The text can be any description of the current version.

The following tags can be used in all documentation comments:

- `@since text`  
This tag makes a “since” entry. The text can be any description of the version that introduced this feature. For example, `@since version 1.7.1`

- `@deprecated text`

This tag adds a comment that the class, method, or variable should no longer be used. The text should suggest a replacement. For example:

```
@deprecated Use <code>setVisible(true)</code> instead
```

You can use hyperlinks to other relevant parts of the javadoc documentation, or to external documents, with the `@see` and `@link` tags.

- `@see reference`

This tag adds a hyperlink in the “see also” section. It can be used with both classes and methods. Here, *reference* can be one of the following:

```
package.class#feature label
<a href="...">label</a>
"text"
```

The first case is the most useful. You supply the name of a class, method, or variable, and javadoc inserts a hyperlink to the documentation. For example,

```
@see com.horstmann.corejava.Employee#raiseSalary(double)
```

makes a link to the `raiseSalary(double)` method in the `com.horstmann.corejava.Employee` class. You can omit the name of the package or both the package and class name. Then, the feature will be located in the current package or class.

Note that you must use a #, not a period, to separate the class from the method or variable name. The Java compiler itself is highly skilled in guessing the various meanings of the period character, as separator between packages, subpackages, classes, inner classes, and methods and variables. But the javadoc utility isn't quite as clever, and you have to help it along.

If the `@see` tag is followed by a < character, then you need to specify a hyperlink. You can link to any URL you like. For example:

```
@see <a href="www.horstmann.com/corejava.html">The Core Java home page</a>
```

In each of these cases, you can specify an optional *label* that will appear as the link anchor. If you omit the label, then the user will see the target code name or URL as the anchor.

If the `@see` tag is followed by a " character, then the text is displayed in the “see also” section. For example:

```
@see "Core Java 2 volume 2"
```

You can add multiple `@see` tags for one feature, but you must keep them all together.

- If you like, you can place hyperlinks to other classes or methods anywhere in any of your documentation comments. You insert a special tag of the form

```
{@link package.class#feature label}
```

anywhere in a comment. The feature description follows the same rules as for the `@see` tag.

### **Package and Overview Comments**

You place class, method, and variable comments directly into the Java source files, delimited by `/** . . . */` documentation comments. However, to generate package comments, you need to add a separate file in each package directory. You have two choices:

1. Supply an HTML file named `package.html`. All text between the tags `<body>...</body>` is extracted.
2. Supply a Java file named `package-info.java`. The file must contain an initial Javadoc comment, delimited with `/**` and `*/`, followed by a package statement. It should contain no further code or comments.

You can also supply an overview comment for all source files. Place it in a file called `overview.html`, located in the parent directory that contains all the source files. All text between the tags `<body>...</body>` is extracted. This comment is displayed when the user selects “Overview” from the navigation bar.

### Comment Extraction

Here, *docDirectory* is the name of the directory where you want the HTML files to go. Follow these steps:

1. Change to the directory that contains the source files you want to document. If you have nested packages to document, such as `com.horstmann.corejava`, you must be working in the directory that contains the subdirectory `com`. (This is the directory that contains the `overview.html` file if you supplied one.)

2. Run the command

```
javadoc -d docDirectory nameOfPackage
```

for a single package. Or run

```
javadoc -d docDirectory nameOfPackage1 nameOfPackage2...
```

to document multiple packages. If your files are in the default package, then instead run

```
javadoc -d docDirectory *.java
```

If you omit the `-d docDirectory` option, then the HTML files are extracted to the current directory. That can get messy, and we don’t recommend it.

The `javadoc` program can be fine-tuned by numerous command-line options. For example, you can use the `-author` and `-version` options to include the `@author` and `@version` tags in the documentation. (By default, they are omitted.) Another useful option is `-link`, to include hyperlinks to standard classes. For example, if you use the command

```
javadoc -link http://java.sun.com/javase/6/docs/api *.java
```

all standard library classes are automatically linked to the documentation on the Sun web site.

If you use the `-linksource` option, each source file is converted to HTML (without color coding, but with line numbers), and each class and method name turns into a hyperlink to the source.

For additional options, we refer you to the on-line documentation of the `javadoc` utility at <http://java.sun.com/javase/javadoc>.



NOTE: If you require further customization, for example, to produce documentation in a format other than HTML, you can supply your own *doclet* to generate the output in any form you desire. Clearly, this is a specialized need, and we refer you to the on-line documentation for details on doclets at <http://java.sun.com/j2se/javadoc>.



TIP: A useful doclet is DocCheck at <http://java.sun.com/j2se/javadoc/doccheck/>. It scans a set of source files for missing documentation comments.

### Class Design Hints

Without trying to be comprehensive or tedious, we want to end this chapter with some hints that may make your classes more acceptable in well-mannered OOP circles.

1. *Always keep data private.*

This is first and foremost: doing anything else violates encapsulation. You may need to write an accessor or mutator method occasionally, but you are still better off keeping the instance fields private. Bitter experience has shown that how the data are represented may change, but how they are used will change much less frequently. When data are kept private, changes in their representation do not affect the user of the class, and bugs are easier to detect.

2. *Always initialize data.*

Java won't initialize local variables for you, but it will initialize instance fields of objects. Don't rely on the defaults, but initialize the variables explicitly, either by supplying a default or by setting defaults in all constructors.

3. *Don't use too many basic types in a class.*

The idea is to replace multiple *related* uses of basic types with other classes. This keeps your classes easier to understand and to change. For example, replace the following instance fields in a `Customer` class

```
private String street;  
private String city;  
private String state;  
private int zip;
```

with a new class called `Address`. This way, you can easily cope with changes to addresses, such as the need to deal with international addresses.

4. *Not all fields need individual field accessors and mutators.*

You may need to get and set an employee's salary. You certainly won't need to change the hiring date once the object is constructed. And, quite often, objects have instance fields that you don't want others to get or set, for example, an array of state abbreviations in an `Address` class.

5. *Use a standard form for class definitions.*

We always list the contents of classes in the following order:

```
public features  
package scope features  
private features
```

Within each section, we list:

```
instance methods  
static methods  
instance fields  
static fields
```

After all, the users of your class are more interested in the public interface than in the details of the private implementation. And they are more interested in methods than in data.

However, there is no universal agreement on what is the best style. The Sun coding style guide for the Java programming language recommends listing fields first and then methods. Whatever style you use, the most important thing is to be consistent.

6. *Break up classes that have too many responsibilities.*

This hint is, of course, vague: “too many” is obviously in the eye of the beholder.

However, if there is an obvious way to make one complicated class into two classes that are conceptually simpler, seize the opportunity. (On the other hand, don’t go overboard; 10 classes, each with only one method, is usually overkill.)

Here is an example of a bad design.

```
public class CardDeck // bad design
{
    public CardDeck() { . . . }
    public void shuffle() { . . . }
    public int getTopValue() { . . . }
    public int getTopSuit() { . . . }
    public void draw() { . . . }

    private int[] value;
    private int[] suit;
}
```

This class really implements two separate concepts: a *deck of cards*, with its `shuffle` and `draw` methods, and a *card*, with the methods to inspect the value and suit of a card. It makes sense to introduce a `Card` class that represents an individual card.

Now you have two classes, each with its own responsibilities:

```
public class CardDeck
{
    public CardDeck() { . . . }
    public void shuffle() { . . . }
    public Card getTop() { . . . }
    public void draw() { . . . }

    private Card[] cards;
}

public class Card
{
    public Card(int aValue, int aSuit) { . . . }
    public int getValue() { . . . }
    public int getSuit() { . . . }

    private int value;
    private int suit;
}
```



7. *Make the names of your classes and methods reflect their responsibilities.*

Just as variables should have meaningful names that reflect what they represent, so should classes. (The standard library certainly contains some dubious examples, such as the `Date` class that describes time.)

A good convention is that a class name should be a noun (`Order`) or a noun preceded by an adjective (`RushOrder`) or a gerund (an “-ing” word, like `BillingAddress`). As for methods, follow the standard convention that accessor methods begin with a lowercase `get` (`getSalary`), and that mutator methods use a lowercase `set` (`setSalary`).

In this chapter, we covered the fundamentals of objects and classes that make Java an “object-based” language. In order to be truly object oriented, a programming language must also support inheritance and polymorphism. The Java support for these features is the topic of the next chapter.



---

# *Chapter*

# 5

## INHERITANCE

- ▼ CLASSES, SUPERCLASSES, AND SUBCLASSES
- ▼ **Object**: THE COSMIC SUPERCLASS
- ▼ GENERIC ARRAY LISTS
- ▼ OBJECT WRAPPERS AND AUTOBOXING
- ▼ METHODS WITH A VARIABLE NUMBER OF PARAMETERS
- ▼ ENUMERATION CLASSES
- ▼ REFLECTION
- ▼ DESIGN HINTS FOR INHERITANCE

Chapter 4 introduced you to classes and objects. In this chapter, you learn about *inheritance*, another fundamental concept of object-oriented programming. The idea behind inheritance is that you can create new classes that are built on existing classes. When you inherit from an existing class, you reuse (or inherit) its methods and fields and you add new methods and fields to adapt your new class to new situations. This technique is essential in Java programming.

As with the previous chapter, if you are coming from a procedure-oriented language like C, Visual Basic, or COBOL, you will want to read this chapter carefully. For experienced C++ programmers or those coming from another object-oriented language like Smalltalk, this chapter will seem largely familiar, but there are many differences between how inheritance is implemented in Java and how it is done in C++ or in other object-oriented languages.

This chapter also covers *reflection*, the ability to find out more about classes and their properties in a running program. Reflection is a powerful feature, but it is undeniably complex. Because reflection is of greater interest to tool builders than to application programmers, you can probably glance over that part of the chapter upon first reading and come back to it later.

### Classes, Superclasses, and Subclasses

Let's return to the `Employee` class that we discussed in the previous chapter. Suppose (alas) you work for a company at which managers are treated differently from other employees. Managers are, of course, just like employees in many respects. Both employees and managers are paid a salary. However, while employees are expected to complete their assigned tasks in return for receiving their salary, managers get *bonuses* if they actually achieve what they are supposed to do. This is the kind of situation that cries out for inheritance. Why? Well, you need to define a new class, `Manager`, and add functionality. But you can retain some of what you have already programmed in the `Employee` class, and *all* the fields of the original class can be preserved. More abstractly, there is an obvious "is-a" relationship between `Manager` and `Employee`. Every manager *is an* employee: This "is-a" relationship is the hallmark of inheritance.

Here is how you define a `Manager` class that inherits from the `Employee` class. You use the Java keyword `extends` to denote inheritance.

```
class Manager extends Employee
{
    added methods and fields
}
```

---

**C++** NOTE: Inheritance is similar in Java and C++. Java uses the `extends` keyword instead of the `:` token. All inheritance in Java is public inheritance; there is no analog to the C++ features of private and protected inheritance.

---

The keyword `extends` indicates that you are making a new class that derives from an existing class. The existing class is called the *superclass*, *base class*, or *parent class*. The new class is called the *subclass*, *derived class*, or *child class*. The terms superclass and subclass are those most commonly used by Java programmers, although some programmers prefer the parent/child analogy, which also ties in nicely with the "inheritance" theme.

The `Employee` class is a superclass, but not because it is superior to its subclass or contains more functionality. *In fact, the opposite is true:* subclasses have *more* functionality than their superclasses. For example, as you will see when we go over the rest of the `Manager` class code, the `Manager` class encapsulates more data and has more functionality than its superclass `Employee`.



NOTE: The prefixes *super* and *sub* come from the language of sets used in theoretical computer science and mathematics. The set of all employees contains the set of all managers, and this is described by saying it is a *superset* of the set of managers. Or, put it another way, the set of all managers is a *subset* of the set of all employees.

Our `Manager` class has a new field to store the bonus, and a new method to set it:

```
class Manager extends Employee
{
    . . .

    public void setBonus(double b)
    {
        bonus = b;
    }

    private double bonus;
}
```

There is nothing special about these methods and fields. If you have a `Manager` object, you can simply apply the `setBonus` method.

```
Manager boss = . . . ;
boss.setBonus(5000);
```

Of course, if you have an `Employee` object, you cannot apply the `setBonus` method—it is not among the methods that are defined in the `Employee` class.

However, you *can* use methods such as `getName` and `getHireDay` with `Manager` objects. Even though these methods are not explicitly defined in the `Manager` class, they are automatically inherited from the `Employee` superclass.

Similarly, the fields `name`, `salary`, and `hireDay` are inherited from the superclass. Every `Manager` object has four fields: `name`, `salary`, `hireDay`, and `bonus`.

When defining a subclass by extending its superclass, you only need to indicate the *differences* between the subclass and the superclass. When designing classes, you place the most general methods into the superclass and more specialized methods in the subclass. Factoring out common functionality by moving it to a superclass is common in object-oriented programming.

However, some of the superclass methods are not appropriate for the `Manager` subclass. In particular, the `getSalary` method should return the sum of the base salary and the bonus. You need to supply a new method to *override* the superclass method:

```
class Manager extends Employee
{
    . . .
```

```

    public double getSalary()
    {
        . . .
    }
    . . .
}

```

How can you implement this method? At first glance, it appears to be simple—just return the sum of the salary and bonus fields:

```

    public double getSalary()
    {
        return salary + bonus; // won't work
    }

```

However, that won't work. The `getSalary` method of the `Manager` class *has no direct access to the private fields of the superclass*. This means that the `getSalary` method of the `Manager` class cannot directly access the `salary` field, even though every `Manager` object has a field called `salary`. Only the methods of the `Employee` class have access to the private fields. If the `Manager` methods want to access those private fields, they have to do what every other method does—use the public interface, in this case, the public `getSalary` method of the `Employee` class.

So, let's try this again. You need to call `getSalary` instead of simply accessing the `salary` field.

```

    public double getSalary()
    {
        double baseSalary = getSalary(); // still won't work
        return baseSalary + bonus;
    }

```

The problem is that the call to `getSalary` simply calls *itself*, because the `Manager` class has a `getSalary` method (namely, the method we are trying to implement). The consequence is an infinite set of calls to the same method, leading to a program crash.

We need to indicate that we want to call the `getSalary` method of the `Employee` superclass, not the current class. You use the special keyword `super` for this purpose. The call

```

    super.getSalary()

```

calls the `getSalary` method of the `Employee` class. Here is the correct version of the `getSalary` method for the `Manager` class:

```

    public double getSalary()
    {
        double baseSalary = super.getSalary();
        return baseSalary + bonus;
    }

```



**NOTE:** Some people think of `super` as being analogous to the `this` reference. However, that analogy is not quite accurate—`super` is not a reference to an object. For example, you cannot assign the value `super` to another object variable. Instead, `super` is a special keyword that directs the compiler to invoke the superclass method.

As you saw, a subclass can *add* fields, and it can *add* or *override* methods of the superclass. However, inheritance can never take away any fields or methods.

---

**C++** NOTE: Java uses the keyword `super` to call a superclass method. In C++, you would use the name of the superclass with the `::` operator instead. For example, the `getSalary` method of the `Manager` class would call `Employee::getSalary` instead of `super.getSalary`.

---

Finally, let us supply a constructor.

```
public Manager(String n, double s, int year, int month, int day)
{
    super(n, s, year, month, day);
    bonus = 0;
}
```

Here, the keyword `super` has a different meaning. The instruction


```
super(n, s, year, month, day);
```

is shorthand for “call the constructor of the `Employee` superclass with `n`, `s`, `year`, `month`, and `day` as parameters.”

Because the `Manager` constructor cannot access the private fields of the `Employee` class, it must initialize them through a constructor. The constructor is invoked with the special `super` syntax. The call using `super` must be the first statement in the constructor for the subclass.

If the subclass constructor does not call a superclass constructor explicitly, then the default (no-parameter) constructor of the superclass is invoked. If the superclass has no default constructor and the subclass constructor does not call another superclass constructor explicitly, then the Java compiler reports an error.

---

 NOTE: Recall that the `this` keyword has two meanings: to denote a reference to the implicit parameter and to call another constructor of the same class. Likewise, the `super` keyword has two meanings: to invoke a superclass method and to invoke a superclass constructor. When used to invoke constructors, the `this` and `super` keywords are closely related. The constructor calls can only occur as the first statement in another constructor. The construction parameters are either passed to another constructor of the same class (`this`) or a constructor of the superclass (`super`).

---

**C++** NOTE: In a C++ constructor, you do not call `super`, but you use the initializer list syntax to construct the superclass. The `Manager` constructor looks like this in C++:

```
Manager::Manager(String n, double s, int year, int month, int day) // C++
: Employee(n, s, year, month, day)
{
    bonus = 0;
}
```

---

Having redefined the `getSalary` method for `Manager` objects, managers will *automatically* have the bonus added to their salaries.

Here's an example of this at work: we make a new manager and set the manager's bonus:

```
Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
boss.setBonus(5000);
```

We make an array of three employees:

```
Employee[] staff = new Employee[3];
```

We populate the array with a mix of managers and employees:

```
staff[0] = boss;
staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
```

We print out everyone's salary:

```
for (Employee e : staff)
    System.out.println(e.getName() + " " + e.getSalary());
```

This loop prints the following data:

```
Carl Cracker 85000.0
Harry Hacker 50000.0
Tommy Tester 40000.0
```

Now `staff[1]` and `staff[2]` each print their base salary because they are `Employee` objects. However, `staff[0]` is a `Manager` object and its `getSalary` method adds the bonus to the base salary.

What is remarkable is that the call

```
e.getSalary()
```

picks out the *correct* `getSalary` method. Note that the *declared* type of `e` is `Employee`, but the *actual* type of the object to which `e` refers can be either `Employee` or `Manager`.

When `e` refers to an `Employee` object, then the call `e.getSalary()` calls the `getSalary` method of the `Employee` class. However, when `e` refers to a `Manager` object, then the `getSalary` method of the `Manager` class is called instead. The virtual machine knows about the actual type of the object to which `e` refers, and therefore can invoke the correct method.

The fact that an object variable (such as the variable `e`) can refer to multiple actual types is called *polymorphism*. Automatically selecting the appropriate method at runtime is called *dynamic binding*. We discuss both topics in more detail in this chapter.



**C++ NOTE:** In Java, you do not need to declare a method as virtual. Dynamic binding is the default behavior. If you do *not* want a method to be virtual, you tag it as `final`. (We discuss the `final` keyword later in this chapter.)

Listing 5-1 contains a program that shows how the salary computation differs for `Employee` and `Manager` objects.



**Listing 5-1** ManagerTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates inheritance.
5.  * @version 1.21 2004-02-21
6.  * @author Cay Horstmann
7.  */
8. public class ManagerTest
9. {
10.     public static void main(String[] args)
11.     {
12.         // construct a Manager object
13.         Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
14.         boss.setBonus(5000);
15.
16.         Employee[] staff = new Employee[3];
17.
18.         // fill the staff array with Manager and Employee objects
19.
20.         staff[0] = boss;
21.         staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
22.         staff[2] = new Employee("Tommy Tester", 40000, 1990, 3, 15);
23.
24.         // print out information about all Employee objects
25.         for (Employee e : staff)
26.             System.out.println("name=" + e.getName() + ", salary=" + e.getSalary());
27.     }
28. }
29.
30. class Employee
31. {
32.     public Employee(String n, double s, int year, int month, int day)
33.     {
34.         name = n;
35.         salary = s;
36.         GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
37.         hireDay = calendar.getTime();
38.     }
39.
40.     public String getName()
41.     {
42.         return name;
43.     }
44.
45.     public double getSalary()
46.     {
47.         return salary;
48.     }
49. }
```

**Listing 5-1** ManagerTest.java (continued)

```
50. public Date getHireDay()
51. {
52.     return hireDay;
53. }
54.
55. public void raiseSalary(double byPercent)
56. {
57.     double raise = salary * byPercent / 100;
58.     salary += raise;
59. }
60.
61. private String name;
62. private double salary;
63. private Date hireDay;
64. }
65.
66. class Manager extends Employee
67. {
68.     /**
69.      * @param n the employee's name
70.      * @param s the salary
71.      * @param year the hire year
72.      * @param month the hire month
73.      * @param day the hire day
74.      */
75.     public Manager(String n, double s, int year, int month, int day)
76.     {
77.         super(n, s, year, month, day);
78.         bonus = 0;
79.     }
80.
81.     public double getSalary()
82.     {
83.         double baseSalary = super.getSalary();
84.         return baseSalary + bonus;
85.     }
86.
87.     public void setBonus(double b)
88.     {
89.         bonus = b;
90.     }
91.
92.     private double bonus;
93. }
```

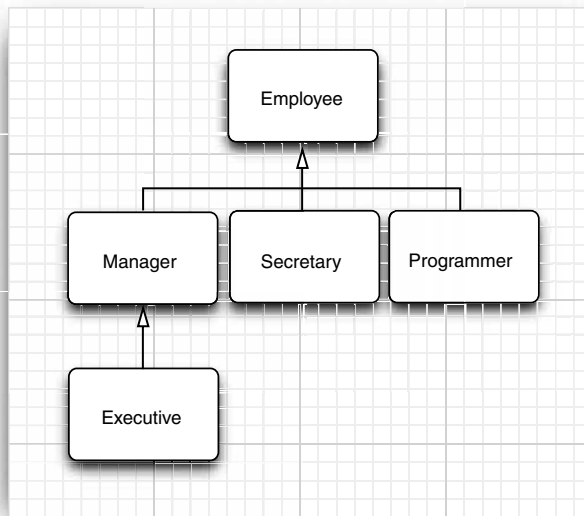
---

### **Inheritance Hierarchies**

Inheritance need not stop at deriving one layer of classes. We could have an `Executive` class that extends `Manager`, for example. The collection of all classes extending from a common superclass is called an *inheritance hierarchy*, as shown in Figure 5–1. The path from a particular class to its ancestors in the inheritance hierarchy is its *inheritance chain*.

There is usually more than one chain of descent from a distant ancestor class. You could form a subclass `Programmer` or `Secretary` that extends `Employee`, and they would have nothing to do with the `Manager` class (or with each other). This process can continue as long as is necessary.

**C++** NOTE: Java does not support multiple inheritance. (For ways to recover much of the functionality of multiple inheritance, see the section on Interfaces in the next chapter.)



**Figure 5–1** Employee inheritance hierarchy

### **Polymorphism**

A simple rule enables you to know whether or not inheritance is the right design for your data. The “is-a” rule states that every object of the subclass is an object of the superclass. For example, every manager is an employee. Thus, it makes sense for the `Manager` class to be a subclass of the `Employee` class. Naturally, the opposite is not true—not every employee is a manager.

Another way of formulating the “is-a” rule is the *substitution principle*. That principle states that you can use a subclass object whenever the program expects a superclass object.

For example, you can assign a subclass object to a superclass variable.

```
Employee e;
e = new Employee(. . .); // Employee object expected
e = new Manager(. . .); // OK, Manager can be used as well
```

In the Java programming language, object variables are *polymorphic*. A variable of type `Employee` can refer to an object of type `Employee` or to an object of any subclass of the `Employee` class (such as `Manager`, `Executive`, `Secretary`, and so on).

We took advantage of this principle in Listing 5-1:

```
Manager boss = new Manager(. . .);
Employee[] staff = new Employee[3];
staff[0] = boss;
```

In this case, the variables `staff[0]` and `boss` refer to the same object. However, `staff[0]` is considered to be only an `Employee` object by the compiler.

That means, you can call

```
boss.setBonus(5000); // OK
```

but you can't call

```
staff[0].setBonus(5000); // ERROR
```

The declared type of `staff[0]` is `Employee`, and the `setBonus` method is not a method of the `Employee` class.

However, you cannot assign a superclass reference to a subclass variable. For example, it is not legal to make the assignment

```
Manager m = staff[i]; // ERROR
```

The reason is clear: Not all employees are managers. If this assignment were to succeed and `m` were to refer to an `Employee` object that is not a manager, then it would later be possible to call `m.setBonus(...)` and a runtime error would occur.



**CAUTION:** In Java, arrays of subclass references can be converted to arrays of superclass references without a cast. For example, consider this array of managers:

```
Manager[] managers = new Manager[10];
```

It is legal to convert this array to an `Employee[]` array:

```
Employee[] staff = managers; // OK
```

Sure, why not, you may think. After all, if `manager[i]` is a `Manager`, it is also an `Employee`. But actually, something surprising is going on. Keep in mind that `managers` and `staff` are references to the same array. Now consider the statement

```
staff[0] = new Employee("Harry Hacker", ...);
```

The compiler will cheerfully allow this assignment. But `staff[0]` and `manager[0]` are the same reference, so it looks as if we managed to smuggle a mere employee into the management ranks. That would be very bad—calling `managers[0].setBonus(1000)` would try to access a nonexistent instance field and would corrupt neighboring memory.

To make sure no such corruption can occur, all arrays remember the element type with which they were created, and they monitor that only compatible references are stored into them. For example, the array created as `new Manager[10]` remembers that it is an array of managers. Attempting to store an `Employee` reference causes an `ArrayStoreException`.

**Dynamic Binding**

It is important to understand what happens when a method call is applied to an object. Here are the details:

1. The compiler looks at the declared type of the object and the method name. Let's say we call `x.f(param)`, and the implicit parameter `x` is declared to be an object of class `C`. Note that there may be multiple methods, all with the same name, `f`, but with different parameter types. For example, there may be a method `f(int)` and a method `f(String)`. The compiler enumerates all methods called `f` in the class `C` and all `public` methods called `f` in the superclasses of `C`.  
Now the compiler knows all possible candidates for the method to be called.
2. Next, the compiler determines the types of the parameters that are supplied in the method call. If among all the methods called `f` there is a unique method whose parameter types are a best match for the supplied parameters, then that method is chosen to be called. This process is called *overloading resolution*. For example, in a call `x.f("Hello")`, the compiler picks `f(String)` and not `f(int)`. The situation can get complex because of type conversions (`int` to `double`, `Manager` to `Employee`, and so on). If the compiler cannot find any method with matching parameter types or if multiple methods all match after applying conversions, then the compiler reports an error.  
Now the compiler knows the name and parameter types of the method that needs to be called.



**NOTE:** Recall that the name and parameter type list for a method is called the method's *signature*. For example, `f(int)` and `f(String)` are two methods with the same name but different signatures. If you define a method in a subclass that has the same signature as a superclass method, then you override that method.

The return type is not part of the signature. However, when you override a method, you need to keep the return type compatible. Prior to Java SE 5.0, the return types had to be identical. However, it is now legal for the subclass to change the return type of an overridden method to a subtype of the original type. For example, suppose that the `Employee` class has a

```
public Employee getBuddy() { ... }
```

Then the `Manager` subclass can override this method as

```
public Manager getBuddy() { ... } // OK in Java SE 5.0
```

We say that the two `getBuddy` methods have *covariant* return types.

3. If the method is `private`, `static`, `final`, or a constructor, then the compiler knows exactly which method to call. (The `final` modifier is explained in the next section.) This is called *static binding*. Otherwise, the method to be called depends on the actual type of the implicit parameter, and dynamic binding must be used at runtime.
4. When the program runs and uses dynamic binding to call a method, then the virtual machine must call the version of the method that is appropriate for the *actual* type of the object to which `x` refers. Let's say the actual type is `D`, a subclass of `C`. If the class `D`

defines a method `f(String)`, that method is called. If not, `D`'s superclass is searched for a method `f(String)`, and so on.

It would be time consuming to carry out this search every time a method is called. Therefore, the virtual machine precomputes for each class a *method table* that lists all method signatures and the actual methods to be called. When a method is actually called, the virtual machine simply makes a table lookup. In our example, the virtual machine consults the method table for the class `D` and looks up the method to call for `f(String)`. That method may be `D.f(String)` or `X.f(String)`, where `X` is some superclass of `D`. There is one twist to this scenario. If the call is `super.f(param)`, then the compiler consults the method table of the superclass of the implicit parameter.

Let's look at this process in detail in the call `e.getSalary()` in Listing 5-1. The declared type of `e` is `Employee`. The `Employee` class has a single method, called `getSalary`, with no method parameters. Therefore, in this case, we don't worry about overloading resolution.

Because the `getSalary` method is not private, static, or final, it is dynamically bound. The virtual machine produces method tables for the `Employee` and `Manager` classes. The `Employee` table shows that all methods are defined in the `Employee` class itself:

```
Employee:
  getName() -> Employee.getName()
  getSalary() -> Employee.getSalary()
  getHireDay() -> Employee.getHireDay()
  raiseSalary(double) -> Employee.raiseSalary(double)
```

Actually, that isn't the whole story—as you will see later in this chapter, the `Employee` class has a superclass `Object` from which it inherits a number of methods. We ignore the `Object` methods for now.

The `Manager` method table is slightly different. Three methods are inherited, one method is redefined, and one method is added.

```
Manager:
  getName() -> Employee.getName()
  getSalary() -> Manager.getSalary()
  getHireDay() -> Employee.getHireDay()
  raiseSalary(double) -> Employee.raiseSalary(double)
  setBonus(double) -> Manager.setBonus(double)
```

At runtime, the call `e.getSalary()` is resolved as follows:

1. First, the virtual machine fetches the method table for the actual type of `e`. That may be the table for `Employee`, `Manager`, or another subclass of `Employee`.
2. Then, the virtual machine looks up the defining class for the `getSalary()` signature. Now it knows which method to call.
3. Finally, the virtual machine calls the method.

Dynamic binding has a very important property: it makes programs *extensible* without the need for modifying existing code. Suppose a new class `Executive` is added and there is the possibility that the variable `e` refers to an object of that class. The code containing the call `e.getSalary()` need not be recompiled. The `Executive.getSalary()` method is called automatically if `e` happens to refer to an object of type `Executive`.



CAUTION: When you override a method, the subclass method must be *at least as visible* as the superclass method. In particular, if the superclass method is `public`, then the subclass method must also be declared as `public`. It is a common error to accidentally omit the `public` specifier for the subclass method. The compiler then complains that you try to supply a weaker access privilege.

### Preventing Inheritance: Final Classes and Methods

Occasionally, you want to prevent someone from forming a subclass from one of your classes. Classes that cannot be extended are called *final* classes, and you use the `final` modifier in the definition of the class to indicate this. For example, let us suppose we want to prevent others from subclassing the `Executive` class. Then, we simply declare the class by using the `final` modifier as follows:

```
final class Executive extends Manager
{
    . . .
}
```

You can also make a specific method in a class `final`. If you do this, then no subclass can override that method. (All methods in a `final` class are automatically `final`.) For example:

```
class Employee
{
    . . .
    public final String getName()
    {
        return name;
    }
    . . .
}
```



NOTE: Recall that fields can also be declared as `final`. A `final` field cannot be changed after the object has been constructed. However, if a class is declared as `final`, only the methods, not the fields, are automatically `final`.

There is only one good reason to make a method or class `final`: to make sure that the semantics cannot be changed in a subclass. For example, the `getTime` and `setTime` methods of the `Calendar` class are `final`. This indicates that the designers of the `Calendar` class have taken over responsibility for the conversion between the `Date` class and the calendar state. No subclass should be allowed to mess up this arrangement. Similarly, the `String` class is a `final` class. That means nobody can define a subclass of `String`. In other words, if you have a `String` reference, then you know it refers to a `String` and nothing but a `String`.

Some programmers believe that you should declare all methods as `final` unless you have a good reason that you want polymorphism. In fact, in C++ and C#, methods do not use polymorphism unless you specifically request it. That may be a bit extreme, but we agree that it is a good idea to think carefully about `final` methods and classes when you design a class hierarchy.

In the early days of Java, some programmers used the `final` keyword in the hope of avoiding the overhead of dynamic binding. If a method is not overridden, and it is short, then a compiler can optimize the method call away—a process called *inlining*. For example, inlining the call `e.getName()` replaces it with the field access `e.name`. This is a worthwhile improvement—CPUs hate branching because it interferes with their strategy of prefetching instructions while processing the current one. However, if `getName` can be overridden in another class, then the compiler cannot inline it because it has no way of knowing what the overriding code may do.

Fortunately, the just-in-time compiler in the virtual machine can do a better job than a traditional compiler. It knows exactly which classes extend a given class, and it can check whether any class actually overrides a given method. If a method is short, frequently called, and not actually overridden, the just-in-time compiler can inline the method. What happens if the virtual machine loads another subclass that overrides an inlined method? Then the optimizer must undo the inlining. That's slow, but it happens rarely.



**C++ NOTE:** In C++, a method is not dynamically bound by default, and you can tag it as `inline` to have method calls replaced with the method source code. However, there is no mechanism that would prevent a subclass from overriding a superclass method. In C++, you can write classes from which no other class can derive, but doing so requires an obscure trick, and there are few reasons to write such a class. (The obscure trick is left as an exercise to the reader. Hint: Use a virtual base class.)

### Casting

Recall from Chapter 3 that the process of forcing a conversion from one type to another is called casting. The Java programming language has a special notation for casts. For example,

```
double x = 3.405;
int nx = (int) x;
```

converts the value of the expression `x` into an integer, discarding the fractional part.

Just as you occasionally need to convert a floating-point number to an integer, you also need to convert an object reference from one class to another. To actually make a cast of an object reference, you use a syntax similar to what you use for casting a numeric expression. Surround the target class name with parentheses and place it before the object reference you want to cast. For example:

```
Manager boss = (Manager) staff[0];
```

There is only one reason why you would want to make a cast—to use an object in its full capacity after its actual type has been temporarily forgotten. For example, in the `ManagerTest` class, the `staff` array had to be an array of `Employee` objects because *some* of its entries were regular employees. We would need to cast the managerial elements of the array back to `Manager` to access any of its new variables. (Note that in the sample code for the first section, we made a special effort to avoid the cast. We initialized the `boss` variable with a `Manager` object before storing it in the array. We needed the correct type to set the bonus of the manager.)



As you know, in Java every object variable has a type. The type describes the kind of object the variable refers to and what it can do. For example, `staff[i]` refers to an `Employee` object (so it can also refer to a `Manager` object).

The compiler checks that you do not promise too much when you store a value in a variable. If you assign a subclass reference to a superclass variable, you are promising less, and the compiler will simply let you do it. If you assign a superclass reference to a subclass variable, you are promising more. Then you must use a cast so that your promise can be checked at runtime.

What happens if you try to cast down an inheritance chain and you are “lying” about what an object contains?

```
Manager boss = (Manager) staff[1]; // ERROR
```

When the program runs, the Java runtime system notices the broken promise and generates a `ClassCastException`. If you do not catch the exception, your program terminates. Thus, it is good programming practice to find out whether a cast will succeed before attempting it. Simply use the `instanceof` operator. For example:

```
if (staff[1] instanceof Manager)
{
    boss = (Manager) staff[1];
    . . .
}
```

Finally, the compiler will not let you make a cast if there is no chance for the cast to succeed. For example, the cast

```
Date c = (Date) staff[1];
```

is a compile-time error because `Date` is not a subclass of `Employee`.

To sum up:

- You can cast only within an inheritance hierarchy.
- Use `instanceof` to check before casting from a superclass to a subclass.



**NOTE:** The test

```
x instanceof C
```

does not generate an exception if `x` is `null`. It simply returns `false`. That makes sense. Because `null` refers to no object, it certainly doesn't refer to an object of type `C`.

Actually, converting the type of an object by performing a cast is not usually a good idea. In our example, you do not need to cast an `Employee` object to a `Manager` object for most purposes. The `getSalary` method will work correctly on both objects of both classes. The dynamic binding that makes polymorphism work locates the correct method automatically.

The only reason to make the cast is to use a method that is unique to managers, such as `setBonus`. If for some reason you find yourself wanting to call `setBonus` on `Employee` objects, ask yourself whether this is an indication of a design flaw in the superclass. It may make sense to redesign the superclass and add a `setBonus` method. Remember, it takes only one uncaught `ClassCastException` to terminate your program. In general, it is best to minimize the use of casts and the `instanceof` operator.



C++ NOTE: Java uses the cast syntax from the “bad old days” of C, but it works like the safe `dynamic_cast` operation of C++. For example,

```
Manager boss = (Manager) staff[1]; // Java
```

is the same as

```
Manager* boss = dynamic_cast<Manager*>(staff[1]); // C++
```

with one important difference. If the cast fails, it does not yield a null object but throws an exception. In this sense, it is like a C++ cast of *references*. This is a pain in the neck. In C++, you can take care of the type test and type conversion in one operation.

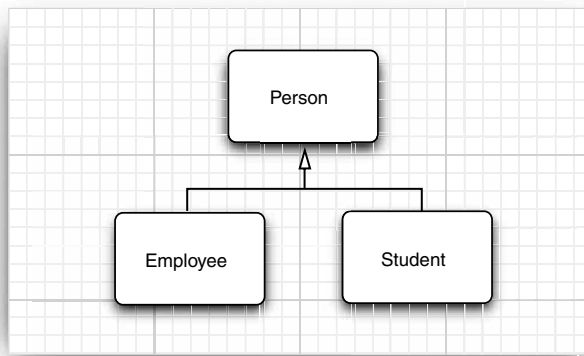
```
Manager* boss = dynamic_cast<Manager*>(staff[1]); // C++
if (boss != NULL) . . .
```

In Java, you use a combination of the `instanceof` operator and a cast.

```
if (staff[1] instanceof Manager)
{
    Manager boss = (Manager) staff[1];
    . . .
}
```

### Abstract Classes

As you move up the inheritance hierarchy, classes become more general and probably more abstract. At some point, the ancestor class becomes *so* general that you think of it more as a basis for other classes than as a class with specific instances you want to use. Consider, for example, an extension of our `Employee` class hierarchy. An employee is a person, and so is a student. Let us extend our class hierarchy to include classes `Person` and `Student`. Figure 5-2 shows the inheritance relationships between these classes.



**Figure 5-2 Inheritance diagram for Person and its subclasses**

Why bother with so high a level of abstraction? There are some attributes that make sense for every person, such as the name. Both students and employees have names,

and introducing a common superclass lets us factor out the `getName` method to a higher level in the inheritance hierarchy.

Now let's add another method, `getDescription`, whose purpose is to return a brief description of the person, such as

```
    an employee with a salary of $50,000.00
    a student majoring in computer science
```

It is easy to implement this method for the `Employee` and `Student` classes. But what information can you provide in the `Person` class? The `Person` class knows nothing about the person except the name. Of course, you could implement `Person.getDescription()` to return an empty string. But there is a better way. If you use the `abstract` keyword, you do not need to implement the method at all.

```
    public abstract String getDescription();
    // no implementation required
```

For added clarity, a class with one or more abstract methods must itself be declared `abstract`.

```
abstract class Person
{
    . . .
    public abstract String getDescription();
}
```

In addition to abstract methods, abstract classes can have fields and concrete methods. For example, the `Person` class stores the name of the person and has a concrete method that returns it.

```
abstract class Person
{
    public Person(String n)
    {
        name = n;
    }

    public abstract String getDescription();

    public String getName()
    {
        return name;
    }

    private String name;
}
```



**TIP:** Some programmers don't realize that abstract classes can have concrete methods. You should always move common fields and methods (whether abstract or not) to the superclass (whether abstract or not).

Abstract methods act as placeholders for methods that are implemented in the subclasses. When you extend an abstract class, you have two choices. You can leave some or all of the abstract methods undefined. Then you must tag the subclass as `abstract` as well. Or you can define all methods. Then the subclass is no longer `abstract`.

For example, we will define a `Student` class that extends the abstract `Person` class and implements the `getDescription` method. Because none of the methods of the `Student` class are abstract, it does not need to be declared as an abstract class.

A class can even be declared as `abstract` even though it has no abstract methods.

Abstract classes cannot be instantiated. That is, if a class is declared as `abstract`, no objects of that class can be created. For example, the expression

```
new Person("Vince Vu")
```

is an error. However, you can create objects of concrete subclasses.

Note that you can still create *object variables* of an abstract class, but such a variable must refer to an object of a nonabstract subclass. For example:

```
Person p = new Student("Vince Vu", "Economics");
```

Here `p` is a variable of the abstract type `Person` that refers to an instance of the nonabstract subclass `Student`.



**C++ NOTE:** In C++, an abstract method is called a *pure virtual function* and is tagged with a trailing `= 0`, such as in

```
class Person // C++
{
public:
    virtual string getDescription() = 0;
    . . .
};
```

A C++ class is abstract if it has at least one pure virtual function. In C++, there is no special keyword to denote abstract classes.

---

Let us define a concrete subclass `Student` that extends the abstract `Person` class:

```
class Student extends Person
{
public Student(String n, String m)
{
    super(n);
    major = m;
}

public String getDescription()
{
    return "a student majoring in " + major;
}

private String major;
}
```

The `Student` class defines the `getDescription` method. Therefore, all methods in the `Student` class are concrete, and the class is no longer an abstract class.

The program shown in Listing 5–2 defines the abstract superclass `Person` and two concrete subclasses, `Employee` and `Student`. We fill an array of `Person` references with employee and student objects:

```
Person[] people = new Person[2];
people[0] = new Employee(. . .);
people[1] = new Student(. . .);
```

We then print the names and descriptions of these objects:

```
for (Person p : people)
    System.out.println(p.getName() + ", " + p.getDescription());
```

Some people are baffled by the call

```
p.getDescription()
```

Isn't this call an undefined method? Keep in mind that the variable `p` never refers to a `Person` object because it is impossible to construct an object of the abstract `Person` class. The variable `p` always refers to an object of a concrete subclass such as `Employee` or `Student`. For these objects, the `getDescription` method is defined.

Could you have omitted the abstract method altogether from the `Person` superclass and simply defined the `getDescription` methods in the `Employee` and `Student` subclasses? If you did that, then you wouldn't have been able to invoke the `getDescription` method on the variable `p`. The compiler ensures that you invoke only methods that are declared in the class.

Abstract methods are an important concept in the Java programming language. You will encounter them most commonly inside *interfaces*. For more information about interfaces, turn to Chapter 6.

#### Listing 5–2 PersonTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates abstract classes.
5.  * @version 1.01 2004-02-21
6.  * @author Cay Horstmann
7.  */
8. public class PersonTest
9. {
10.     public static void main(String[] args)
11.     {
12.         Person[] people = new Person[2];
13.
14.         // fill the people array with Student and Employee objects
15.         people[0] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
16.         people[1] = new Student("Maria Morris", "computer science");
17.
18.         // print out names and descriptions of all Person objects
19.         for (Person p : people)
20.             System.out.println(p.getName() + ", " + p.getDescription());
21.     }
```

**Listing 5-2** PersonTest.java (continued)

```
22. }
23.
24. abstract class Person
25. {
26.     public Person(String n)
27.     {
28.         name = n;
29.     }
30.
31.     public abstract String getDescription();
32.
33.     public String getName()
34.     {
35.         return name;
36.     }
37.
38.     private String name;
39. }
40.
41. class Employee extends Person
42. {
43.     public Employee(String n, double s, int year, int month, int day)
44.     {
45.         super(n);
46.         salary = s;
47.         GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
48.         hireDay = calendar.getTime();
49.     }
50.
51.     public double getSalary()
52.     {
53.         return salary;
54.     }
55.
56.     public Date getHireDay()
57.     {
58.         return hireDay;
59.     }
60.
61.     public String getDescription()
62.     {
63.         return String.format("an employee with a salary of $%.2F", salary);
64.     }
65.
66.     public void raiseSalary(double byPercent)
67.     {
68.         double raise = salary * byPercent / 100;
69.         salary += raise;
70.     }
```

**Listing 5-2** PersonTest.java (continued)

```
71.
72.     private double salary;
73.     private Date hireDay;
74. }
75.
76. class Student extends Person
77. {
78.     /**
79.      * @param n the student's name
80.      * @param m the student's major
81.      */
82.     public Student(String n, String m)
83.     {
84.         // pass n to superclass constructor
85.         super(n);
86.         major = m;
87.     }
88.
89.     public String getDescription()
90.     {
91.         return "a student majoring in " + major;
92.     }
93.
94.     private String major;
95. }
```

### ***Protected Access***

As you know, fields in a class are best tagged as `private`, and methods are usually tagged as `public`. Any features declared `private` won't be visible to other classes. As we said at the beginning of this chapter, this is also true for subclasses: a subclass cannot access the private fields of its superclass.

There are times, however, when you want to restrict a method to subclasses only or, less commonly, to allow subclass methods to access a superclass field. In that case, you declare a class feature as `protected`. For example, if the superclass `Employee` declares the `hireDay` field as `protected` instead of `private`, then the `Manager` methods can access it directly.

However, the `Manager` class methods can peek inside the `hireDay` field of `Manager` objects only, not of other `Employee` objects. This restriction is made so that you can't abuse the `protected` mechanism and form subclasses just to gain access to the `protected` fields.

In practice, use `protected` fields with caution. Suppose your class is used by other programmers and you designed it with `protected` fields. Unknown to you, other programmers may inherit classes from your class and then start accessing your `protected` fields. In this case, you can no longer change the implementation of your class without upsetting the other programmers. That is against the spirit of OOP, which encourages data encapsulation.

Protected methods make more sense. A class may declare a method as protected if it is tricky to use. This indicates that the subclasses (which, presumably, know their ancestors well) can be trusted to use the method correctly, but other classes cannot.

A good example of this kind of method is the `clone` method of the `Object` class—see Chapter 6 for more details.



**C++ NOTE:** As it happens, protected features in Java are visible to all subclasses as well as to all other classes in the same package. This is slightly different from the C++ meaning of protected, and it makes the notion of protected in Java even less safe than in C++.

Here is a summary of the four access modifiers in Java that control visibility:

1. Visible to the class only (*private*).
2. Visible to the world (*public*).
3. Visible to the package and all subclasses (*protected*).
4. Visible to the package—the (unfortunate) default. No modifiers are needed.

### Object: The Cosmic Superclass

The `Object` class is the ultimate ancestor—every class in Java extends `Object`. However, you never have to write

```
class Employee extends Object
```

The ultimate superclass `Object` is taken for granted if no superclass is explicitly mentioned. Because *every* class in Java extends `Object`, it is important to be familiar with the services provided by the `Object` class. We go over the basic ones in this chapter and refer you to later chapters or to the on-line documentation for what is not covered here. (Several methods of `Object` come up only when dealing with threads—see Volume II for more on threads.)

You can use a variable of type `Object` to refer to objects of any type:

```
Object obj = new Employee("Harry Hacker", 35000);
```

Of course, a variable of type `Object` is only useful as a generic holder for arbitrary values. To do anything specific with the value, you need to have some knowledge about the original type and then apply a cast:

```
Employee e = (Employee) obj;
```

In Java, only the *primitive types* (numbers, characters, and boolean values) are not objects.

All array types, no matter whether they are arrays of objects or arrays of primitive types, are class types that extend the `Object` class.

```
Employee[] staff = new Employee[10];
obj = staff; // OK
obj = new int[10]; // OK
```



**C++ NOTE:** In C++, there is no cosmic root class. However, every pointer can be converted to a `void*` pointer.



### The equals Method

The `equals` method in the `Object` class tests whether one object is considered equal to another. The `equals` method, as implemented in the `Object` class, determines whether two object references are identical. This is a pretty reasonable default—if two objects are identical, they should certainly be equal. For quite a few classes, nothing else is required. For example, it makes little sense to compare two `PrintStream` objects for equality. However, you will often want to implement state-based equality testing, in which two objects are considered equal when they have the same state.

For example, let us consider two employees equal if they have the same name, salary, and hire date. (In an actual employee database, it would be more sensible to compare IDs instead. We use this example to demonstrate the mechanics of implementing the `equals` method.)

```
class Employee
{
    . . .
    public boolean equals(Object otherObject)
    {
        // a quick test to see if the objects are identical
        if (this == otherObject) return true;

        // must return false if the explicit parameter is null
        if (otherObject == null) return false;

        // if the classes don't match, they can't be equal
        if (getClass() != otherObject.getClass())
            return false;

        // now we know otherObject is a non-null Employee
        Employee other = (Employee) otherObject;

        // test whether the fields have identical values
        return name.equals(other.name)
            && salary == other.salary
            && hireDay.equals(other.hireDay);
    }
}
```

The `getClass` method returns the class of an object—we discuss this method in detail later in this chapter. In our test, two objects can only be equal when they belong to the same class.

When you define the `equals` method for a subclass, first call `equals` on the superclass. If that test doesn't pass, then the objects can't be equal. If the superclass fields are equal, then you are ready to compare the instance fields of the subclass.

```
class Manager extends Employee
{
    . . .
    public boolean equals(Object otherObject)
    {
        if (!super.equals(otherObject)) return false;
    }
}
```

```

        // super.equals checked that this and otherObject belong to the same class
        Manager other = (Manager) otherObject;
        return bonus == other.bonus;
    }
}

```

### **Equality Testing and Inheritance**

How should the `equals` method behave if the implicit and explicit parameters don't belong to the same class? This has been an area of some controversy. In the preceding example, the `equals` method returns false if the classes don't match exactly. But many programmers use an `instanceof` test instead:

```
if (!(otherObject instanceof Employee)) return false;
```

This leaves open the possibility that `otherObject` can belong to a subclass. However, this approach can get you into trouble. Here is why. The Java Language Specification requires that the `equals` method has the following properties:

1. It is *reflexive*: For any non-null reference `x`, `x.equals(x)` should return true.
2. It is *symmetric*: For any references `x` and `y`, `x.equals(y)` should return true if and only if `y.equals(x)` returns true.
3. It is *transitive*: For any references `x`, `y`, and `z`, if `x.equals(y)` returns true and `y.equals(z)` returns true, then `x.equals(z)` should return true.
4. It is *consistent*: If the objects to which `x` and `y` refer haven't changed, then repeated calls to `x.equals(y)` return the same value.
5. For any non-null reference `x`, `x.equals(null)` should return false.

These rules are certainly reasonable. You wouldn't want a library implementor to ponder whether to call `x.equals(y)` or `y.equals(x)` when locating an element in a data structure.

However, the symmetry rule has subtle consequences when the parameters belong to different classes. Consider a call

```
e.equals(m)
```

where `e` is an `Employee` object and `m` is a `Manager` object, both of which happen to have the same name, salary, and hire date. If `Employee.equals` uses an `instanceof` test, the call returns true. But that means that the reverse call

```
m.equals(e)
```

also needs to return true—the symmetry rule does not allow it to return false or to throw an exception.

That leaves the `Manager` class in a bind. Its `equals` method must be willing to compare itself to any `Employee`, without taking manager-specific information into account! All of a sudden, the `instanceof` test looks less attractive!

Some authors have gone on record that the `getClass` test is wrong because it violates the substitution principle. A commonly cited example is the `equals` method in the `AbstractSet` class that tests whether two sets have the same elements. The `AbstractSet` class has two concrete subclasses, `TreeSet` and `HashSet`, that use different algorithms for locating set elements. You really want to be able to compare any two sets, no matter how they are implemented.

However, the set example is rather specialized. It would make sense to declare `AbstractSet.equals` as `final`, because nobody should redefine the semantics of set equality. (The method is not actually `final`. This allows a subclass to implement a more efficient algorithm for the equality test.)

The way we see it, there are two distinct scenarios:

- If subclasses can have their own notion of equality, then the symmetry requirement forces you to use the `getClass` test.
- If the notion of equality is fixed in the superclass, then you can use the `instanceof` test and allow objects of different subclasses to be equal to another.

In the example of the employees and managers, we consider two objects to be equal when they have matching fields. If we have two `Manager` objects with the same name, salary, and hire date, but with different bonuses, we want them to be different. Therefore, we used the `getClass` test.

But suppose we used an employee ID for equality testing. This notion of equality makes sense for all subclasses. Then we could use the `instanceof` test, and we should declare `Employee.equals` as `final`.



NOTE: The standard Java library contains over 150 implementations of `equals` methods, with a mishmash of using `instanceof`, calling `getClass`, catching a `ClassCastException`, or doing nothing at all.

Here is a recipe for writing the perfect `equals` method:

1. Name the explicit parameter `otherObject`—later, you need to cast it to another variable that you should call `other`.
2. Test whether this happens to be identical to `otherObject`:
 

```
if (this == otherObject) return true;
```

This statement is just an optimization. In practice, this is a common case. It is much cheaper to check for identity than to compare the fields.
3. Test whether `otherObject` is `null` and return `false` if it is. This test is required.
 

```
if (otherObject == null) return false;
```
4. Compare the classes of `this` and `otherObject`. If the semantics of `equals` can change in subclasses, use the `getClass` test:
 

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

If the same semantics holds for *all* subclasses, you can use an `instanceof` test:
 

```
if (!(otherObject instanceof ClassName)) return false;
```
5. Cast `otherObject` to a variable of your class type:
 

```
ClassName other = (ClassName) otherObject
```
6. Now compare the fields, as required by your notion of equality. Use `==` for primitive type fields, `equals` for object fields. Return `true` if all fields match, `false` otherwise.
 

```
return field1 == other.field1
    && field2.equals(other.field2)
    && . . . ;
```

If you redefine `equals` in a subclass, include a call to `super.equals(other)`.



TIP: If you have fields of array type, you can use the static `Arrays.equals` method to check that corresponding array elements are equal.



CAUTION: Here is a common mistake when implementing the `equals` method. Can you spot the problem?

```
public class Employee
{
    public boolean equals(Employee other)
    {
        return name.equals(other.name)
            && salary == other.salary
            && hireDay.equals(other.hireDay);
    }
    ...
}
```

This method declares the explicit parameter type as `Employee`. As a result, it does not override the `equals` method of the `Object` class but defines a completely unrelated method.

Starting with Java SE 5.0, you can protect yourself against this type of error by tagging methods that are intended to override superclass methods with `@Override`:

```
@Override public boolean equals(Object other)
```

If you made a mistake and you are defining a new method, the compiler reports an error. For example, suppose you add the following declaration to the `Employee` class:

```
@Override public boolean equals(Employee other)
```

An error is reported because this method doesn't override any method from the `Object` superclass.

#### API `java.util.Arrays` 1.2

- `static boolean equals(type[] a, type[] b)` 5.0  
returns true if the arrays have equal lengths and equal elements in corresponding positions. The arrays can have component types `Object`, `int`, `long`, `short`, `char`, `byte`, `boolean`, `float`, or `double`.

#### The hashCode Method

A hash code is an integer that is derived from an object. Hash codes should be scrambled—if `x` and `y` are two distinct objects, there should be a high probability that `x.hashCode()` and `y.hashCode()` are different. Table 5-1 lists a few examples of hash codes that result from the `hashCode` method of the `String` class.

The `String` class uses the following algorithm to compute the hash code:

```
int hash = 0;
for (int i = 0; i < length(); i++)
    hash = 31 * hash + charAt(i);
```

**Table 5-1 Hash Codes Resulting from the hashCode Function**

String	Hash Code
Hello	69609650
Harry	69496448
Hacker	-2141031506

The `hashCode` method is defined in the `Object` class. Therefore, every object has a default hash code. That hash code is derived from the object's memory address. Consider this example:

```
String s = "Ok";
StringBuilder sb = new StringBuilder(s);
System.out.println(s.hashCode() + " " + sb.hashCode());
String t = new String("Ok");
StringBuilder tb = new StringBuilder(t);
System.out.println(t.hashCode() + " " + tb.hashCode());
```

Table 5-2 shows the result.

**Table 5-2 Hash Codes of Strings and String Builders**

Object	Hash Code
s	2556
sb	20526976
t	2556
tb	20527144

Note that the strings `s` and `t` have the same hash code because, for strings, the hash codes are derived from their *contents*. The string builders `sb` and `tb` have different hash codes because no `hashCode` method has been defined for the `StringBuilder` class, and the default `hashCode` method in the `Object` class derives the hash code from the object's memory address.

If you redefine the `equals` method, you will also need to redefine the `hashCode` method for objects that users might insert into a hash table. (We discuss hash tables in Chapter 2 of Volume II.)

The `hashCode` method should return an integer (which can be negative). Just combine the hash codes of the instance fields so that the hash codes for different objects are likely to be widely scattered.

For example, here is a `hashCode` method for the `Employee` class:

```
class Employee
{
    public int hashCode()
    {
```

```

        return 7 * name.hashCode()
            + 11 * new Double(salary).hashCode()
            + 13 * hireDay.hashCode();
    }
    . . .
}

```

Your definitions of `equals` and `hashCode` must be compatible: if `x.equals(y)` is true, then `x.hashCode()` must be the same value as `y.hashCode()`. For example, if you define `Employee.equals` to compare employee IDs, then the `hashCode` method needs to hash the IDs, not employee names or memory addresses.



TIP: If you have fields of array type, you can use the static `Arrays.hashCode` method to compute a hash code that is composed of the hash codes of the array elements.



#### `java.lang.Object` 1.0

- `int hashCode()`  
returns a hash code for this object. A hash code can be any integer, positive or negative. Equal objects need to return identical hash codes.



#### `java.util.Arrays` 1.2

- static `int hashCode(type[] a)` 5.0  
computes the hash code of the array `a`, which can have component type `Object`, `int`, `long`, `short`, `char`, `byte`, `boolean`, `float`, or `double`.

### The `toString` Method

Another important method in `Object` is the `toString` method that returns a string representing the value of this object. Here is a typical example. The `toString` method of the `Point` class returns a string like this:

```
java.awt.Point[x=10,y=20]
```

Most (but not all) `toString` methods follow this format: the name of the class, followed by the field values enclosed in square brackets. Here is an implementation of the `toString` method for the `Employee` class:

```

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

```

Actually, you can do a little better. Rather than hardwiring the class name into the `toString` method, call `getClass().getName()` to obtain a string with the class name.

```

public String toString()
{
    return getClass().getName()

```

```

    + "[name=" + name
    + ",salary=" + salary
    + ",hireDay=" + hireDay
    + "];"
}

```

The `toString` method then also works for subclasses.

Of course, the subclass programmer should define its own `toString` method and add the subclass fields. If the superclass uses `getClass().getName()`, then the subclass can simply call `super.toString()`. For example, here is a `toString` method for the `Manager` class:

```

class Manager extends Employee
{
    . . .
    public String toString()
    {
        return super.toString()
            + "[bonus=" + bonus
            + "];"
    }
}

```

Now a `Manager` object is printed as

```
Manager[name=...,salary=...,hireDay=...][bonus=...]
```

The `toString` method is ubiquitous for an important reason: whenever an object is concatenated with a string by the “+” operator, the Java compiler automatically invokes the `toString` method to obtain a string representation of the object. For example:

```

Point p = new Point(10, 20);
String message = "The current position is " + p;
// automatically invokes p.toString()

```



**TIP:** Instead of writing `x.toString()`, you can write `"" + x`. This statement concatenates the empty string with the string representation of `x` that is exactly `x.toString()`. Unlike `toString`, this statement even works if `x` is of primitive type.

If `x` is any object and you call

```
System.out.println(x);
```

then the `println` method simply calls `x.toString()` and prints the resulting string.

The `Object` class defines the `toString` method to print the class name and the hash code of the object. For example, the call

```
System.out.println(System.out)
```

produces an output that looks like this:

```
java.io.PrintStream@2f6684
```

The reason is that the implementor of the `PrintStream` class didn't bother to override the `toString` method.



**CAUTION:** Annoyingly, arrays inherit the `toString` method from `Object`, with the added twist that the array type is printed in an archaic format. For example,

```
int[] luckyNumbers = { 2, 3, 5, 7, 11, 13 };
String s = "" + luckyNumbers;
```

yields the string "[I@1a46e30". (The prefix `[I` denotes an array of integers.) The remedy is to call the static `Arrays.toString` method instead. The code

```
String s = Arrays.toString(luckyNumbers);
```

yields the string "[2, 3, 5, 7, 11, 13]".

To correctly print multidimensional arrays (that is, arrays of arrays), use `Arrays.deepToString`.

The `toString` method is a great tool for logging. Many classes in the standard class library define the `toString` method so that you can get useful information about the state of an object. This is particularly useful in logging messages like this:

```
System.out.println("Current position = " + position);
```

As we explain in Chapter 11, an even better solution is

```
Logger.global.info("Current position = " + position);
```



**TIP:** We strongly recommend that you add a `toString` method to each class that you write. You, as well as other programmers who use your classes, will be grateful for the logging support.

The program in Listing 5–3 implements the `equals`, `hashCode`, and `toString` methods for the `Employee` and `Manager` classes.

#### Listing 5–3 EqualsTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates the equals method.
5.  * @version 1.11 2004-02-21
6.  * @author Cay Horstmann
7.  */
8. public class EqualsTest
9. {
10.     public static void main(String[] args)
11.     {
12.         Employee alice1 = new Employee("Alice Adams", 75000, 1987, 12, 15);
13.         Employee alice2 = alice1;
14.         Employee alice3 = new Employee("Alice Adams", 75000, 1987, 12, 15);
15.         Employee bob = new Employee("Bob Brandson", 50000, 1989, 10, 1);
16.
17.         System.out.println("alice1 == alice2: " + (alice1 == alice2));
18.     }
19. }
```



**Listing 5-3** EqualsTest.java (continued)

```
19.     System.out.println("alice1 == alice3: " + (alice1 == alice3));
20.
21.     System.out.println("alice1.equals(alice3): " + alice1.equals(alice3));
22.
23.     System.out.println("alice1.equals(bob): " + alice1.equals(bob));
24.
25.     System.out.println("bob.toString(): " + bob);
26.
27.     Manager carl = new Manager("Carl Cracker", 80000, 1987, 12, 15);
28.     Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
29.     boss.setBonus(5000);
30.     System.out.println("boss.toString(): " + boss);
31.     System.out.println("carl.equals(boss): " + carl.equals(boss));
32.     System.out.println("alice1.hashCode(): " + alice1.hashCode());
33.     System.out.println("alice3.hashCode(): " + alice3.hashCode());
34.     System.out.println("bob.hashCode(): " + bob.hashCode());
35.     System.out.println("carl.hashCode(): " + carl.hashCode());
36. }
37. }
38.
39. class Employee
40. {
41.     public Employee(String n, double s, int year, int month, int day)
42.     {
43.         name = n;
44.         salary = s;
45.         GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
46.         hireDay = calendar.getTime();
47.     }
48.
49.     public String getName()
50.     {
51.         return name;
52.     }
53.
54.     public double getSalary()
55.     {
56.         return salary;
57.     }
58.
59.     public Date getHireDay()
60.     {
61.         return hireDay;
62.     }
63.
64.     public void raiseSalary(double byPercent)
65.     {
66.         double raise = salary * byPercent / 100;
67.         salary += raise;
68.     }
```

**Listing 5-3** EqualsTest.java (continued)

```
69.
70. public boolean equals(Object otherObject)
71. {
72.     // a quick test to see if the objects are identical
73.     if (this == otherObject) return true;
74.
75.     // must return false if the explicit parameter is null
76.     if (otherObject == null) return false;
77.
78.     // if the classes don't match, they can't be equal
79.     if (getClass() != otherObject.getClass()) return false;
80.
81.     // now we know otherObject is a non-null Employee
82.     Employee other = (Employee) otherObject;
83.
84.     // test whether the fields have identical values
85.     return name.equals(other.name) && salary == other.salary && hireDay.equals(other.hireDay);
86. }
87.
88. public int hashCode()
89. {
90.     return 7 * name.hashCode() + 11 * new Double(salary).hashCode() + 13 * hireDay.hashCode();
91. }
92.
93. public String toString()
94. {
95.     return getClass().getName() + "[name=" + name + ",salary=" + salary + ",hireDay=" + hireDay
96.         + " ]";
97. }
98.
99. private String name;
100. private double salary;
101. private Date hireDay;
102. }
103.
104. class Manager extends Employee
105. {
106.     public Manager(String n, double s, int year, int month, int day)
107.     {
108.         super(n, s, year, month, day);
109.         bonus = 0;
110.     }
111.
112.     public double getSalary()
113.     {
114.         double baseSalary = super.getSalary();
115.         return baseSalary + bonus;
116.     }
117.
```

**Listing 5-3** EqualsTest.java (continued)

```
118.
119. public void setBonus(double b)
120. {
121.     bonus = b;
122. }
123.
124. public boolean equals(Object otherObject)
125. {
126.     if (!super.equals(otherObject)) return false;
127.     Manager other = (Manager) otherObject;
128.     // super.equals checked that this and other belong to the same class
129.     return bonus == other.bonus;
130. }
131.
132. public int hashCode()
133. {
134.     return super.hashCode() + 17 * new Double(bonus).hashCode();
135. }
136.
137. public String toString()
138. {
139.     return super.toString() + "[bonus=" + bonus + "]";
140. }
141.
142. private double bonus;
143. }
```

**API** java.lang.Object 1.0

- `Class getClass()`  
returns a class object that contains information about the object. As you see later in this chapter, Java has a runtime representation for classes that is encapsulated in the `Class` class.
- `boolean equals(Object otherObject)`  
compares two objects for equality; returns `true` if the objects point to the same area of memory, and `false` otherwise. You should override this method in your own classes.
- `String toString()`  
returns a string that represents the value of this object. You should override this method in your own classes.
- `Object clone()`  
creates a clone of the object. The Java runtime system allocates memory for the new instance and copies the memory allocated for the current object.



NOTE: Cloning an object is important, but it also turns out to be a fairly subtle process filled with potential pitfalls for the unwary. We will have a lot more to say about the `clone` method in Chapter 6.



`java.lang.Class` 1.0

- `String getName()`  
returns the name of this class.
- `Class getSuperclass()`  
returns the superclass of this class as a `Class` object.

### Generic Array Lists

In many programming languages—in particular, in C—you have to fix the sizes of all arrays at compile time. Programmers hate this because it forces them into uncomfortable trade-offs. How many employees will be in a department? Surely no more than 100. What if there is a humongous department with 150 employees? Do we want to waste 90 entries for every department with just 10 employees?

In Java, the situation is much better. You can set the size of an array at runtime.

```
int actualSize = . . . ;
Employee[] staff = new Employee[actualSize];
```

Of course, this code does not completely solve the problem of dynamically modifying arrays at runtime. Once you set the array size, you cannot change it easily. Instead, the easiest way in Java to deal with this common situation is to use another Java class, called `ArrayList`. The `ArrayList` class is similar to an array, but it automatically adjusts its capacity as you add and remove elements, without your needing to write any code.

As of Java SE 5.0, `ArrayList` is a *generic class* with a *type parameter*. To specify the type of the element objects that the array list holds, you append a class name enclosed in angle brackets, such as `ArrayList<Employee>`. You will see in Chapter 13 how to define your own generic class, but you don't need to know any of those technicalities to use the `ArrayList` type.

Here we declare and construct an array list that holds `Employee` objects:

```
ArrayList<Employee> staff = new ArrayList<Employee>();
```



NOTE: Before Java SE 5.0, there were no generic classes. Instead, there was a single `ArrayList` class, a “one size fits all” collection that holds elements of type `Object`. If you must use an older version of Java, simply drop all `<...>` suffixes. You can still use `ArrayList` without a `<...>` suffix in Java SE 5.0 and beyond. It is considered a “raw” type, with the type parameter erased.



NOTE: In even older versions of the Java programming language, programmers used the `Vector` class for dynamic arrays. However, the `ArrayList` class is more efficient, and there is no longer any good reason to use the `Vector` class.

You use the `add` method to add new elements to an array list. For example, here is how you populate an array list with employee objects:

```
staff.add(new Employee("Harry Hacker", . . .));  
staff.add(new Employee("Tony Tester", . . .));
```

The array list manages an internal array of object references. Eventually, that array will run out of space. This is where array lists work their magic: If you call `add` and the internal array is full, the array list automatically creates a bigger array and copies all the objects from the smaller to the bigger array.

If you already know, or have a good guess, how many elements you want to store, then call the `ensureCapacity` method before filling the array list:

```
staff.ensureCapacity(100);
```

That call allocates an internal array of 100 objects. Then, the first 100 calls to `add` do not involve any costly reallocation.

You can also pass an initial capacity to the `ArrayList` constructor:

```
ArrayList<Employee> staff = new ArrayList<Employee>(100);
```



CAUTION: Allocating an array list as

```
new ArrayList<Employee>(100) // capacity is 100
```

is *not* the same as allocating a new array as

```
new Employee[100] // size is 100
```

There is an important distinction between the capacity of an array list and the size of an array. If you allocate an array with 100 entries, then the array has 100 slots, ready for use. An array list with a capacity of 100 elements has the *potential* of holding 100 elements (and, in fact, more than 100, at the cost of additional reallocations); but at the beginning, even after its initial construction, an array list holds no elements at all.

The `size` method returns the actual number of elements in the array list. For example,

```
staff.size()
```

returns the current number of elements in the `staff` array list. This is the equivalent of

```
a.length
```

for an array `a`.

Once you are reasonably sure that the array list is at its permanent size, you can call the `trimToSize` method. This method adjusts the size of the memory block to use exactly as much storage space as is required to hold the current number of elements. The garbage collector will reclaim any excess memory.

Once you trim the size of an array list, adding new elements will move the block again, which takes time. You should only use `trimToSize` when you are sure you won't add any more elements to the array list.

**C++** NOTE: The `ArrayList` class is similar to the C++ vector template. Both `ArrayList` and vector are generic types. But the C++ vector template overloads the `[]` operator for convenient element access. Because Java does not have operator overloading, it must use explicit method calls instead. Moreover, C++ vectors are copied by value. If `a` and `b` are two vectors, then the assignment `a = b` makes `a` into a new vector with the same length as `b`, and all elements are copied from `b` to `a`. The same assignment in Java makes both `a` and `b` refer to the same array list.

**API** `java.util.ArrayList<T>` 1.2

- `ArrayList<T>()`  
constructs an empty array list.
- `ArrayList<T>(int initialCapacity)`  
constructs an empty array list with the specified capacity.  
*Parameters:*    `initialCapacity`            the initial storage capacity of the array list
- `boolean add(T obj)`  
appends an element at the end of the array list. Always returns `true`.  
*Parameters:*    `obj`                                    the element to be added
- `int size()`  
returns the number of elements currently stored in the array list. (Of course, this is never larger than the array list's capacity.)
- `void ensureCapacity(int capacity)`  
ensures that the array list has the capacity to store the given number of elements without reallocating its internal storage array.  
*Parameters:*    `capacity`                            the desired storage capacity
- `void trimToSize()`  
reduces the storage capacity of the array list to its current size.

### Accessing Array List Elements

Unfortunately, nothing comes for free. The automatic growth convenience that array lists give requires a more complicated syntax for accessing the elements. The reason is that the `ArrayList` class is not a part of the Java programming language; it is just a utility class programmed by someone and supplied in the standard library.

Instead of using the pleasant `[]` syntax to access or change the element of an array, you use the `get` and `set` methods.


For example, to set the `i`th element, you use

```
staff.set(i, harry);
```

This is equivalent to

```
a[i] = harry;
```

for an array `a`. (As with arrays, the index values are zero-based.)

 CAUTION: Do not call `list.set(i, x)` until the *size* of the array list is larger than `i`. For example, the following code is wrong:

```
ArrayList<Employee> list = new ArrayList<Employee>(100); // capacity 100, size 0
list.set(0, x); // no element 0 yet
```


Use the `add` method instead of `set` to fill up an array, and use `set` only to replace a previously added element.

To get an array list element, use

```
Employee e = staff.get(i);
```

This is equivalent to

```
Employee e = a[i];
```

 NOTE: Before Java SE 5.0, there were no generic classes, and the `get` method of the raw `ArrayList` class had no choice but to return an `Object`. Consequently, callers of `get` had to cast the returned value to the desired type:

```
Employee e = (Employee) staff.get(i);
```

The raw `ArrayList` is also a bit dangerous. Its `add` and `set` methods accept objects of any type. A call

```
staff.set(i, new Date());
```

compiles without so much as a warning, and you run into grief only when you retrieve the object and try to cast it. If you use an `ArrayList<Employee>` instead, the compiler will detect this error.

You can sometimes get the best of both worlds—flexible growth and convenient element access—with the following trick. First, make an array list and add all the elements:

```
ArrayList<X> list = new ArrayList<X>();
while (. . .)
{
    x = . . . ;
    list.add(x);
}
```

When you are done, use the `toArray` method to copy the elements into an array:

```
X[] a = new X[list.size()];
list.toArray(a);
```

Sometimes, you need to add elements in the middle of an array list. Use the `add` method with an index parameter:

```
int n = staff.size() / 2;
staff.add(n, e);
```

The elements at locations `n` and above are shifted up to make room for the new entry. If the new size of the array list after the insertion exceeds the capacity, then the array list reallocates its storage array.

Similarly, you can remove an element from the middle of an array list:

```
Employee e = staff.remove(n);
```

The elements located above it are copied down, and the size of the array is reduced by one.

Inserting and removing elements is not terribly efficient. It is probably not worth worrying about for small array lists. But if you store many elements and frequently insert and remove in the middle of a collection, consider using a linked list instead. We explain how to program with linked lists in Chapter 13.

As of Java SE 5.0, you can use the “for each” loop to traverse the contents of an array list:

```
for (Employee e : staff)
    do something with e
```

This loop has the same effect as

```
for (int i = 0; i < staff.size(); i++)
{
    Employee e = staff.get(i);
    do something with e
}
```

Listing 5–4 is a modification of the `EmployeeTest` program of Chapter 4. The `Employee[]` array is replaced by an `ArrayList<Employee>`. Note the following changes:

- You don’t have to specify the array size.
- You use `add` to add as many elements as you like.
- You use `size()` instead of `length` to count the number of elements.
- You use `a.get(i)` instead of `a[i]` to access an element.

#### Listing 5–4 ArrayListTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates the ArrayList class.
5.  * @version 1.1 2004-02-21
6.  * @author Cay Horstmann
7.  */
8. public class ArrayListTest
9. {
10.     public static void main(String[] args)
11.     {
12.         // fill the staff array list with three Employee objects
13.         ArrayList<Employee> staff = new ArrayList<Employee>();
14.
15.         staff.add(new Employee("Carl Cracker", 75000, 1987, 12, 15));
16.         staff.add(new Employee("Harry Hacker", 50000, 1989, 10, 1));
17.         staff.add(new Employee("Tony Tester", 40000, 1990, 3, 15));
18.
```



**Listing 5-4** ArrayListTest.java (continued)

```
19.    // raise everyone's salary by 5%
20.    for (Employee e : staff)
21.        e.raiseSalary(5);
22.
23.    // print out information about all Employee objects
24.    for (Employee e : staff)
25.        System.out.println("name=" + e.getName() + ",salary=" + e.getSalary() + ",hireDay="
26.            + e.getHireDay());
27.    }
28. }
29.
30. class Employee
31. {
32.     public Employee(String n, double s, int year, int month, int day)
33.     {
34.         name = n;
35.         salary = s;
36.         GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
37.         hireDay = calendar.getTime();
38.     }
39.
40.     public String getName()
41.     {
42.         return name;
43.     }
44.
45.     public double getSalary()
46.     {
47.         return salary;
48.     }
49.
50.     public Date getHireDay()
51.     {
52.         return hireDay;
53.     }
54.
55.     public void raiseSalary(double byPercent)
56.     {
57.         double raise = salary * byPercent / 100;
58.         salary += raise;
59.     }
60.
61.     private String name;
62.     private double salary;
63.     private Date hireDay;
64. }
```

**API** `java.util.ArrayList<T>` 1.2

- `void set(int index, T obj)`  
puts a value in the array list at the specified index, overwriting the previous contents.  
*Parameters:*    `index`        the position (must be between 0 and `size() - 1`)  
   `obj`                the new value
- `T get(int index)`  
gets the value stored at a specified index.  
*Parameters:*    `index`        the index of the element to get (must be between 0 and `size() - 1`)
- `void add(int index, T obj)`  
shifts up elements to insert an element.  
*Parameters:*    `index`        the insertion position (must be between 0 and `size()`)  
   `obj`                the new element
- `T remove(int index)`  
removes an element and shifts down all elements above it. The removed element is returned.  
*Parameters:*    `index`        the position of the element to be removed (must be between 0 and `size() - 1`)

**Compatibility between Typed and Raw Array Lists**

When you write new code with Java SE 5.0 and beyond, you should use type parameters, such as `ArrayList<Employee>`, for array lists. However, you may need to interoperate with existing code that uses the raw `ArrayList` type.

Suppose that you have the following legacy class:

```
public class EmployeeDB
{
    public void update(ArrayList list) { ... }
    public ArrayList find(String query) { ... }
}
```

You can pass a typed array list to the `update` method without any casts.

```
ArrayList<Employee> staff = ...;
employeeDB.update(staff);
```

The `staff` object is simply passed to the `update` method.



**CAUTION:** Even though you get no error or warning from the compiler, this call is not completely safe. The `update` method might add elements into the array list that are not of type `Employee`. When these elements are retrieved, an exception occurs. This sounds scary, but if you think about it, the behavior is simply as it was before Java SE 5.0. The integrity of the virtual machine is never jeopardized. In this situation, you do not lose security, but you also do not benefit from the compile-time checks.

Conversely, when you assign a raw `ArrayList` to a typed one, you get a warning.

```
ArrayList<Employee> result = employeeDB.find(query); // yields warning
```



NOTE: To see the text of the warning, compile with the option `-Xlint:unchecked`.

Using a cast does not make the warning go away.

```
ArrayList<Employee> result = (ArrayList<Employee>)
    employeeDB.find(query); // yields another warning
```

Instead, you get a different warning, telling you that the cast is misleading.

This is the consequence of a somewhat unfortunate limitation of generic types in Java. For compatibility, the compiler translates all typed array lists into raw `ArrayList` objects after checking that the type rules were not violated. In a running program, all array lists are the same—there are no type parameters in the virtual machine. Thus, the casts `(ArrayList)` and `(ArrayList<Employee>)` carry out identical runtime checks.

There isn't much you can do about that situation. When you interact with legacy code, study the compiler warnings and satisfy yourself that the warnings are not serious.

### Object Wrappers and Autoboxing

Occasionally, you need to convert a primitive type like `int` to an object. All primitive types have class counterparts. For example, a class `Integer` corresponds to the primitive type `int`. These kinds of classes are usually called *wrappers*. The wrapper classes have obvious names: `Integer`, `Long`, `Float`, `Double`, `Short`, `Byte`, `Character`, `Void`, and `Boolean`. (The first six inherit from the common superclass `Number`.) The wrapper classes are immutable—you cannot change a wrapped value after the wrapper has been constructed. They are also `final`, so you cannot subclass them.

Suppose we want an array list of integers. Unfortunately, the type parameter inside the angle brackets cannot be a primitive type. It is not possible to form an `ArrayList<int>`.

Here, the `Integer` wrapper class comes in. It is ok to declare an array list of `Integer` objects.

```
ArrayList<Integer> list = new ArrayList<Integer>();
```



CAUTION: An `ArrayList<Integer>` is far less efficient than an `int[]` array because each value is separately wrapped inside an object. You would only want to use this construct for small collections when programmer convenience is more important than efficiency.

Another Java SE 5.0 innovation makes it easy to add and get array elements. The call

```
list.add(3);
```

is automatically translated to

```
list.add(new Integer(3));
```

This conversion is called *autoboxing*.



NOTE: You might think that *autowrapping* would be more consistent, but the “boxing” metaphor was taken from C#.

Conversely, when you assign an `Integer` object to an `int` value, it is automatically unboxed. That is, the compiler translates

```
int n = list.get(i);
```

into

```
int n = list.get(i).intValue();
```

Automatic boxing and unboxing even works with arithmetic expressions. For example, you can apply the increment operator to a wrapper reference:

```
Integer n = 3;  
n++;
```

The compiler automatically inserts instructions to unbox the object, increment the resulting value, and box it back.

In most cases, you get the illusion that the primitive types and their wrappers are one and the same. There is just one point in which they differ considerably: identity. As you know, the `==` operator, applied to wrapper objects, only tests whether the objects have identical memory locations. The following comparison would therefore probably fail:

```
Integer a = 1000;  
Integer b = 1000;  
if (a == b) ...
```

However, a Java implementation *may*, if it chooses, wrap commonly occurring values into identical objects, and thus the comparison might succeed. This ambiguity is not what you want. The remedy is to call the `equals` method when comparing wrapper objects.



**NOTE:** The autoboxing specification requires that `boolean`, `byte`, `char`  $\leq 127$ , and `short` and `int` between  $-128$  and  $127$  are wrapped into fixed objects. For example, if `a` and `b` had been initialized with `100` in the preceding example, then the comparison would have had to succeed.

Finally, let us emphasize that boxing and unboxing is a courtesy of the *compiler*, not the virtual machine. The compiler inserts the necessary calls when it generates the bytecodes of a class. The virtual machine simply executes those bytecodes.

You will often see the number wrappers for another reason. The designers of Java found the wrappers a convenient place to put certain basic methods, like the ones for converting strings of digits to numbers.

To convert a string to an integer, you use the following statement:

```
int x = Integer.parseInt(s);
```

This has nothing to do with `Integer` objects—`parseInt` is a static method. But the `Integer` class was a good place to put it.

The API notes show some of the more important methods of the `Integer` class. The other number classes implement corresponding methods.

**X** CAUTION: Some people think that the wrapper classes can be used to implement methods that can modify numeric parameters. However, that is not correct. Recall from Chapter 4 that it is impossible to write a Java method that increments an integer parameter because parameters to Java methods are always passed by value.

```
public static void triple(int x) // won't work
{
    x = 3 * x; // modifies local variable
}
```

Could we overcome this by using an Integer instead of an int?

```
public static void triple(Integer x) // won't work
{
    ...
}
```

The problem is that Integer objects are *immutable*: the information contained inside the wrapper can't change. You cannot use these wrapper classes to create a method that modifies numeric parameters.

If you do want to write a method to change numeric parameters, you can use one of the *holder* types defined in the `org.omg.CORBA` package. There are types `IntHolder`, `BooleanHolder`, and so on. Each holder type has a public (!) field `value` through which you can access the stored value.

```
public static void triple(IntHolder x)
{
    x.value = 3 * x.value;
}
```

#### API `java.lang.Integer` 1.0

- `int intValue()`  
returns the value of this `Integer` object as an `int` (overrides the `intValue` method in the `Number` class).
- `static String toString(int i)`  
returns a new `String` object representing the number `i` in base 10.
- `static String toString(int i, int radix)`  
lets you return a representation of the number `i` in the base specified by the `radix` parameter.
- `static int parseInt(String s)`
- `static int parseInt(String s, int radix)`  
returns the integer whose digits are contained in the string `s`. The string must represent an integer in base 10 (for the first method) or in the base given by the `radix` parameter (for the second method).
- `static Integer valueOf(String s)`
- `static Integer valueOf(String s, int radix)`  
returns a new `Integer` object initialized to the integer whose digits are contained in the string `s`. The string must represent an integer in base 10 (for the first method) or in the base given by the `radix` parameter (for the second method).

**API** `java.text.NumberFormat 1.1`

- `Number parse(String s)`  
returns the numeric value, assuming the specified `String` represents a number.

**Methods with a Variable Number of Parameters**

Before Java SE 5.0, every Java method had a fixed number of parameters. However, it is now possible to provide methods that can be called with a variable number of parameters. (These are sometimes called “varargs” methods.)

You have already seen such a method: `printf`. For example, the calls

```
System.out.printf("%d", n);
```

and

```
System.out.printf("%d %s", n, "widgets");
```

both call the same method, even though one call has two parameters and the other has three.

The `printf` method is defined like this:

```
public class PrintStream
{
    public PrintStream printf(String fmt, Object... args) { return format(fmt, args); }
}
```

Here, the ellipsis `...` is a part of the Java code. It denotes that the method can receive an arbitrary number of objects (in addition to the `fmt` parameter).

The `printf` method actually receives two parameters, the format string, and an `Object[]` array that holds all other parameters. (If the caller supplies integers or other primitive type values, autoboxing turns them into objects.) It now has the unenviable task of scanning the `fmt` string and matching up the *i*th format specifier with the value `args[i]`.

In other words, for the implementor of `printf`, the `Object...` parameter type is exactly the same as `Object[]`.

The compiler needs to transform each call to `printf`, bundling the parameters into an array and autoboxing as necessary:

```
System.out.printf("%d %s", new Object[] { new Integer(n), "widgets" } );
```

You can define your own methods with variable parameters, and you can specify any type for the parameters, even a primitive type. Here is a simple example: a function that computes the maximum of a variable number of values.

```
public static double max(double... values)
{
    double largest = Double.MIN_VALUE;
    for (double v : values) if (v > largest) largest = v;
    return largest;
}
```

Simply call the function like this:

```
double m = max(3.1, 40.4, -5);
```

The compiler passes a new `double[] { 3.1, 40.4, -5 }` to the `max` function.



NOTE: It is legal to pass an array as the last parameter of a method with variable parameters. For example:

```
System.out.printf("%d %s", new Object[] { new Integer(1), "widgets" } );
```

Therefore, you can redefine an existing function whose last parameter is an array to a method with variable parameters, without breaking any existing code. For example, `MessageFormat.format` was enhanced in this way in Java SE 5.0. If you like, you can even declare the main method as

```
public static void main(String... args)
```

## Enumeration Classes

You saw in Chapter 3 how to define enumerated types in Java SE 5.0 and beyond. Here is a typical example:

```
public enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };
```

The type defined by this declaration is actually a class. The class has exactly four instances—it is not possible to construct new objects.

Therefore, you never need to use `equals` for values of enumerated types. Simply use `==` to compare them.

You can, if you like, add constructors, methods, and fields to an enumerated type. Of course, the constructors are only invoked when the enumerated constants are constructed. Here is an example.

```
enum Size
{
    SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");

    private Size(String abbreviation) { this.abbreviation = abbreviation; }
    public String getAbbreviation() { return abbreviation; }

    private String abbreviation;
}
```

All enumerated types are subclasses of the class `Enum`. They inherit a number of methods from that class. The most useful one is `toString`, which returns the name of the enumerated constant. For example, `Size.SMALL.toString()` returns the string "SMALL".

The converse of `toString` is the static `valueOf` method. For example, the statement

```
Size s = (Size) Enum.valueOf(Size.class, "SMALL");
```

sets `s` to `Size.SMALL`.

Each enumerated type has a static `values` method that returns an array of all values of the enumeration. For example, the call

```
Size[] values = Size.values();
```

returns the array with elements `Size.SMALL`, `Size.MEDIUM`, `Size.LARGE`, and `Size.EXTRA_LARGE`.

The `ordinal` method yields the position of an enumerated constant in the `enum` declaration, counting from zero. For example, `Size.MEDIUM.ordinal()` returns 1.

The short program in Listing 5–5 demonstrates how to work with enumerated types.



NOTE: The Enum class has a type parameter that we have ignored for simplicity. For example, the enumerated type Size actually extends Enum<Size>. The type parameter is used in the compareTo method. (We discuss the compareTo method in Chapter 6 and type parameters in Chapter 12.)

**Listing 5-5** EnumTest.java

```

1. import java.util.*;
2.
3. /**
4.  * This program demonstrates enumerated types.
5.  * @version 1.0 2004-05-24
6.  * @author Cay Horstmann
7.  */
8. public class EnumTest
9. {
10.     public static void main(String[] args)
11.     {
12.         Scanner in = new Scanner(System.in);
13.         System.out.print("Enter a size: (SMALL, MEDIUM, LARGE, EXTRA_LARGE) ");
14.         String input = in.next().toUpperCase();
15.         Size size = Enum.valueOf(Size.class, input);
16.         System.out.println("size=" + size);
17.         System.out.println("abbreviation=" + size.getAbbreviation());
18.         if (size == Size.EXTRA_LARGE)
19.             System.out.println("Good job--you paid attention to the _.");
20.     }
21. }
22.
23. enum Size
24. {
25.     SMALL("S"), MEDIUM("M"), LARGE("L"), EXTRA_LARGE("XL");
26.
27.     private Size(String abbreviation) { this.abbreviation = abbreviation; }
28.     public String getAbbreviation() { return abbreviation; }
29.
30.     private String abbreviation;
31. }

```

**API** java.lang.Enum<E> 5.0

- static Enum valueOf(Class enumClass, String name)  
returns the enumerated constant of the given class with the given name.
- String toString()  
returns the name of this enumerated constant.
- int ordinal()  
returns the zero-based position of this enumerated constant in the enum declaration.



- `int compareTo(E other)`  
returns a negative integer if this enumerated constant comes before `other`, zero if `this == other`, and a positive integer otherwise. The ordering of the constants is given by the `enum` declaration.

## Reflection

The *reflection library* gives you a very rich and elaborate toolset to write programs that manipulate Java code dynamically. This feature is heavily used in *JavaBeans*, the component architecture for Java (see Volume II for more on JavaBeans). Using reflection, Java can support tools like the ones to which users of Visual Basic have grown accustomed. In particular, when new classes are added at design or runtime, rapid application development tools can dynamically inquire about the capabilities of the classes that were added.

A program that can analyze the capabilities of classes is called *reflective*. The reflection mechanism is extremely powerful. As the next sections show, you can use it to

- Analyze the capabilities of classes at runtime;
- Inspect objects at runtime, for example, to write a `toString` method that works for *all* classes;
- Implement generic array manipulation code; and
- Take advantage of `Method` objects that work just like function pointers in languages such as C++.

Reflection is a powerful and complex mechanism; however, it is of interest mainly to tool builders, not application programmers. If you are interested in programming applications rather than tools for other Java programmers, you can safely skip the remainder of this chapter and return to it later.

### The Class Class

While your program is running, the Java runtime system always maintains what is called runtime type identification on all objects. This information keeps track of the class to which each object belongs. Runtime type information is used by the virtual machine to select the correct methods to execute.

However, you can also access this information by working with a special Java class. The class that holds this information is called, somewhat confusingly, `Class`. The `getClass()` method in the `Object` class returns an instance of `Class` type.

```
Employee e;
. . .
Class c1 = e.getClass();
```

Just like an `Employee` object describes the properties of a particular employee, a `Class` object describes the properties of a particular class. Probably the most commonly used method of `Class` is `getName`. This returns the name of the class. For example, the statement

```
System.out.println(e.getClass().getName() + " " + e.getName());
```

prints

```
Employee Harry Hacker
```

if `e` is an employee, or

```
Manager Harry Hacker
```

if `e` is a manager.

If the class is in a package, the package name is part of the class name:

```
Date d = new Date();
Class c1 = d.getClass();
String name = c1.getName(); // name is set to "java.util.Date"
```

You can obtain a `Class` object corresponding to a class name by using the static `forName` method.

```
String className = "java.util.Date";
Class c1 = Class.forName(className);
```

You would use this method if the class name is stored in a string that varies at runtime. This works if `className` is the name of a class or interface. Otherwise, the `forName` method throws a *checked exception*. See the section “A Primer on Catching Exceptions” on page 219 to see how to supply an *exception handler* whenever you use this method.



**TIP:** At startup, the class containing your main method is loaded. It loads all classes that it needs. Each of those loaded classes loads the classes that it needs, and so on. That can take a long time for a big application, frustrating the user. You can give users of your program the illusion of a faster start with the following trick. Make sure that the class containing the main method does not explicitly refer to other classes. First display a splash screen. Then manually force the loading of other classes by calling `Class.forName`.

A third method for obtaining an object of type `Class` is a convenient shorthand. If `T` is any Java type, then `T.class` is the matching class object. For example:

```
Class c1 = Date.class; // if you import java.util.*;
Class c2 = int.class;
Class c3 = Double[].class;
```

Note that a `Class` object really describes a *type*, which may or may not be a class. For example, `int` is not a class, but `int.class` is nevertheless an object of type `Class`.



**NOTE:** As of Java SE 5.0, the `Class` class is parameterized. For example, `Employee.class` is of type `Class<Employee>`. We are not dwelling on this issue because it would further complicate an already abstract concept. For most practical purposes, you can ignore the type parameter and work with the raw `Class` type. See Chapter 13 for more information on this issue.



**CAUTION:** For historical reasons, the `getName` method returns somewhat strange names for array types:

- `Double[].class.getName()` returns "[Ljava.lang.Double;"
- `int[].class.getName()` returns "[I"

The virtual machine manages a unique `Class` object for each type. Therefore, you can use the `==` operator to compare class objects. For example:

```
if (e.getClass() == Employee.class) . . .
```

Another example of a useful method is one that lets you create an instance of a class on the fly. This method is called, naturally enough, `newInstance()`. For example,

```
e.getClass().newInstance();
```

creates a new instance of the same class type as `e`. The `newInstance` method calls the default constructor (the one that takes no parameters) to initialize the newly created object. An exception is thrown if the class has no default constructor.

Using a combination of `forName` and `newInstance` lets you create an object from a class name stored in a string.

```
String s = "java.util.Date";  
Object m = Class.forName(s).newInstance();
```



**NOTE:** If you need to provide parameters for the constructor of a class you want to create by name in this manner, then you can't use statements like the preceding. Instead, you must use the `newInstance` method in the `Constructor` class.



**C++ NOTE:** The `newInstance` method corresponds to the idiom of a *virtual constructor* in C++. However, virtual constructors in C++ are not a language feature but just an idiom that needs to be supported by a specialized library. The `Class` class is similar to the `type_info` class in C++, and the `getClass` method is equivalent to the `typeid` operator. The Java `Class` is quite a bit more versatile than `type_info`, though. The C++ `type_info` can only reveal a string with the name of the type, not create new objects of that type.

### A Primer on Catching Exceptions

We cover exception handling fully in Chapter 11, but in the meantime you will occasionally encounter methods that threaten to throw exceptions.

When an error occurs at runtime, a program can “throw an exception.” Throwing an exception is more flexible than terminating the program because you can provide a *handler* that “catches” the exception and deals with it.

If you don't provide a handler, the program still terminates and prints a message to the console, giving the type of the exception. You may already have seen exception reports when you accidentally used a `null` reference or overstepped the bounds of an array.

There are two kinds of exceptions: *unchecked* exceptions and *checked* exceptions. With checked exceptions, the compiler checks that you provide a handler. However, many common exceptions, such as accessing a null reference, are unchecked. The compiler does not check whether you provide a handler for these errors—after all, you should spend your mental energy on avoiding these mistakes rather than coding handlers for them.

But not all errors are avoidable. If an exception can occur despite your best efforts, then the compiler insists that you provide a handler. The `Class.forName` method is an example of a method that throws a checked exception. In Chapter 11, you will see several exception handling strategies. For now, we just show you the simplest handler implementation.

Place one or more statements that might throw checked exceptions inside a try block. Then provide the handler code in the catch clause.

```
try
{
    statements that might throw exceptions
}
catch(Exception e)
{
    handler action
}
```

Here is an example:

```
try
{
    String name = . . . ; // get class name
    Class c1 = Class.forName(name); // might throw exception
    . . . // do something with c1
}
catch(Exception e)
{
    e.printStackTrace();
}
```

If the class name doesn't exist, the remainder of the code in the try block is skipped and the program enters the catch clause. (Here, we print a stack trace by using the `printStackTrace` method of the `Throwable` class. `Throwable` is the superclass of the `Exception` class.) If none of the methods in the try block throws an exception, the handler code in the catch clause is skipped.

You only need to supply an exception handler for checked exceptions. It is easy to find out which methods throw checked exceptions—the compiler will complain whenever you call a method that threatens to throw a checked exception and you don't supply a handler.

#### API `java.lang.Class` 1.0

- `static Class.forName(String className)`  
returns the `Class` object representing the class with name `className`.
- `Object newInstance()`  
returns a new instance of this class.

#### API `java.lang.reflect.Constructor` 1.1

- `Object newInstance(Object[] args)`  
constructs a new instance of the constructor's declaring class.  
*Parameters:*    `args`            the parameters supplied to the constructor. See the section on reflection for more information on how to supply parameters.

**API** `java.lang.Throwable` 1.0

- `void printStackTrace()`  
prints the `Throwable` object and the stack trace to the standard error stream.

**Using Reflection to Analyze the Capabilities of Classes**

Here is a brief overview of the most important parts of the reflection mechanism for letting you examine the structure of a class.

The three classes `Field`, `Method`, and `Constructor` in the `java.lang.reflect` package describe the fields, methods, and constructors of a class, respectively. All three classes have a method called `getName` that returns the name of the item. The `Field` class has a method `getType` that returns an object, again of type `Class`, that describes the field type. The `Method` and `Constructor` classes have methods to report the types of the parameters, and the `Method` class also reports the return type. All three of these classes also have a method called `getModifiers` that returns an integer, with various bits turned on and off, that describes the modifiers used, such as `public` and `static`. You can then use the static methods in the `Modifier` class in the `java.lang.reflect` package to analyze the integer that `getModifiers` returns. Use methods like `isPublic`, `isPrivate`, or `isFinal` in the `Modifier` class to tell whether a method or constructor was `public`, `private`, or `final`. All you have to do is have the appropriate method in the `Modifier` class work on the integer that `getModifiers` returns. You can also use the `Modifier.toString` method to print the modifiers.

The `getFields`, `getMethods`, and `getConstructors` methods of the `Class` class return arrays of the *public* fields, methods, and constructors that the class supports. This includes *public* members of superclasses. The `getDeclaredFields`, `getDeclaredMethods`, and `getDeclaredConstructors` methods of the `Class` class return arrays consisting of all fields, methods, and constructors that are declared in the class. This includes *private* and *protected* members, but not members of superclasses.

Listing 5–6 shows you how to print out all information about a class. The program prompts you for the name of a class and then writes out the signatures of all methods and constructors as well as the names of all data fields of a class. For example, if you enter

```
java.lang.Double
```

the program prints

```
public class java.lang.Double extends java.lang.Number
{
    public java.lang.Double(java.lang.String);
    public java.lang.Double(double);

    public int hashCode();
    public int compareTo(java.lang.Object);
    public int compareTo(java.lang.Double);
    public boolean equals(java.lang.Object);
    public java.lang.String toString();
    public static java.lang.String toString(double);
    public static java.lang.Double valueOf(java.lang.String);
    public static boolean isNaN(double);
    public boolean isNaN();
    public static boolean isInfinite(double);
    public boolean isInfinite();
```

```

    public byte byteValue();
    public short shortValue();
    public int intValue();
    public long longValue();
    public float floatValue();
    public double doubleValue();
    public static double parseDouble(java.lang.String);
    public static native long doubleToLongBits(double);
    public static native long doubleToRawLongBits(double);
    public static native double longBitsToDouble(long);

    public static final double POSITIVE_INFINITY;
    public static final double NEGATIVE_INFINITY;
    public static final double NaN;
    public static final double MAX_VALUE;
    public static final double MIN_VALUE;
    public static final java.lang.Class TYPE;
    private double value;
    private static final long serialVersionUID;
}

```

What is remarkable about this program is that it can analyze any class that the Java interpreter can load, not just the classes that were available when the program was compiled. We use this program in the next chapter to peek inside the inner classes that the Java compiler generates automatically.

**Listing 5-6** ReflectionTest.java

```

1. import java.util.*;
2. import java.lang.reflect.*;
3.
4. /**
5.  * This program uses reflection to print all features of a class.
6.  * @version 1.1 2004-02-21
7.  * @author Cay Horstmann
8.  */
9. public class ReflectionTest
10. {
11.     public static void main(String[] args)
12.     {
13.         // read class name from command line args or user input
14.         String name;
15.         if (args.length > 0) name = args[0];
16.         else
17.         {
18.             Scanner in = new Scanner(System.in);
19.             System.out.println("Enter class name (e.g. java.util.Date): ");
20.             name = in.next();
21.         }
22.
23.         try
24.         {

```

**Listing 5-6** ReflectionTest.java (continued)

```
25.     // print class name and superclass name (if != Object)
26.     Class c1 = Class.forName(name);
27.     Class supercl = c1.getSuperclass();
28.     String modifiers = Modifier.toString(c1.getModifiers());
29.     if (modifiers.length() > 0) System.out.print(modifiers + " ");
30.     System.out.print("class " + name);
31.     if (supercl != null && supercl != Object.class) System.out.print(" extends "
32.         + supercl.getName());
33.
34.     System.out.print("\n\n");
35.     printConstructors(c1);
36.     System.out.println();
37.     printMethods(c1);
38.     System.out.println();
39.     printFields(c1);
40.     System.out.println(";");
41. }
42. catch (ClassNotFoundException e)
43. {
44.     e.printStackTrace();
45. }
46. System.exit(0);
47. }
48.
49. /**
50.  * Prints all constructors of a class
51.  * @param c1 a class
52.  */
53. public static void printConstructors(Class c1)
54. {
55.     Constructor[] constructors = c1.getDeclaredConstructors();
56.
57.     for (Constructor c : constructors)
58.     {
59.         String name = c.getName();
60.         System.out.print(" ");
61.         String modifiers = Modifier.toString(c.getModifiers());
62.         if (modifiers.length() > 0) System.out.print(modifiers + " ");
63.         System.out.print(name + "(");
64.
65.         // print parameter types
66.         Class[] paramTypes = c.getParameterTypes();
67.         for (int j = 0; j < paramTypes.length; j++)
68.         {
69.             if (j > 0) System.out.print(", ");
70.             System.out.print(paramTypes[j].getName());
71.         }
72.         System.out.println(");");
73.     }
```

**Listing 5-6** ReflectionTest.java (continued)

```
74.     }
75.
76.     /**
77.      * Prints all methods of a class
78.      * @param cl a class
79.      */
80.     public static void printMethods(Class cl)
81.     {
82.         Method[] methods = cl.getDeclaredMethods();
83.
84.         for (Method m : methods)
85.         {
86.             Class retType = m.getReturnType();
87.             String name = m.getName();
88.
89.             System.out.print(" ");
90.             // print modifiers, return type, and method name
91.             String modifiers = Modifier.toString(m.getModifiers());
92.             if (modifiers.length() > 0) System.out.print(modifiers + " ");
93.             System.out.print(retType.getName() + " " + name + "(");
94.
95.             // print parameter types
96.             Class[] paramTypes = m.getParameterTypes();
97.             for (int j = 0; j < paramTypes.length; j++)
98.             {
99.                 if (j > 0) System.out.print(", ");
100.                System.out.print(paramTypes[j].getName());
101.            }
102.            System.out.println(")");
103.        }
104.    }
105.
106.    /**
107.     * Prints all fields of a class
108.     * @param cl a class
109.     */
110.    public static void printFields(Class cl)
111.    {
112.        Field[] fields = cl.getDeclaredFields();
113.
114.        for (Field f : fields)
115.        {
116.            Class type = f.getType();
117.            String name = f.getName();
118.            System.out.print(" ");
119.            String modifiers = Modifier.toString(f.getModifiers());
120.            if (modifiers.length() > 0) System.out.print(modifiers + " ");
121.            System.out.println(type.getName() + " " + name + ";");
122.        }
123.    }
124. }
```



**API** `java.lang.Class` 1.0

- `Field[] getFields()` 1.1
- `Field[] getDeclaredFields()` 1.1  
`getFields` returns an array containing `Field` objects for the public fields of this class or its superclasses; `getDeclaredField` returns an array of `Field` objects for all fields of this class. The methods return an array of length 0 if there are no such fields or if the `Class` object represents a primitive or array type.
- `Method[] getMethods()` 1.1
- `Method[] getDeclaredMethods()` 1.1  
returns an array containing `Method` objects: `getMethods` returns public methods and includes inherited methods; `getDeclaredMethods` returns all methods of this class or interface but does not include inherited methods.
- `Constructor[] getConstructors()` 1.1
- `Constructor[] getDeclaredConstructors()` 1.1  
returns an array containing `Constructor` objects that give you all the public constructors (for `getConstructors`) or all constructors (for `getDeclaredConstructors`) of the class represented by this `Class` object.

**API** `java.lang.reflect.Field` 1.1**API** `java.lang.reflect.Method` 1.1**API** `java.lang.reflect.Constructor` 1.1

- `Class getDeclaringClass()`  
returns the `Class` object for the class that defines this constructor, method, or field.
- `Class[] getExceptionTypes()` (in `Constructor` and `Method` classes)  
returns an array of `Class` objects that represent the types of the exceptions thrown by the method.
- `int getModifiers()`  
returns an integer that describes the modifiers of this constructor, method, or field. Use the methods in the `Modifier` class to analyze the return value.
- `String getName()`  
returns a string that is the name of the constructor, method, or field.
- `Class[] getParameterTypes()` (in `Constructor` and `Method` classes)  
returns an array of `Class` objects that represent the types of the parameters.
- `Class getReturnType()` (in `Method` classes)  
returns a `Class` object that represents the return type.

**API** `java.lang.reflect.Modifier` 1.1

- `static String toString(int modifiers)`  
returns a string with the modifiers that correspond to the bits set in `modifiers`.

- `static boolean isAbstract(int modifiers)`
  - `static boolean isFinal(int modifiers)`
  - `static boolean isInterface(int modifiers)`
  - `static boolean isNative(int modifiers)`
  - `static boolean isPrivate(int modifiers)`
  - `static boolean isProtected(int modifiers)`
  - `static boolean isPublic(int modifiers)`
  - `static boolean isStatic(int modifiers)`
  - `static boolean isStrict(int modifiers)`
  - `static boolean isSynchronized(int modifiers)`
  - `static boolean isVolatile(int modifiers)`
- tests the bit in the `modifiers` value that corresponds to the modifier in the method name.

### Using Reflection to Analyze Objects at Runtime

In the preceding section, we saw how we can find out the *names* and *types* of the data fields of any object:

- Get the corresponding `Class` object.
- Call `getDeclaredFields` on the `Class` object.

In this section, we go one step further and actually look at the *contents* of the data fields. Of course, it is easy to look at the contents of a specific field of an object whose name and type are known when you write a program. But reflection lets you look at fields of objects that were not known at compile time.

The key method to achieve this examination is the `get` method in the `Field` class. If `f` is an object of type `Field` (for example, one obtained from `getDeclaredFields`) and `obj` is an object of the class of which `f` is a field, then `f.get(obj)` returns an object whose value is the current value of the field of `obj`. This is all a bit abstract, so let's run through an example.

```
Employee harry = new Employee("Harry Hacker", 35000, 10, 1, 1989);
Class c1 = harry.getClass();
// the class object representing Employee
Field f = c1.getDeclaredField("name");
// the name field of the Employee class
Object v = f.get(harry);
// the value of the name field of the harry object
// i.e., the String object "Harry Hacker"
```

Actually, there is a problem with this code. Because the `name` field is a private field, the `get` method will throw an `IllegalAccessException`. You can only use the `get` method to get the values of accessible fields. The security mechanism of Java lets you find out what fields any object has, but it won't let you read the values of those fields unless you have access permission.

The default behavior of the reflection mechanism is to respect Java access control. However, if a Java program is not controlled by a security manager that disallows it, you can override access control. To do this, invoke the `setAccessible` method on a `Field`, `Method`, or `Constructor` object. For example:

```
f.setAccessible(true); // now OK to call f.get(harry);
```

The `setAccessible` method is a method of the `AccessibleObject` class, the common superclass of the `Field`, `Method`, and `Constructor` classes. This feature is provided for debuggers, persistent storage, and similar mechanisms. We use it for a generic `toString` method later in this section.

There is another issue with the `get` method that we need to deal with. The `name` field is a `String`, and so it is not a problem to return the value as an `Object`. But suppose we want to look at the `salary` field. That is a `double`, and in Java, number types are not objects. To handle this, you can either use the `getDouble` method of the `Field` class, or you can call `get`, whereby the reflection mechanism automatically wraps the field value into the appropriate wrapper class, in this case, `Double`.

Of course, you can also set the values that you can get. The call `f.set(obj, value)` sets the field represented by `f` of the object `obj` to the new value.

Listing 5-7 shows how to write a generic `toString` method that works for *any* class. It uses `getDeclaredFields` to obtain all data fields. It then uses the `setAccessible` convenience method to make all fields accessible. For each field, it obtains the name and the value. Listing 5-7 turns each value into a string by recursively invoking `toString`.

```
class ObjectAnalyzer
{
    public String toString(Object obj)
    {
        Class c1 = obj.getClass();
        . . .
        String r = c1.getName();
        // inspect the fields of this class and all superclasses
        do
        {
            r += "[";
            Field[] fields = c1.getDeclaredFields();
            AccessibleObject.setAccessible(fields, true);
            // get the names and values of all fields
            for (Field f : fields)
            {
                if (!Modifier.isStatic(f.getModifiers()))
                {
                    if (!r.endsWith("[") r += " ";
                    r += f.getName() + "=";
                    try
                    {
                        Object val = f.get(obj);
                        r += toString(val);
                    }
                    catch (Exception e) { e.printStackTrace(); }
                }
            }
            r += "]";
            c1 = c1.getSuperclass();
        }
        while (c1 != null);
        return r;
    }
    . . .
}
```

The complete code in Listing 5–7 needs to address a couple of complexities. Cycles of references could cause an infinite recursion. Therefore, the `ObjectAnalyzer` keeps track of objects that were already visited. Also, to peek inside arrays, you need a different approach. You’ll learn about the details in the next section.

You can use this `toString` method to peek inside any object. For example, the call

```
ArrayList<Integer> squares = new ArrayList<Integer>();
for (int i = 1; i <= 5; i++) squares.add(i * i);
System.out.println(new ObjectAnalyzer().toString(squares));
```

yields the printout

```
java.util.ArrayList[elementData=class java.lang.Object[]{java.lang.Integer[value=1][[]],
java.lang.Integer[value=4][[]],java.lang.Integer[value=9][[]],java.lang.Integer[value=16][[]],
java.lang.Integer[value=25][[]],null,null,null,null,null},size=5][modCount=5][[]]
```

You can use this generic `toString` method to implement the `toString` methods of your own classes, like this:

```
public String toString()
{
    return new ObjectAnalyzer().toString(this);
}
```

This is a hassle-free method for supplying a `toString` method that you may find useful in your own programs.

#### Listing 5–7 ObjectAnalyzerTest.java

```
1. import java.lang.reflect.*;
2. import java.util.*;
3.
4. /**
5.  * This program uses reflection to spy on objects.
6.  * @version 1.11 2004-02-21
7.  * @author Cay Horstmann
8.  */
9. public class ObjectAnalyzerTest
10. {
11.     public static void main(String[] args)
12.     {
13.         ArrayList<Integer> squares = new ArrayList<Integer>();
14.         for (int i = 1; i <= 5; i++)
15.             squares.add(i * i);
16.         System.out.println(new ObjectAnalyzer().toString(squares));
17.     }
18. }
19.
20. class ObjectAnalyzer
21. {
```

**Listing 5-7** ObjectAnalyzerTest.java (continued)

```
22.  /**
23.   * Converts an object to a string representation that lists all fields.
24.   * @param obj an object
25.   * @return a string with the object's class name and all field names and
26.   * values
27.   */
28.  public String toString(Object obj)
29.  {
30.      if (obj == null) return "null";
31.      if (visited.contains(obj)) return "...";
32.      visited.add(obj);
33.      Class cl = obj.getClass();
34.      if (cl == String.class) return (String) obj;
35.      if (cl.isArray())
36.      {
37.          String r = cl.getComponentType() + "[]{";
38.          for (int i = 0; i < Array.getLength(obj); i++)
39.          {
40.              if (i > 0) r += ",";
41.              Object val = Array.get(obj, i);
42.              if (cl.getComponentType().isPrimitive()) r += val;
43.              else r += toString(val);
44.          }
45.          return r + "}";
46.      }
47.
48.      String r = cl.getName();
49.      // inspect the fields of this class and all superclasses
50.      do
51.      {
52.          r += "[";
53.          Field[] fields = cl.getDeclaredFields();
54.          AccessibleObject.setAccessible(fields, true);
55.          // get the names and values of all fields
56.          for (Field f : fields)
57.          {
58.              if (!Modifier.isStatic(f.getModifiers()))
59.              {
60.                  if (!r.endsWith("[") r += ",";
61.                  r += f.getName() + "=";
62.                  try
63.                  {
64.                      Class t = f.getType();
65.                      Object val = f.get(obj);
66.                      if (t.isPrimitive()) r += val;
67.                      else r += toString(val);
68.                  }
69.                  catch (Exception e)
```

**Listing 5-7** ObjectAnalyzerTest.java (continued)

```
70.         {
71.             e.printStackTrace();
72.         }
73.     }
74. }
75.     r += " ";
76.     cl = cl.getSuperclass();
77. }
78. while (cl != null);
79.
80.     return r;
81. }
82.
83. private ArrayList<Object> visited = new ArrayList<Object>();
84. }
```

**API** java.lang.reflect.AccessibleObject 1.2

- void setAccessible(boolean flag)  
sets the accessibility flag for this reflection object. A value of true indicates that Java language access checking is suppressed and that the private properties of the object can be queried and set.
- boolean isAccessible()  
gets the value of the accessibility flag for this reflection object.
- static void setAccessible(AccessibleObject[] array, boolean flag)  
is a convenience method to set the accessibility flag for an array of objects.

**API** java.lang.Class 1.1

- Field getField(String name)
- Field[] getFields()  
gets the public field with the given name, or an array of all fields.
- Field getDeclaredField(String name)
- Field[] getDeclaredFields()  
gets the field that is declared in this class with the given name, or an array of all fields.

**API** java.lang.reflect.Field 1.1

- Object get(Object obj)  
gets the value of the field described by this Field object in the object obj.
- void set(Object obj, Object newValue)  
sets the field described by this Field object in the object obj to a new value.

### Using Reflection to Write Generic Array Code

The `Array` class in the `java.lang.reflect` package allows you to create arrays dynamically. For example, when you use this feature with the `arraycopy` method from Chapter 3, you can dynamically expand an existing array while preserving the current contents.

The problem we want to solve is pretty typical. Suppose you have an array of some type that is full and you want to grow it. And suppose you are sick of writing the grow-and-copy code by hand. You want to write a generic method to grow an array.

```
Employee[] a = new Employee[100];
...
// array is full
a = (Employee[]) arrayGrow(a);
```

How can we write such a generic method? It helps that an `Employee[]` array can be converted to an `Object[]` array. That sounds promising. Here is a first attempt to write a generic method. We simply grow the array by 10% + 10 elements (because the 10 percent growth is not substantial enough for small arrays).

```
static Object[] badArrayGrow(Object[] a) // not useful
{
    int newLength = a.length * 11 / 10 + 10;
    Object[] newArray = new Object[newLength];
    System.arraycopy(a, 0, newArray, 0, a.length);
    return newArray;
}
```

However, there is a problem with actually *using* the resulting array. The type of array that this code returns is an array of *objects* (`Object[]`) because we created the array using the line of code

```
new Object[newLength]
```

An array of objects *cannot* be cast to an array of employees (`Employee[]`). Java would generate a `ClassCastException` at runtime. The point is, as we mentioned earlier, that a Java array remembers the type of its entries, that is, the element type used in the `new` expression that created it. It is legal to cast an `Employee[]` temporarily to an `Object[]` array and then cast it back, but an array that started its life as an `Object[]` array can never be cast into an `Employee[]` array. To write this kind of generic array code, we need to be able to make a new array of the *same* type as the original array. For this, we need the methods of the `Array` class in the `java.lang.reflect` package. The key is the static `newInstance` method of the `Array` class that constructs a new array. You must supply the type for the entries and the desired length as parameters to this method.

```
Object newArray = Array.newInstance(componentType, newLength);
```

To actually carry this out, we need to get the length and component type of the new array.

We obtain the length by calling `Array.getLength(a)`. The static `getLength` method of the `Array` class returns the length of any array. To get the component type of the new array:

1. First, get the class object of `a`.
2. Confirm that it is indeed an array.
3. Use the `getComponentType` method of the `Class` class (which is defined only for class objects that represent arrays) to find the right type for the array.

Why is `getLength` a method of `Array` but `getComponentType` a method of `Class`? We don't know—the distribution of the reflection methods seems a bit ad hoc at times.

Here's the code:

```
static Object goodArrayGrow(Object a) // useful
{
    Class cl = a.getClass();
    if (!cl.isArray()) return null;
    Class componentType = cl.getComponentType();
    int length = Array.getLength(a);
    int newLength = length * 11 / 10 + 10;
    Object newArray = Array.newInstance(componentType, newLength);
    System.arraycopy(a, 0, newArray, 0, length);
    return newArray;
}
```

Note that this `arrayGrow` method can be used to grow arrays of any type, not just arrays of objects.

```
int[] a = { 1, 2, 3, 4 };
a = (int[]) goodArrayGrow(a);
```

To make this possible, the parameter of `goodArrayGrow` is declared to be of type `Object`, *not an array of objects* (`Object[]`). The integer array type `int[]` can be converted to an `Object`, but not to an array of objects!

Listing 5–8 shows both array grow methods in action. Note that the cast of the return value of `badArrayGrow` will throw an exception.



**NOTE:** We present this program to illustrate how to work with arrays through reflection. If you just want to grow an array, use the `copyOf` method in the `Arrays` class.

```
Employee[] a = new Employee[100];
. . .
// array is full
a = Arrays.copyOf(a, a.length * 11 / 10 + 10);
```

#### Listing 5–8 ArrayGrowTest.java

```
1. import java.lang.reflect.*;
2.
3. /**
4.  * This program demonstrates the use of reflection for manipulating arrays.
5.  * @version 1.01 2004-02-21
6.  * @author Cay Horstmann
7.  */
8. public class ArrayGrowTest
9. {
10.     public static void main(String[] args)
11.     {
12.         int[] a = { 1, 2, 3 };
13.         a = (int[]) goodArrayGrow(a);
```



**Listing 5-8** ArrayGrowTest.java (continued)

```
14.     arrayPrint(a);
15.
16.     String[] b = { "Tom", "Dick", "Harry" };
17.     b = (String[]) goodArrayGrow(b);
18.     arrayPrint(b);
19.
20.     System.out.println("The following call will generate an exception.");
21.     b = (String[]) badArrayGrow(b);
22. }
23.
24. /**
25.  * This method attempts to grow an array by allocating a new array and copying all elements.
26.  * @param a the array to grow
27.  * @return a larger array that contains all elements of a. However, the returned array has
28.  * type Object[], not the same type as a
29.  */
30. static Object[] badArrayGrow(Object[] a)
31. {
32.     int newLength = a.length * 11 / 10 + 10;
33.     Object[] newArray = new Object[newLength];
34.     System.arraycopy(a, 0, newArray, 0, a.length);
35.     return newArray;
36. }
37.
38. /**
39.  * This method grows an array by allocating a new array of the same type and
40.  * copying all elements.
41.  * @param a the array to grow. This can be an object array or a primitive
42.  * type array
43.  * @return a larger array that contains all elements of a.
44.  */
45. static Object goodArrayGrow(Object a)
46. {
47.     Class c1 = a.getClass();
48.     if (!c1.isArray()) return null;
49.     Class componentType = c1.getComponentType();
50.     int length = Array.getLength(a);
51.     int newLength = length * 11 / 10 + 10;
52.
53.     Object newArray = Array.newInstance(componentType, newLength);
54.     System.arraycopy(a, 0, newArray, 0, length);
55.     return newArray;
56. }
57.
58. /**
59.  * A convenience method to print all elements in an array
60.  * @param a the array to print. It can be an object array or a primitive type array
61.  */
62. static void arrayPrint(Object a)
```

**Listing 5-8** ArrayGrowTest.java (continued)

```

63.  {
64.      Class c1 = a.getClass();
65.      if (!c1.isArray()) return;
66.      Class componentType = c1.getComponentType();
67.      int length = Array.getLength(a);
68.      System.out.print(componentType.getName() + "[" + length + "] = { ");
69.      for (int i = 0; i < Array.getLength(a); i++)
70.          System.out.print(Array.get(a, i) + " ");
71.      System.out.println("}");
72.  }
73. }

```

**API** java.lang.reflect.Array 1.1

- static Object get(Object array, int index)
- static xxx getXxx(Object array, int index)  
(xxx is one of the primitive types boolean, byte, char, double, float, int, long, short.) These methods return the value of the given array that is stored at the given index.
- static void set(Object array, int index, Object newValue)
- static setXxx(Object array, int index, xxx newValue)  
(xxx is one of the primitive types boolean, byte, char, double, float, int, long, short.) These methods store a new value into the given array at the given index.
- static int getLength(Object array)  
returns the length of the given array.
- static Object newInstance(Class componentType, int length)
- static Object newInstance(Class componentType, int[] lengths)  
returns a new array of the given component type with the given dimensions.

**Method Pointers!**

On the surface, Java does not have method pointers—ways of giving the location of a method to another method so that the second method can invoke it later. In fact, the designers of Java have said that method pointers are dangerous and error prone and that Java *interfaces* (discussed in the next chapter) are a superior solution. However, as of Java 1.1, it turns out that Java does have method pointers, as a (perhaps accidental) by-product of the reflection package.



**NOTE:** Among the nonstandard language extensions that Microsoft added to its Java derivative J++ (and its successor, C#) is another method pointer type, called a *delegate*, that is different from the Method class that we discuss in this section. However, inner classes (which we will introduce in the next chapter) are a more useful construct than delegates.

To see method pointers at work, recall that you can inspect a field of an object with the get method of the Field class. Similarly, the Method class has an invoke method that lets you call the method that is wrapped in the current Method object. The signature for the invoke method is

```
Object invoke(Object obj, Object... args)
```

The first parameter is the implicit parameter, and the remaining objects provide the explicit parameters. (Before Java SE 5.0, you had to pass an array of objects or `null` if the method had no explicit parameters.)

For a static method, the first parameter is ignored—you can set it to `null`.

For example, if `m1` represents the `getName` method of the `Employee` class, the following code shows how you can call it:

```
String n = (String) m1.invoke(harry);
```

As with the `get` and `set` methods of the `Field` type, there's a problem if the parameter or return type is not a class but a primitive type. You either rely on autoboxing or, before Java SE 5.0, wrap primitive types into their corresponding wrappers.

Conversely, if the return type is a primitive type, the `invoke` method will return the wrapper type instead. For example, suppose that `m2` represents the `getSalary` method of the `Employee` class. Then, the returned object is actually a `Double`, and you must cast it accordingly. As of Java SE 5.0, automatic unboxing takes care of the rest.

```
double s = (Double) m2.invoke(harry);
```

How do you obtain a `Method` object? You can, of course, call `getDeclaredMethods` and search through the returned array of `Method` objects until you find the method that you want. Or, you can call the `getMethod` method of the `Class` class. This is similar to the `getField` method that takes a string with the field name and returns a `Field` object. However, there may be several methods with the same name, so you need to be careful that you get the right one. For that reason, you must also supply the parameter types of the desired method. The signature of `getMethod` is

```
Method getMethod(String name, Class... parameterTypes)
```

For example, here is how you can get method pointers to the `getName` and `raiseSalary` methods of the `Employee` class:

```
Method m1 = Employee.class.getMethod("getName");
Method m2 = Employee.class.getMethod("raiseSalary", double.class);
```

(Before Java SE 5.0, you had to package the `Class` objects into an array or to supply `null` if there were no parameters.)

Now that you have seen the rules for using `Method` objects, let's put them to work. Listing 5-9 is a program that prints a table of values for a mathematical function such as `Math.sqrt` or `Math.sin`. The printout looks like this:

```
public static native double java.lang.Math.sqrt(double)
1.0000 | 1.0000
2.0000 | 1.4142
3.0000 | 1.7321
4.0000 | 2.0000
5.0000 | 2.2361
6.0000 | 2.4495
7.0000 | 2.6458
8.0000 | 2.8284
9.0000 | 3.0000
10.0000 | 3.1623
```

The code for printing a table is, of course, independent of the actual function that is being tabulated.

```
double dx = (to - from) / (n - 1);
for (double x = from; x <= to; x += dx)
{
    double y = (Double) f.invoke(null, x);
    System.out.printf("%10.4f | %10.4f%n", x, y);
}
```

Here, `f` is an object of type `Method`. The first parameter of `invoke` is `null` because we are calling a static method.

To tabulate the `Math.sqrt` function, we set `f` to

```
Math.class.getMethod("sqrt", double.class)
```

That is the method of the `Math` class that has the name `sqrt` and a single parameter of type `double`.

Listing 5-9 shows the complete code of the generic tabulator and a couple of test runs.

#### Listing 5-9 MethodPointerTest.java

```
1. import java.lang.reflect.*;
2.
3. /**
4.  * This program shows how to invoke methods through reflection.
5.  * @version 1.1 2004-02-21
6.  * @author Cay Horstmann
7.  */
8. public class MethodPointerTest
9. {
10.     public static void main(String[] args) throws Exception
11.     {
12.         // get method pointers to the square and sqrt methods
13.         Method square = MethodPointerTest.class.getMethod("square", double.class);
14.         Method sqrt = Math.class.getMethod("sqrt", double.class);
15.
16.         // print tables of x- and y-values
17.
18.         printTable(1, 10, 10, square);
19.         printTable(1, 10, 10, sqrt);
20.     }
21.
22.     /**
23.      * Returns the square of a number
24.      * @param x a number
25.      * @return x squared
26.      */
27.     public static double square(double x)
28.     {
29.         return x * x;
30.     }
}
```

**Listing 5-9** MethodPointerTest.java (continued)

```
31.
32.  /**
33.   * Prints a table with x- and y-values for a method
34.   * @param from the lower bound for the x-values
35.   * @param to the upper bound for the x-values
36.   * @param n the number of rows in the table
37.   * @param f a method with a double parameter and double return value
38.   */
39. public static void printTable(double from, double to, int n, Method f)
40. {
41.     // print out the method as table header
42.     System.out.println(f);
43.
44.     double dx = (to - from) / (n - 1);
45.
46.     for (double x = from; x <= to; x += dx)
47.     {
48.         try
49.         {
50.             double y = (Double) f.invoke(null, x);
51.             System.out.printf("%10.4f | %10.4f%n", x, y);
52.         }
53.         catch (Exception e)
54.         {
55.             e.printStackTrace();
56.         }
57.     }
58. }
59. }
```

As this example shows clearly, you can do anything with `Method` objects that you can do with function pointers in C (or delegates in C#). Just as in C, this style of programming is usually quite inconvenient and always error prone. What happens if you invoke a method with the wrong parameters? The `invoke` method throws an exception.

Also, the parameters and return values of `invoke` are necessarily of type `Object`. That means you must cast back and forth a lot. As a result, the compiler is deprived of the chance to check your code. Therefore, errors surface only during testing, when they are more tedious to find and fix. Moreover, code that uses reflection to get at method pointers is significantly slower than code that simply calls methods directly.

For that reason, we suggest that you use `Method` objects in your own programs only when absolutely necessary. Using interfaces and inner classes (the subject of the next chapter) is almost always a better idea. In particular, we echo the developers of Java and suggest not using `Method` objects for callback functions. Using interfaces for the callbacks (see the next chapter as well) leads to code that runs faster and is a lot more maintainable.

**API** `java.lang.reflect.Method` 1.1

- `public Object invoke(Object implicitParameter, Object[] explicitParameters)`  
invokes the method described by this object, passing the given parameters and returning the value that the method returns. For static methods, pass `null` as the implicit parameter. Pass primitive type values by using wrappers. Primitive type return values must be unwrapped.

**Design Hints for Inheritance**

We want to end this chapter with some hints that we have found useful when using inheritance.

1. *Place common operations and fields in the superclass.*

This is why we put the `name` field into the `Person` class rather than replicating it in the `Employee` and `Student` classes.

2. *Don't use protected fields.*

Some programmers think it is a good idea to define most instance fields as `protected`, “just in case,” so that subclasses can access these fields if they need to. However, the `protected` mechanism doesn't give much protection, for two reasons. First, the set of subclasses is unbounded—anyone can form a subclass of your classes and then write code that directly accesses `protected` instance fields, thereby breaking encapsulation. And second, in the Java programming language, all classes in the same package have access to `protected` fields, whether or not they are subclasses.

However, `protected` methods can be useful to indicate methods that are not ready for general use and should be redefined in subclasses. The `clone` method is a good example.

3. *Use inheritance to model the “is-a” relationship.*

Inheritance is a handy code-saver, and sometimes people overuse it. For example, suppose we need a `Contractor` class. Contractors have names and hire dates, but they do not have salaries. Instead, they are paid by the hour, and they do not stay around long enough to get a raise. There is the temptation to form a subclass `Contractor` from `Employee` and add an `hourlyWage` field.

```
class Contractor extends Employee
{
    . . .
    private double hourlyWage;
}
```

This is *not* a good idea, however, because now each contractor object has both a salary and hourly wage field. It will cause you no end of grief when you implement methods for printing paychecks or tax forms. You will end up writing more code than you would have by not inheriting in the first place.

The contractor/employee relationship fails the “is-a” test. A contractor is not a special case of an employee.

4. *Don't use inheritance unless all inherited methods make sense.*

Suppose we want to write a `Holiday` class. Surely every holiday is a day, and days can be expressed as instances of the `GregorianCalendar` class, so we can use inheritance.

```
class Holiday extends GregorianCalendar { . . . }
```

Unfortunately, the set of holidays is not *closed* under the inherited operations. One of the public methods of `GregorianCalendar` is `add`. And `add` can turn holidays into nonholidays:

```
Holiday christmas;
christmas.add(Calendar.DAY_OF_MONTH, 12);
```

Therefore, inheritance is not appropriate in this example.

5. *Don't change the expected behavior when you override a method.*

The substitution principle applies not just to syntax but, more important, to behavior. When you override a method, you should not unreasonably change its behavior. The compiler can't help you—it cannot check whether your redefinitions make sense. For example, you can “fix” the issue of the `add` method in the `Holiday` class by redefining `add`, perhaps to do nothing, or to throw an exception, or to move on to the next holiday.

However, such a fix violates the substitution principle. The sequence of statements

```
int d1 = x.get(Calendar.DAY_OF_MONTH);
x.add(Calendar.DAY_OF_MONTH, 1);
int d2 = x.get(Calendar.DAY_OF_MONTH);
System.out.println(d2 - d1);
```

should have the *expected behavior*, no matter whether `x` is of type `GregorianCalendar` or `Holiday`.

Of course, therein lies the rub. Reasonable and unreasonable people can argue at length what the expected behavior is. For example, some authors argue that the substitution principle requires `Manager.equals` to ignore the `bonus` field because `Employee.equals` ignores it. These discussions are always pointless if they occur in a vacuum. Ultimately, what matters is that you do not circumvent the intent of the original design when you override methods in subclasses.

6. *Use polymorphism, not type information.*

Whenever you find code of the form

```
if (x is of type 1)
    action1(x);
else if (x is of type 2)
    action2(x);
```

think polymorphism.

Do `action1` and `action2` represent a common concept? If so, make the concept a method of a common superclass or interface of both types. Then, you can simply call

```
x.action();
```

and have the dynamic dispatch mechanism inherent in polymorphism launch the correct action.

Code using polymorphic methods or interface implementations is much easier to maintain and extend than code that uses multiple type tests.

7. *Don't overuse reflection.*

The reflection mechanism lets you write programs with amazing generality, by detecting fields and methods at runtime. This capability can be extremely useful for systems programming, but it is usually not appropriate in applications. Reflection is fragile—the compiler cannot help you find programming errors. Any errors are found at runtime and result in exceptions.

You have now seen how Java supports the fundamentals of object-oriented programming: classes, inheritance, and polymorphism. In the next chapter, we will tackle two advanced topics that are very important for using Java effectively: interfaces and inner classes.



---

*Chapter*

6

INTERFACES AND  
INNER CLASSES

- ▼ INTERFACES
- ▼ OBJECT CLONING
- ▼ INTERFACES AND CALLBACKS
- ▼ INNER CLASSES
- ▼ PROXIES

**Y**ou 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, called an *interface*, 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 anytime that conformance to the interface is required. After we cover interfaces, we take up cloning an object (or deep copying, as it is sometimes called). A clone of an object is a new object that has the same state as the original. In particular, you can modify the clone without affecting the original.

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

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

## Interfaces

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

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

Here is what the `Comparable` interface looks like:

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

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



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

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

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

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

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

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

Of course, there is an additional requirement that the interface cannot spell out: When calling `x.compareTo(y)`, the `compareTo` method must actually be able to compare 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 more than one method. As you will see later, interfaces can also define constants. What is more important, however, is what interfaces *cannot* supply. Interfaces never have instance fields, and the methods are never implemented in the interface. Supplying instance fields and method implementations is the job of the classes that implement the interface. You can think of an interface as being similar to an abstract class with no instance fields. However, there are some differences between these two concepts—we look at them later in some detail.

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

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

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

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

```
class Employee implements Comparable
```

Of course, now the `Employee` class needs to supply the `compareTo` method. Let's suppose that we want to compare employees by their salary. Here is a `compareTo` method that returns `-1` if the first employee's salary is less than the second employee's salary, `0` if they are equal, and `1` otherwise.

```
public int compareTo(Object otherObject)
{
    Employee other = (Employee) otherObject;
    if (salary < other.salary) return -1;
    if (salary > other.salary) return 1;
    return 0;
}
```



**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*. Then the compiler complains that you try to supply a weaker access privilege.

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

```
class Employee implements Comparable<Employee>
{
    public int compareTo(Employee other)
    {
        if (salary < other.salary) return -1;
```

```

        if (salary > other.salary) return 1;
        return 0;
    }
    . . .
}

```

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 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 that the subtraction does not overflow. If you know that the IDs are not negative or that their absolute value is at most  $(Integer.MAX\_VALUE - 1) / 2$ , you are safe.

Of course, the subtraction trick doesn't work for floating-point numbers. The difference `salary - other.salary` can round to 0 if the salaries are close together but not identical.

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:

```

// from 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, then the virtual machine throws an exception.

Listing 6–1 presents the full code for sorting an employee array.

**Listing 6–1** EmployeeSortTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates the use of the Comparable interface.
5.  * @version 1.30 2004-02-27
6.  * @author Cay Horstmann
7.  */
8. public class EmployeeSortTest
9. {
10.     public static void main(String[] args)
11.     {
12.         Employee[] staff = new Employee[3];
13.
14.         staff[0] = new Employee("Harry Hacker", 35000);
15.         staff[1] = new Employee("Carl Cracker", 75000);
16.         staff[2] = new Employee("Tony Tester", 38000);
17.
18.         Arrays.sort(staff);
19.
20.         // print out information about all Employee objects
21.         for (Employee e : staff)
22.             System.out.println("name=" + e.getName() + ",salary=" + e.getSalary());
23.     }
24. }
25.
26. class Employee implements Comparable<Employee>
27. {
28.     public Employee(String n, double s)
29.     {
30.         name = n;
31.         salary = s;
32.     }
33.
34.     public String getName()
35.     {
36.         return name;
37.     }
38.
39.     public double getSalary()
40.     {
41.         return salary;
42.     }
43.
44.     public void raiseSalary(double byPercent)
45.     {
46.         double raise = salary * byPercent / 100;
47.         salary += raise;
48.     }
}
```

**Listing 6-1** EmployeeSortTest.java (continued)

```

49.
50.  /**
51.   * Compares employees by salary
52.   * @param other another Employee object
53.   * @return a negative value if this employee has a lower salary than
54.   * otherObject, 0 if the salaries are the same, a positive value otherwise
55.   */
56.  public int compareTo(Employee other)
57.  {
58.      if (salary < other.salary) return -1;
59.      if (salary > other.salary) return 1;
60.      return 0;
61.  }
62.
63.  private String name;
64.  private double salary;
65. }


```

**API** 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.

**API** java.util.Arrays 1.2

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

 **NOTE:** According to the language standard: “The implementor must ensure  $\text{sgn}(x.\text{compareTo}(y)) = -\text{sgn}(y.\text{compareTo}(x))$  for all  $x$  and  $y$ . (This implies that  $x.\text{compareTo}(y)$  must throw an exception if  $y.\text{compareTo}(x)$  throws an exception.)” Here, “*sgn*” is the *sign* of a number:  $\text{sgn}(n)$  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.

Because `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 the employee to a manager:

```

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

```

```

    }
    ...
}

```

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

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

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

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

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

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

### 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 discuss this interface in detail in the next section.) If your class implements `Cloneable`, the `clone` method in the `Object` class will make an exact copy of your class's objects. Suppose, therefore, you want cloneability and comparability. Then you simply implement both interfaces.

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

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

### **Interfaces and Abstract Classes**

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

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

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

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



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

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

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

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

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

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



**C++ 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 “mix in” style inheritance. In the mix-in style, a primary base class describes the parent object, and additional base classes (the so-called mix-ins) may supply auxiliary characteristics. That style is similar to a Java class with a single base class and additional interfaces. However, in C++, mix-ins can add default behavior, whereas Java interfaces cannot.

## Object Cloning

When you make a copy of a variable, the original and the copy are references to the same object. (See Figure 6–1.) This means a change to either variable also affects the other.

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

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

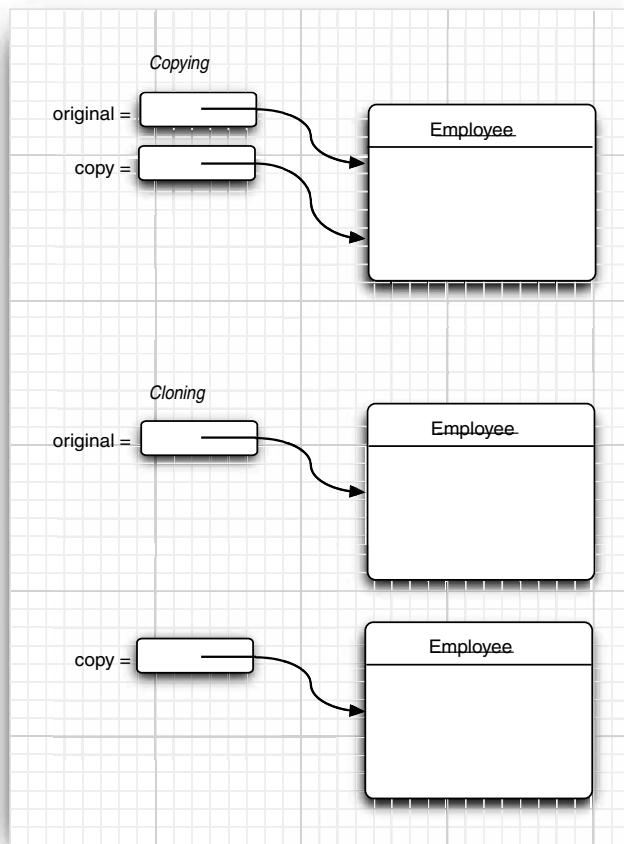
```
Employee copy = original.clone();
copy.raiseSalary(10); // OK--original unchanged
```

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

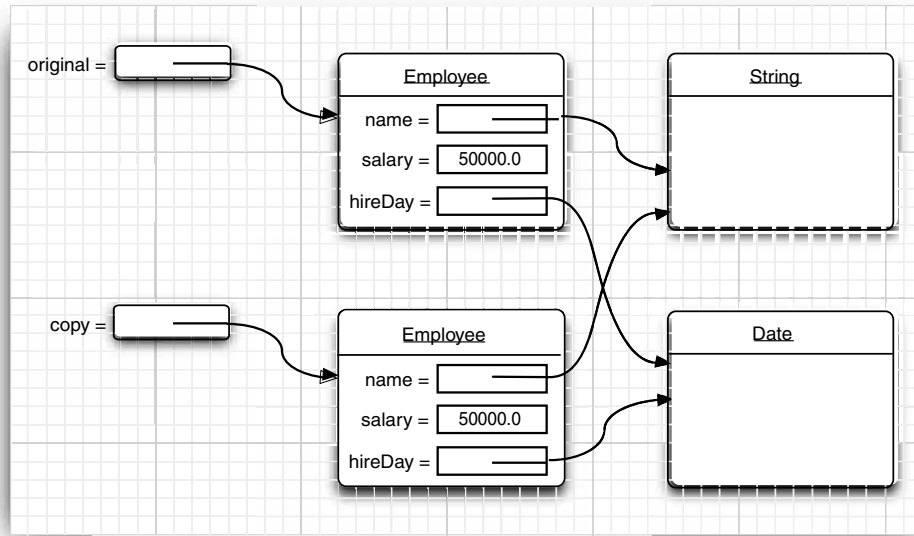
To visualize that phenomenon, let's consider the `Employee` class that was introduced in Chapter 4. Figure 6–2 shows what happens when you use the `clone` method of the `Object`

class to clone such an `Employee` object. As you can see, the default cloning operation is “shallow”—it doesn’t clone objects that are referenced inside other objects.

Does it matter if the copy is shallow? It depends. If the subobject that is 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.



**Figure 6-1 Copying and cloning**



**Figure 6-2** A shallow copy

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

For every class, you need to decide whether

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

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

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



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

In this case, the appearance of the `Cloneable` interface has nothing to do with the normal use of interfaces. In particular, it does *not* specify the `clone` method—that method is inherited from the `Object` class. The interface merely serves as a tag, indicating that the

class designer understands the cloning process. Objects are so paranoid about cloning that they generate a checked exception if an object requests cloning but does not implement that interface.



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

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

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

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

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



**NOTE:** Before Java SE 5.0, the `clone` method always had return type `Object`. The covariant return types of Java SE 5.0 let you specify the correct return type for your `clone` methods.

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

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

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

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

        return cloned;
    }
}
```

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

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

Would it be better to catch the exception instead?

```
public Employee clone()
{
    try
    {
        return 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 that 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 6–2 clones an `Employee` object, 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 = (int[]) luckyNumbers.clone();
cloned[5] = 12; // doesn't change luckyNumbers[5]
```



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

**Listing 6-2** CloneTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates cloning.
5.  * @version 1.10 2002-07-01
6.  * @author Cay Horstmann
7.  */
8. public class CloneTest
9. {
10.     public static void main(String[] args)
11.     {
12.         try
13.         {
14.             Employee original = new Employee("John Q. Public", 50000);
15.             original.setHireDay(2000, 1, 1);
16.             Employee copy = original.clone();
17.             copy.raiseSalary(10);
18.             copy.setHireDay(2002, 12, 31);
19.             System.out.println("original=" + original);
20.             System.out.println("copy=" + copy);
21.         }
22.         catch (CloneNotSupportedException e)
23.         {
24.             e.printStackTrace();
25.         }
26.     }
27. }
28.
29. class Employee implements Cloneable
30. {
31.     public Employee(String n, double s)
32.     {
33.         name = n;
34.         salary = s;
35.         hireDay = new Date();
36.     }
37.
38.     public Employee clone() throws CloneNotSupportedException
39.     {
40.         // call Object.clone()
41.         Employee cloned = (Employee) super.clone();
42.     }
43. }
```

**Listing 6-2** CloneTest.java (continued)

```
43.    // clone mutable fields
44.    cloned.hireDay = (Date) hireDay.clone();
45.
46.    return cloned;
47. }
48.
49. /**
50.  * Set the hire day to a given date.
51.  * @param year the year of the hire day
52.  * @param month the month of the hire day
53.  * @param day the day of the hire day
54.  */
55. public void setHireDay(int year, int month, int day)
56. {
57.     Date newHireDay = new GregorianCalendar(year, month - 1, day).getTime();
58.
59.     // Example of instance field mutation
60.     hireDay.setTime(newHireDay.getTime());
61. }
62.
63. public void raiseSalary(double byPercent)
64. {
65.     double raise = salary * byPercent / 100;
66.     salary += raise;
67. }
68.
69. public String toString()
70. {
71.     return "Employee[name=" + name + ",salary=" + salary + ",hireDay=" + hireDay + "];"
72. }
73.
74. private String name;
75. private double salary;
76. private Date hireDay;
77. }
```

## Interfaces and Callbacks

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

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

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

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

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

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



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

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

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

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

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

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

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

Finally, you start the timer.

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



Every 10 seconds, a message like

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

is displayed, followed by a beep.

Listing 6–3 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 in 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** TimerTest.java

```
1. /**
2.  * @version 1.00 2000-04-13
3.  * @author Cay Horstmann
4.  */
5.
6. import java.awt.*;
7. import java.awt.event.*;
8. import java.util.*;
9. import javax.swing.*;
10. import javax.swing.Timer;
11. // to resolve conflict with java.util.Timer
12.
13. public class TimerTest
14. {
15.     public static void main(String[] args)
16.     {
17.         ActionListener listener = new TimePrinter();
18.
19.         // construct a timer that calls the listener
20.         // once every 10 seconds
21.         Timer t = new Timer(10000, listener);
22.         t.start();
23.
24.         JOptionPane.showMessageDialog(null, "Quit program?");
25.         System.exit(0);
26.     }
27. }
28.
29. class TimePrinter implements ActionListener
30. {
```

**Listing 6-3** TimerTest.java (continued)

```
31. public void actionPerformed(ActionEvent event)
32. {
33.     Date now = new Date();
34.     System.out.println("At the tone, the time is " + now);
35.     Toolkit.getDefaultToolkit().beep();
36. }
37. }
```

**API** 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.

**API** 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.

**API** javax.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.

**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 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.

- Starting on page 260, you will see a simple inner class that accesses an instance field of its outer class.
- On page 263, we cover the special syntax rules for inner classes.

- Starting on page 264, we peek inside inner classes to see how they are translated into regular classes. Squeamish readers may want to skip that section.
- Starting on page 266, we discuss *local inner classes* that can access local variables of the enclosing scope.
- Starting on page 269, we introduce *anonymous inner classes* and show how they are commonly used to implement callbacks.
- Finally, starting on page 271, 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*. Because the name `Iterator` is nested inside the `LinkedList` class, it is externally known as `LinkedList::Iterator` and cannot conflict with another class called `Iterator`. In Java, this benefit is not as important because Java *packages* give the same kind of name control. Note that the `Link` class is in the *private* part of the `LinkedList` class. It is completely hidden from all other code. For that reason, it is safe to make its data fields public. They can be accessed by the methods of the `LinkedList` class (which has a legitimate need to access them), and they are not visible elsewhere. In Java, this kind of control was not possible until inner classes were introduced.

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

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

**Use of an Inner Class to Access Object State**

The syntax for inner classes is rather complex. For that reason, we use 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
{
    public TalkingClock(int interval, boolean beep) { . . . }
    public void start() { . . . }

    private int interval;
    private boolean beep;

    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.

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

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

For this to work, an object of an inner class always gets an implicit reference to the object that created it. (See Figure 6–3.)

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

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

```

    if (outer.beep) Toolkit.getDefaultToolkit().beep();
}

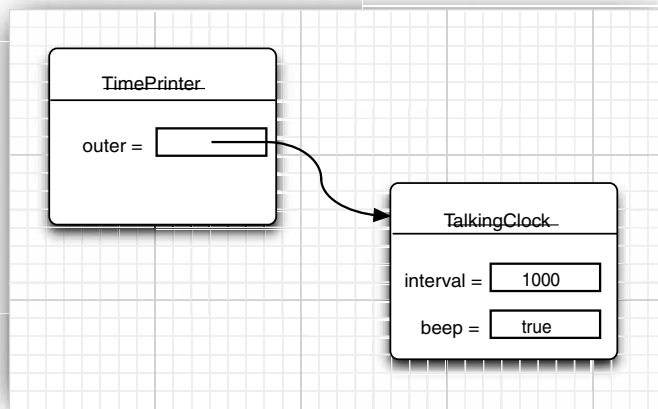
```

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

```

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

```



**Figure 6-3 An inner class object has a reference to an outer class object**

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

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

```

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

```

Listing 6-4 shows the complete program that tests the inner class. Have another look at the access control. Had the `TimePrinter` class been a regular class, then 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 6-4** InnerClassTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.util.*;
4. import javax.swing.*;
5. import javax.swing.Timer;
6.
7. /**
8.  * This program demonstrates the use of inner classes.
9.  * @version 1.10 2004-02-27
10. * @author Cay Horstmann
11. */
12. public class InnerClassTest
13. {
14.     public static void main(String[] args)
15.     {
16.         TalkingClock clock = new TalkingClock(1000, true);
17.         clock.start();
18.
19.         // keep program running until user selects "Ok"
20.         JOptionPane.showMessageDialog(null, "Quit program?");
21.         System.exit(0);
22.     }
23. }
24.
25. /**
26.  * A clock that prints the time in regular intervals.
27.  */
28. class TalkingClock
29. {
30.     /**
31.      * Constructs a talking clock
32.      * @param interval the interval between messages (in milliseconds)
33.      * @param beep true if the clock should beep
34.      */
35.     public TalkingClock(int interval, boolean beep)
36.     {
37.         this.interval = interval;
38.         this.beep = beep;
39.     }
40.
41.     /**
42.      * Starts the clock.
43.      */
44.     public void start()
45.     {
46.         ActionListener listener = new TimePrinter();
47.         Timer t = new Timer(interval, listener);
48.         t.start();
49.     }
}
```

**Listing 6-4** InnerClassTest.java (continued)

```

50.
51. private int interval;
52. private boolean beep;
53.
54. public class TimePrinter implements ActionListener
55. {
56.     public void actionPerformed(ActionEvent event)
57.     {
58.         Date now = new Date();
59.         System.out.println("At the tone, the time is " + now);
60.         if (beep) Toolkit.getDefaultToolkit().beep();
61.     }
62. }
63. }

```

**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, because `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.

**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 that has afflicted so many other languages?

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

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

```
javap -private ClassName
```



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

```
java ReflectionTest TalkingClock\TimePrinter
```

or

```
javap -private 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
{
    public TimePrinter(TalkingClock clock)
    {
        outer = clock;
    }
    . . .
    private TalkingClock outer;
}

```

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, because inner classes 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
{
    public TalkingClock(int, boolean);

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

    private int interval;
    private boolean beep;
}

```

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 inner class methods call that method. The statement

```
if (beep)
```

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

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

Is this a security risk? You bet it is. It is an easy matter for someone else to invoke the `access$0` method to read the private `beep` field. Of course, `access$0` is not a legal name for a Java method. However, hackers who are familiar with the structure of class files can easily produce a class file with virtual machine instructions to call that method, for example, by using a hex editor. Because 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 that are 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 compiler translates the constructor call in the `start` method of the `TalkingClock` class to

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

### 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.

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

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

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

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

Local classes have a 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.

### **Accessing final Variables from Outer Methods**

Local classes have another advantage over other inner classes. Not only can they access the fields of their outer classes, they can even access local variables! However, those local variables must be declared `final`. Here is a typical example. Let's move the `interval` and `beep` parameters from the `TalkingClock` constructor to the `start` method.

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

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

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

Maybe this should not be so surprising. The line

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

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

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

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

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

```

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

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

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

```

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

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

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



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

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

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

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

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

```

int counter = 0;
Date[] dates = new Date[100];
for (int i = 0; i < dates.length; i++)
    dates[i] = new Date()
    {
        public int compareTo(Date other)
        {
            counter++; // ERROR
        }
    }

```

```

        return super.compareTo(other);
    }
};
Arrays.sort(dates);
System.out.println(counter + " comparisons.");

```

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

```

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

```

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

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

### **Anonymous Inner Classes**

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

```

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

```

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

In general, the syntax is

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

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

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

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

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

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

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

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

Listing 6–5 contains the complete source code for the talking clock program with an anonymous inner class. If you compare this program with Listing 6–4, you will find that in this case the solution with the anonymous inner class is quite a bit shorter, and, hopefully, with a bit of practice, as easy to comprehend.

**Listing 6–5** AnonymousInnerClassTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.util.*;
4. import javax.swing.*;
5. import javax.swing.Timer;
6.
```

**Listing 6-5** AnonymousInnerClassTest.java (continued)

```
7. /**
8.  * This program demonstrates anonymous inner classes.
9.  * @version 1.10 2004-02-27
10. * @author Cay Horstmann
11. */
12. public class AnonymousInnerClassTest
13. {
14.     public static void main(String[] args)
15.     {
16.         TalkingClock clock = new TalkingClock();
17.         clock.start(1000, true);
18.
19.         // keep program running until user selects "Ok"
20.         JOptionPane.showMessageDialog(null, "Quit program?");
21.         System.exit(0);
22.     }
23. }
24.
25. /**
26.  * A clock that prints the time in regular intervals.
27.  */
28. class TalkingClock
29. {
30.     /**
31.      * Starts the clock.
32.      * @param interval the interval between messages (in milliseconds)
33.      * @param beep true if the clock should beep
34.      */
35.     public void start(int interval, final boolean beep)
36.     {
37.         ActionListener listener = new ActionListener()
38.         {
39.             public void actionPerformed(ActionEvent event)
40.             {
41.                 Date now = new Date();
42.                 System.out.println("At the tone, the time is " + now);
43.                 if (beep) Toolkit.getDefaultToolkit().beep();
44.             }
45.         };
46.         Timer t = new Timer(interval, listener);
47.         t.start();
48.     }
49. }
```

### ***Static Inner Classes***

Occasionally, you 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, then the array is traversed twice. It would be more efficient to traverse the array only once, computing both the minimum and the maximum simultaneously.

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

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

```
class Pair
{
    public Pair(double f, double s)
    {
        first = f;
        second = s;
    }
    public double getFirst() { return first; }
    public double getSecond() { return second; }

    private double first;
    private double second;
}
```

The `minmax` function 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 function 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, except that the other programmer 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: You use a static inner class whenever the inner class does not need to access an outer class object. Some programmers use the term *nested class* to describe static inner classes.



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

Listing 6–6 contains the complete source code of the `ArrayAlg` class and the nested `Pair` class.

#### Listing 6–6 StaticInnerClassTest.java

```
1. /**
2.  * This program demonstrates the use of static inner classes.
3.  * @version 1.01 2004-02-27
4.  * @author Cay Horstmann
5.  */
6. public class StaticInnerClassTest
7. {
8.     public static void main(String[] args)
9.     {
10.         double[] d = new double[20];
```

**Listing 6-6** StaticInnerClassTest.java (continued)

```
11.     for (int i = 0; i < d.length; i++)
12.         d[i] = 100 * Math.random();
13.     ArrayAlg.Pair p = ArrayAlg.minmax(d);
14.     System.out.println("min = " + p.getFirst());
15.     System.out.println("max = " + p.getSecond());
16. }
17. }
18.
19. class ArrayAlg
20. {
21.     /**
22.      * A pair of floating-point numbers
23.      */
24.     public static class Pair
25.     {
26.         /**
27.          * Constructs a pair from two floating-point numbers
28.          * @param f the first number
29.          * @param s the second number
30.          */
31.         public Pair(double f, double s)
32.         {
33.             first = f;
34.             second = s;
35.         }
36.
37.         /**
38.          * Returns the first number of the pair
39.          * @return the first number
40.          */
41.         public double getFirst()
42.         {
43.             return first;
44.         }
45.
46.         /**
47.          * Returns the second number of the pair
48.          * @return the second number
49.          */
50.         public double getSecond()
51.         {
52.             return second;
53.         }
54.
55.         private double first;
56.         private double second;
57.     }
58. }
```

**Listing 6-6** StaticInnerClassTest.java (continued)

```
59.  /**
60.   * Computes both the minimum and the maximum of an array
61.   * @param values an array of floating-point numbers
62.   * @return a pair whose first element is the minimum and whose second element
63.   * is the maximum
64.   */
65.  public static Pair minmax(double[] values)
66.  {
67.      double min = Double.MAX_VALUE;
68.      double max = Double.MIN_VALUE;
69.      for (double v : values)
70.      {
71.          if (min > v) min = v;
72.          if (max < v) max = v;
73.      }
74.      return new Pair(min, max);
75.  }
76. }
```

## Proxies

In the final section of this chapter, we discuss *proxies*, a feature that became available with Java SE 1.3. You 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 system programming applications, the flexibility that proxies offer can be very important.

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

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

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

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

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

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

To create a proxy object, you 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 for system classes, classes that are downloaded from the Internet, and so on, can be used. We 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; and
- 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
{
    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);
    }

    private Object target;
}
```

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 6–7, 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.newInstance(. . .); // 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, as of Java SE 5.0, 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) . . .
```

Because 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 6-7** ProxyTest.java

```

1. import java.lang.reflect.*;
2. import java.util.*;
3.
4. /**
5.  * This program demonstrates the use of proxies.
6.  * @version 1.00 2000-04-13
7.  * @author Cay Horstmann
8.  */
9. public class ProxyTest
10. {
11.     public static void main(String[] args)
12.     {
13.         Object[] elements = new Object[1000];
14.
15.         // fill elements with proxies for the integers 1 ... 1000
16.         for (int i = 0; i < elements.length; i++)
17.         {
18.             Integer value = i + 1;
19.             InvocationHandler handler = new TraceHandler(value);
20.             Object proxy = Proxy.newProxyInstance(null, new Class[] { Comparable.class }, handler);
21.             elements[i] = proxy;
22.         }
23.
24.         // construct a random integer
25.         Integer key = new Random().nextInt(elements.length) + 1;
26.
27.         // search for the key
28.         int result = Arrays.binarySearch(elements, key);
29.
30.         // print match if found
31.         if (result >= 0) System.out.println(elements[result]);
32.     }
33. }
34.
35. /**
36.  * An invocation handler that prints out the method name and parameters, then
37.  * invokes the original method
38.  */
39. class TraceHandler implements InvocationHandler
40. {

```

**Listing 6-7** ProxyTest.java (continued)

```
41.  /**
42.   * Constructs a TraceHandler
43.   * @param t the implicit parameter of the method call
44.   */
45.  public TraceHandler(Object t)
46.  {
47.      target = t;
48.  }
49.
50.  public Object invoke(Object proxy, Method m, Object[] args) throws Throwable
51.  {
52.      // print implicit argument
53.      System.out.print(target);
54.      // print method name
55.      System.out.print("." + m.getName() + "(");
56.      // print explicit arguments
57.      if (args != null)
58.      {
59.          for (int i = 0; i < args.length; i++)
60.          {
61.              System.out.print(args[i]);
62.              if (i < args.length - 1) System.out.print(", ");
63.          }
64.      }
65.      System.out.println(")");
66.
67.      // invoke actual method
68.      return m.invoke(target, args);
69.  }
70.
71.  private Object target;
72. }
```

### **Properties of Proxy Classes**

Now that you have seen proxy classes in action, we want to 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 that are 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 6-7, 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 Sun'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, then 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`, then the proxy class does not belong to any particular package. Otherwise, all non-`public` interfaces must belong to the same package, and then the proxy class also belongs to that package.

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

**API** `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.

**API** `java.lang.reflect.Proxy` 1.3

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

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



# *Chapter*

# 7

# GRAPHICS PROGRAMMING

- ▼ INTRODUCING SWING
- ▼ CREATING A FRAME
- ▼ POSITIONING A FRAME
- ▼ DISPLAYING INFORMATION IN A COMPONENT
- ▼ WORKING WITH 2D SHAPES
- ▼ USING COLOR
- ▼ USING SPECIAL FONTS FOR TEXT
- ▼ DISPLAYING IMAGES

To this point, you have seen only how to write programs that take input from the keyboard, fuss with it, and then display the results on a console screen. This is not what most users want now. Modern programs don't work this way and neither do web pages. This chapter starts you on the road to writing Java programs that use a graphical user interface (GUI). In particular, you learn how to write programs that size and locate windows on the screen, display text with multiple fonts in a window, display images, and so on. This gives you a useful, valuable repertoire of skills that you will put to good use in subsequent chapters as you write interesting programs.

The next two chapters show you how to process events, such as keystrokes and mouse clicks, and how to add interface elements, such as menus and buttons, to your applications. When you finish these three chapters, you will know the essentials for writing graphical applications. For more sophisticated graphics programming techniques, we refer you to Volume II.

If, on the other hand, you intend to use Java for server-side programming only and are not interested in writing GUI programming, you can safely skip these chapters.


### Introducing Swing

When Java 1.0 was introduced, it contained a class library, which Sun called the Abstract Window Toolkit (AWT), for basic GUI programming. The basic AWT library deals with user interface elements by delegating their creation and behavior to the native GUI toolkit on each target platform (Windows, Solaris, Macintosh, and so on). For example, if you used the original AWT to put a text box on a Java window, an underlying "peer" text box actually handled the text input. The resulting program could then, in theory, run on any of these platforms, with the "look and feel" of the target platform—hence Sun's trademarked slogan "Write Once, Run Anywhere."

The peer-based approach worked well for simple applications, but it soon became apparent that it was fiendishly difficult to write a high-quality portable graphics library that depended on native user interface elements. User interface elements such as menus, scrollbars, and text fields can have subtle differences in behavior on different platforms. It was hard, therefore, to give users a consistent and predictable experience with this approach. Moreover, some graphical environments (such as X11/Motif) do not have as rich a collection of user interface components as does Windows or the Macintosh. This in turn further limits a portable library based on peers to a "lowest common denominator" approach. As a result, GUI applications built with the AWT simply did not look as nice as native Windows or Macintosh applications, nor did they have the kind of functionality that users of those platforms had come to expect. More depressingly, there were *different* bugs in the AWT user interface library on the different platforms. Developers complained that they needed to test their applications on each platform, a practice derisively called "write once, debug everywhere."

In 1996, Netscape created a GUI library they called the IFC (Internet Foundation Classes) that used an entirely different approach. User interface elements, such as buttons, menus, and so on, were *painted* onto blank windows. The only functionality required from the underlying windowing system was a way to put up windows and to paint on the window. Thus, Netscape's IFC widgets looked and behaved the same no matter which platform the program ran on. Sun worked with Netscape to perfect this approach, creating a user interface library with the code name "Swing." Swing was available as an extension to Java 1.1 and became a part of the standard library in Java SE 1.2.

Since, as Duke Ellington said, “It Don’t Mean a Thing If It Ain’t Got That Swing,” Swing is now the official name for the non-peer-based GUI toolkit. Swing is part of the Java Foundation Classes (JFC). The full JFC is vast and contains far more than the Swing GUI toolkit. JFC features not only include the Swing components but also an accessibility API, a 2D API, and a drag-and-drop API.

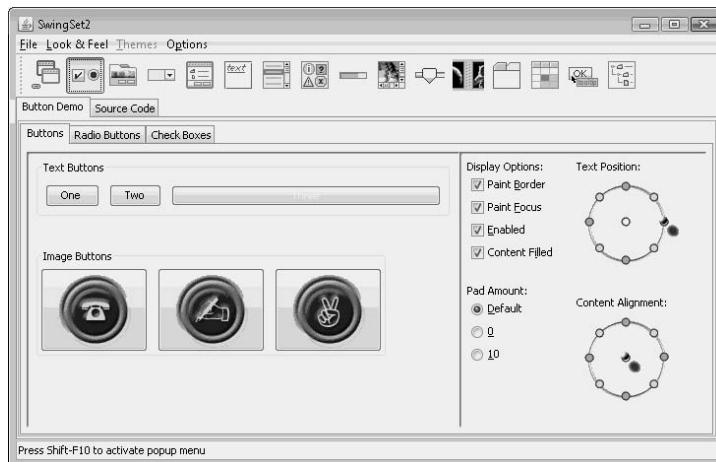
 NOTE: Swing is not a complete replacement for the AWT—it is built on top of the AWT architecture. Swing simply gives you more capable user interface components. You use the foundations of the AWT, in particular, event handling, whenever you write a Swing program. From now on, we say “Swing” when we mean the “painted” user interface classes, and we say “AWT” when we mean the underlying mechanisms of the windowing toolkit, such as event handling.

Of course, Swing-based user interface elements will be somewhat slower to appear on the user’s screen than the peer-based components used by the AWT. Our experience is that on any reasonably modern machine, the speed difference shouldn’t be a problem. On the other hand, the reasons to choose Swing are overwhelming:

- Swing has a rich and convenient set of user interface elements.
- Swing has few dependencies on the underlying platform; it is therefore less prone to platform-specific bugs.
- Swing gives a consistent user experience across platforms.

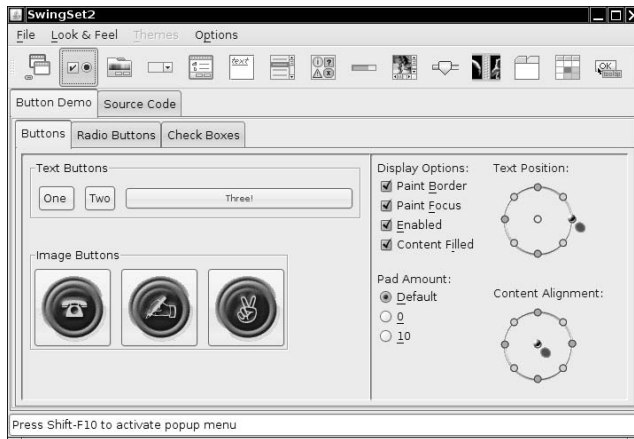
Still, the third plus is also a potential drawback: If the user interface elements look the same on all platforms, then they will look *different* from the native controls and thus users will be less familiar with them.

Swing solves this problem in a very elegant way. Programmers writing Swing programs can give the program a specific “look and feel.” For example, Figures 7–1 and 7–2 show the same program running with the Windows and the GTK look and feel.



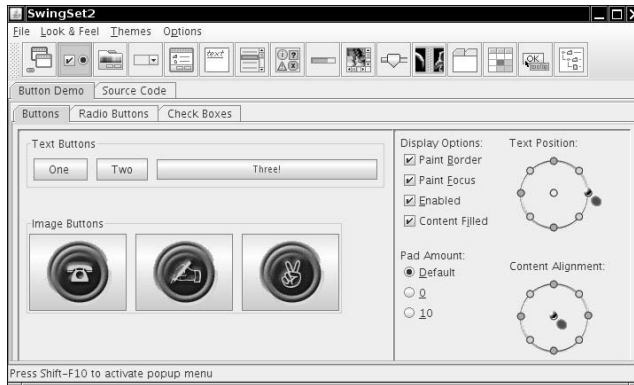
**Figure 7–1** The Windows look and feel of Swing

Furthermore, Sun developed a platform-independent look and feel that was called “Metal” until the marketing folks renamed it as the “Java look and feel.” However, most programmers continue to use the term “Metal,” and we will do the same in this book.



**Figure 7-2** The GTK look and feel of Swing

Some people criticized Metal as being stodgy, and the look was freshened up for the Java SE 5.0 release (see Figure 7-3). Now the Metal look supports multiple themes—minor variations in colors and fonts. The default theme is called “Ocean.”



**Figure 7-3** The Ocean theme of the Metal look and feel

In Java SE 6, Sun improved the support for the native look and feel for Windows and GTK. A Swing application will now pick up color scheme customizations and faithfully render the throbbing buttons and scrollbars that have become fashionable.

Some users prefer that their Java applications use the native look and feel of their platforms, others like Metal or a third-party look and feel. As you will see in Chapter 8, it is very easy to let your users choose their favorite look and feel



**NOTE:** Although we won't have space in this book to tell you how to do it, Java programmers can extend an existing look and feel or even design a totally new look and feel. This is a tedious process that involves specifying how each Swing component is painted. Some developers have done just that, especially when porting Java to nontraditional platforms such as kiosk terminals or handheld devices. See <http://www.javatoo.com> for a collection of interesting look-and-feel implementations.

Java SE 5.0 introduced a look and feel, called Synth, that makes this process easier. In Synth, you can define a new look and feel by providing image files and XML descriptors, without doing any programming.



**NOTE:** Most Java user interface programming is nowadays done in Swing, with one notable exception. The Eclipse integrated development environment uses a graphics toolkit called SWT that is similar to the AWT, mapping to native components on various platforms. You can find articles describing SWT at <http://www.eclipse.org/articles/>.

If you have programmed Microsoft Windows applications with Visual Basic or C#, you know about the ease of use that comes with the graphical layout tools and resource editors these products provide. These tools let you design the visual appearance of your application, and then they generate much (often all) of the GUI code for you. GUI builders are available for Java programming, but we feel that in order to use these tools effectively, you should know how to build a user interface manually. The remainder of this chapter tells you the basics about displaying a window and painting its contents.

### Creating a Frame

A top-level window (that is, a window that is not contained inside another window) is called a *frame* in Java. The AWT library has a class, called `Frame`, for this top level. The Swing version of this class is called `JFrame` and extends the `Frame` class. The `JFrame` is one of the few Swing components that is not painted on a canvas. Thus, the decorations (buttons, title bar, icons, and so on) are drawn by the user's windowing system, not by Swing.



**CAUTION:** Most Swing component classes start with a "J": `JButton`, `JFrame`, and so on. There are classes such as `Button` and `Frame`, but they are AWT components. If you accidentally omit a "J", your program may still compile and run, but the mixture of Swing and AWT components can lead to visual and behavioral inconsistencies.

In this section, we go over the most common methods for working with a Swing `JFrame`. Listing 7-1 lists a simple program that displays an empty frame on the screen, as illustrated in Figure 7-4.

**Figure 7-4** The simplest visible frame**Listing 7-1** SimpleFrameTest.java

```
1. import javax.swing.*;
2.
3. /**
4.  * @version 1.32 2007-06-12
5.  * @author Cay Horstmann
6.  */
7. public class SimpleFrameTest
8. {
9.     public static void main(String[] args)
10.    {
11.        SimpleFrame frame = new SimpleFrame();
12.        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
13.        frame.setVisible(true);
14.    }
15. }
16.
17. class SimpleFrame extends JFrame
18. {
19.     public SimpleFrame()
20.     {
21.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
22.     }
23.
24.     public static final int DEFAULT_WIDTH = 300;
25.     public static final int DEFAULT_HEIGHT = 200;
26. }
```

Let's work through this program, line by line.

The Swing classes are placed in the `javax.swing` package. The package name `javax` indicates a Java extension package, not a core package. For historical reasons, Swing is considered an extension. However, it is present in every Java SE implementation since version 1.2.

By default, a frame has a rather useless size of  $0 \times 0$  pixels. We define a subclass `SimpleFrame` whose constructor sets the size to  $300 \times 200$  pixels. This is the only difference between a `SimpleFrame` and a `JFrame`.

In the `main` method of the `SimpleFrameTest` class, we construct a `SimpleFrame` object and make it visible.

There are two technical issues that we need to address in every Swing program.

First, all Swing components must be configured from the *event dispatch thread*, the thread of control that passes events such as mouse clicks and keystrokes to the user interface components. The following code fragment is used to execute statements in the event dispatch thread:

```
EventQueue.invokeLater(new Runnable()
{
    public void run()
    {
        statements
    }
});
```

We discuss the details in Chapter 14. For now, you should simply consider it a magic incantation that is used to start a Swing program.



**NOTE:** You will see many Swing programs that do not initialize the user interface in the event dispatch thread. It used to be perfectly acceptable to carry out the initialization in the main thread. Sadly, as Swing components got more complex, the programmers at Sun were no longer able to guarantee the safety of that approach. The probability of an error is extremely low, but you would not want to be one of the unlucky few who encounter an intermittent problem. It is better to do the right thing, even if the code looks rather mysterious.

Next, we define what should happen when the user closes the application's frame. For this particular program, we want the program to exit. To select this behavior, we use the statement

```
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

In other programs with multiple frames, you would not want the program to exit just because the user closes one of the frames. By default, a frame is hidden when the user closes it, but the program does not terminate. (It might have been nice if the program terminated after the *last* frame became invisible, but that is not how Swing works.)

Simply constructing a frame does not automatically display it. Frames start their life invisible. That gives the programmer the chance to add components into the frame before showing it for the first time. To show the frame, the `main` method calls the `setVisible` method of the frame.



**NOTE:** Before Java SE 5.0, it was possible to use the `show` method that the `JFrame` class inherits from the superclass `Window`. The `Window` class has a superclass `Component` that also has a `show` method. The `Component.show` method was deprecated in Java SE 1.2. You are supposed to call `setVisible(true)` instead if you want to show a component. However, until Java SE 1.4, the `Window.show` method was *not* deprecated. In fact, it was quite useful, making the window visible *and* bringing it to the front. Sadly, that benefit was lost on the deprecation police, and Java SE 5.0 deprecated the `show` method for windows as well.

After scheduling the initialization statements, the `main` method exits. Note that exiting `main` does not terminate the program, just the main thread. The event dispatch thread keeps the program alive until it is terminated, either by closing the frame or by calling the `System.exit` method.

The running program is shown in Figure 7-4 on page 286—it is a truly boring top-level window. As you can see in the figure, the title bar and surrounding decorations, such as resize corners, are drawn by the operating system and not the Swing library. If you run the same program in Windows, GTK, or the Mac, the frame decorations are different. The Swing library draws everything inside the frame. In this program, it just fills the frame with a default background color.



NOTE: As of Java SE 1.4, you can turn off all frame decorations by calling `frame.setUndecorated(true)`.

### Positioning a Frame

The `JFrame` class itself has only a few methods for changing how frames look. Of course, through the magic of inheritance, most of the methods for working with the size and position of a frame come from the various superclasses of `JFrame`. Here are some of the most important methods:

- The `setLocation` and `setBounds` methods for setting the position of the frame
- The `setIconImage` method, which tells the windowing system which icon to display in the title bar, task switcher window, and so on
- The `setTitle` method for changing the text in the title bar
- The `setResizable` method, which takes a `boolean` to determine if a frame will be resizable by the user

Figure 7-5 illustrates the inheritance hierarchy for the `JFrame` class.



TIP: The API notes for this section give what we think are the most important methods for giving frames the proper look and feel. Some of these methods are defined in the `JFrame` class. Others come from the various superclasses of `JFrame`. At some point, you may need to search the API docs to see if there are methods for some special purpose. Unfortunately, that is a bit tedious to do with inherited methods. For example, the `ToFront` method is applicable to objects of type `JFrame`, but because it is simply inherited from the `Window` class, the `JFrame` documentation doesn't explain it. If you feel that there should be a method to do something and it isn't explained in the documentation for the class you are working with, try looking at the API documentation for the methods of the *superclasses* of that class. The top of each API page has hyperlinks to the superclasses, and inherited methods are listed below the method summary for the new and overridden methods.

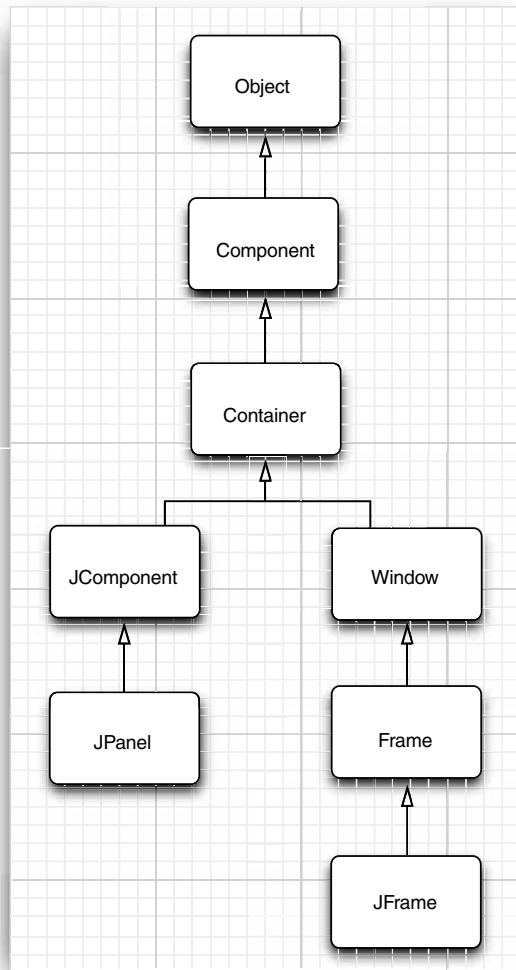
As the API notes indicate, the `Component` class (which is the ancestor of all GUI objects) and the `Window` class (which is the superclass of the `Frame` class) are where you need to look to find the methods to resize and reshape frames. For example, the `setLocation` method in the `Component` class is one way to reposition a component. If you make the call

```
setLocation(x, y)
```



the top-left corner is located  $x$  pixels across and  $y$  pixels down, where  $(0, 0)$  is the top-left corner of the screen. Similarly, the `setBounds` method in `Component` lets you resize and relocate a component (in particular, a `JFrame`) in one step, as

```
setBounds(x, y, width, height)
```



**Figure 7-5** Inheritance hierarchy for the frame and component classes in AWT and Swing

Alternatively, you can give the windowing system control on window placement. If you call

```
setLocationByPlatform(true);
```

before displaying the window, the windowing system picks the location (but not the size), typically with a slight offset from the last window.



NOTE: For a frame, the coordinates of the `setLocation` and `setBounds` are taken relative to the whole screen. As you will see in Chapter 9, for other components inside a container, the measurements are taken relative to the container.

### Frame Properties

Many methods of component classes come in getter/setter pairs, such as the following methods of the `Frame` class:

```
public String getTitle()
public void setTitle(String title)
```

Such a getter/setter pair is called a *property*. A property has a name and a type. The name is obtained by changing the first letter after the `get` or `set` to lowercase. For example, the `Frame` class has a property with name `title` and type `String`.

Conceptually, `title` is a property of the frame. When we set the property, we expect that the title changes on the user's screen. When we get the property, we expect that we get back the value that we set.

We do not know (or care) how the `Frame` class implements this property. Perhaps it simply uses its peer frame to store the title. Perhaps it has an instance field

```
private String title; // not required for property
```

If the class does have a matching instance field, we don't know (or care) how the getter and setter methods are implemented. Perhaps they just read and write the instance field. Perhaps they do more, such as notifying the windowing system whenever the title changes.

There is one exception to the `get/set` convention: For properties of type `boolean`, the getter starts with `is`. For example, the following two methods define the `locationByPlatform` property:

```
public boolean isLocationByPlatform()
public void setLocationByPlatform(boolean b)
```

We will look at properties in much greater detail in Chapter 8 of Volume II.



NOTE: Many programming languages, in particular, Visual Basic and C#, have built-in support for properties. It is possible that a future version of Java will also have a language construct for properties.

### Determining a Good Frame Size

Remember: if you don't explicitly size a frame, all frames will default to being 0 by 0 pixels. To keep our example programs simple, we resize the frames to a size that we hope works acceptably on most displays. However, in a professional application,

you should check the resolution of the user's screen and write code that resizes the frames accordingly: a window that looks nice on a laptop screen will look like a postage stamp on a high-resolution screen.

To find out the screen size, use the following steps. Call the static `getDefaultToolkit` method of the `Toolkit` class to get the `Toolkit` object. (The `Toolkit` class is a dumping ground for a variety of methods that interface with the native windowing system.) Then call the `getScreenSize` method, which returns the screen size as a `Dimension` object. A `Dimension` object simultaneously stores a width and a height, in public (!) instance variables `width` and `height`. Here is the code:

```
Toolkit kit = Toolkit.getDefaultToolkit();
Dimension screenSize = kit.getScreenSize();
int screenWidth = screenSize.width;
int screenHeight = screenSize.height;
```

We use 50% of these values for the frame size, and tell the windowing system to position the frame:

```
setSize(screenWidth / 2, screenHeight / 2);
setLocationByPlatform(true);
```

We also supply an icon. Because the representation of images is also system dependent, we again use the toolkit to load an image. Then, we set the image as the icon for the frame:

```
Image img = kit.getImage("icon.gif");
setIconImage(img);
```

Depending on your operating system, you can see the icon in various places. For example, in Windows, the icon is displayed in the top-left corner of the window, and you can see it in the list of active tasks when you press `ALT+TAB`.

Listing 7-2 is the complete program. When you run the program, pay attention to the "Core Java" icon.

Here are a few additional tips for dealing with frames:

- If your frame contains only standard components such as buttons and text fields, you can simply call the `pack` method to set the frame size. The frame will be set to the smallest size that contains all components. It is quite common to set the main frame of a program to the maximum size. As of Java SE 1.4, you can simply maximize a frame by calling  

```
frame.setExtendedState(Frame.MAXIMIZED_BOTH);
```
- It is also a good idea to remember how the user positions and sizes the frame of your application and restore those bounds when you start the application again. You will see in Chapter 10 how to use the Preferences API for this purpose.
- If you write an application that takes advantage of multiple display screens, use the `GraphicsEnvironment` and `GraphicsDevice` classes to find the dimensions of the display screens.
- The `GraphicsDevice` class also lets you execute your application in full-screen mode.

**Listing 7-2** SizedFrameTest.java

```
1. import java.awt.*;
2.
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.32 2007-04-14
7.  * @author Cay Horstmann
8.  */
9. public class SizedFrameTest
10. {
11.     public static void main(String[] args)
12.     {
13.         EventQueue.invokeLater(new Runnable()
14.         {
15.             public void run()
16.             {
17.                 SizedFrame frame = new SizedFrame();
18.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.                 frame.setVisible(true);
20.             }
21.         });
22.     }
23. }
24.
25. class SizedFrame extends JFrame
26. {
27.     public SizedFrame()
28.     {
29.         // get screen dimensions
30.
31.         Toolkit kit = Toolkit.getDefaultToolkit();
32.         Dimension screenSize = kit.getScreenSize();
33.         int screenHeight = screenSize.height;
34.         int screenWidth = screenSize.width;
35.
36.         // set frame width, height and let platform pick screen location
37.
38.         setSize(screenWidth / 2, screenHeight / 2);
39.         setLocationByPlatform(true);
40.
41.         // set frame icon and title
42.
43.         Image img = kit.getImage("icon.gif");
44.         setIconImage(img);
45.         setTitle("SizedFrame");
46.     }
47. }
```

---

**API** `java.awt.Component` 1.0

- `boolean isVisible()`
- `void setVisible(boolean b)`  
gets or sets the visible property. Components are initially visible, with the exception of top-level components such as `JFrame`.
- `void setSize(int width, int height)` 1.1  
resizes the component to the specified width and height.
- `void setLocation(int x, int y)` 1.1  
moves the component to a new location. The *x*- and *y*-coordinates use the coordinates of the container if the component is not a top-level component, or the coordinates of the screen if the component is top level (for example, a `JFrame`).
- `void setBounds(int x, int y, int width, int height)` 1.1  
moves and resizes this component.
- `Dimension getSize()` 1.1
- `void setSize(Dimension d)` 1.1  
gets or sets the size property of this component.

**API** `java.awt.Window` 1.0

- `void toFront()`  
shows this window on top of any other windows.
- `void toBack()`  
moves this window to the back of the stack of windows on the desktop and rearranges all other visible windows accordingly.
- `boolean isLocationByPlatform()` 5.0
- `void setLocationByPlatform(boolean b)` 5.0  
gets or sets the `locationByPlatform` property. When the property is set before this window is displayed, the platform picks a suitable location.

**API** `java.awt.Frame` 1.0

- `boolean isResizable()`
- `void setResizable(boolean b)`  
gets or sets the resizable property. When the property is set, the user can resize the frame.
- `String getTitle()`
- `void setTitle(String s)`  
gets or sets the title property that determines the text in the title bar for the frame.
- `Image getIconImage()`
- `void setIconImage(Image image)`  
gets or sets the `iconImage` property that determines the icon for the frame. The windowing system may display the icon as part of the frame decoration or in other locations.

- `boolean isUndecorated()` **1.4**
- `void setUndecorated(boolean b)` **1.4**  
gets or sets the undecorated property. When the property is set, the frame is displayed without decorations such as a title bar or close button. This method must be called before the frame is displayed.
- `int getExtendedState()` **1.4**
- `void setExtendedState(int state)` **1.4**  
gets or sets the extended window state. The state is one of  
`Frame.NORMAL`  
`Frame.ICONIFIED`  
`Frame.MAXIMIZED_HORIZ`  
`Frame.MAXIMIZED_VERT`  
`Frame.MAXIMIZED_BOTH`

**API** `java.awt.Toolkit` **1.0**

- `static Toolkit getDefaultToolkit()`  
returns the default toolkit.
- `Dimension getScreenSize()`  
gets the size of the user's screen.
- `Image getImage(String filename)`  
loads an image from the file with name `filename`.

**Displaying Information in a Component**

In this section, we show you how to display information inside a frame. For example, rather than displaying “Not a Hello, World program” in text mode in a console window as we did in Chapter 3, we display the message in a frame, as shown in Figure 7-6.



**Figure 7-6** A frame that displays information

You could draw the message string directly onto a frame, but that is not considered good programming practice. In Java, frames are really designed to be containers for components such as a menu bar and other user interface elements. You normally draw on another component which you add to the frame.

The structure of a `JFrame` is surprisingly complex. Look at Figure 7-7, which shows the makeup of a `JFrame`. As you can see, four panes are layered in a `JFrame`. The root pane, layered pane, and glass pane are of no interest to us; they are required to organize the menu bar and content pane and to implement the look and feel. The part that most con-

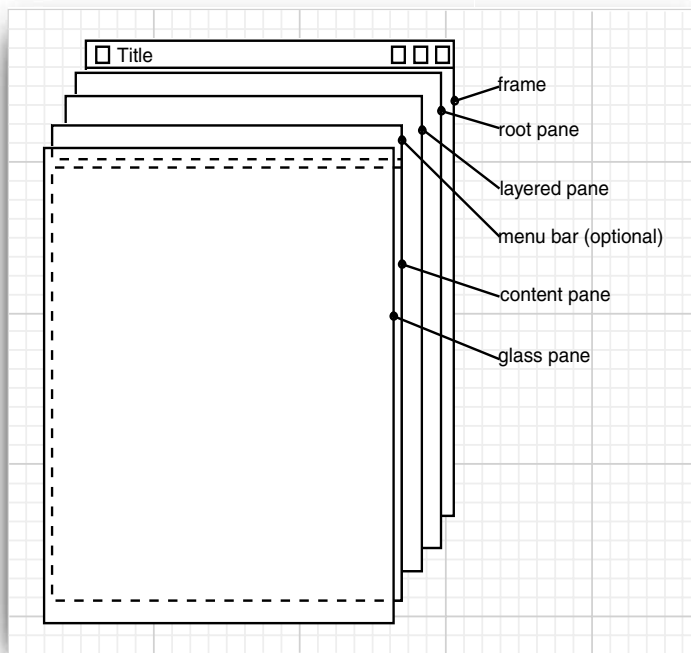
cerns Swing programmers is the *content pane*. When designing a frame, you add components into the content pane, using code such as the following:

```
Container contentPane = frame.getContentPane();
Component c = . . . ;
contentPane.add(c);
```

Up to Java SE 1.4, the `add` method of the `JFrame` class was defined to throw an exception with the message “Do not use `JFrame.add()`. Use `JFrame.getContentPane().add()` instead.” As of Java SE 5.0, the `JFrame.add` method has given up trying to reeducate programmers, and it simply calls `add` on the content pane.

Thus, as of Java SE 5.0, you can simply use the call

```
frame.add(c);
```



**Figure 7-7** Internal structure of a `JFrame`

In our case, we want to add a single component to the frame onto which we will draw our message. To draw on a component, you define a class that extends `JComponent` and override the `paintComponent` method in that class.

The `paintComponent` method takes one parameter of type `Graphics`. A `Graphics` object remembers a collection of settings for drawing images and text, such as the font you set or the current color. All drawing in Java must go through a `Graphics` object. It has methods that draw patterns, images, and text.



NOTE: The `Graphics` parameter is similar to a device context in Windows or a graphics context in X11 programming.

Here's how to make a component onto which you can draw:

```
class MyComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        code for drawing
    }
}
```

Each time a window needs to be redrawn, no matter what the reason, the event handler notifies the component. This causes the `paintComponent` methods of all components to be executed.

Never call the `paintComponent` method yourself. It is called automatically whenever a part of your application needs to be redrawn, and you should not interfere with this automatic process.

What sorts of actions trigger this automatic response? For example, painting occurs because the user increased the size of the window or minimized and then restored the window. If the user popped up another window and it covered an existing window and then made the overlaid window disappear, the application window that was covered is now corrupted and will need to be repainted. (The graphics system does not save the pixels underneath.) And, of course, when the window is displayed for the first time, it needs to process the code that specifies how and where it should draw the initial elements.



TIP: If you need to force repainting of the screen, call the `repaint` method instead of `paintComponent`. The `repaint` method will cause `paintComponent` to be called for all components, with a properly configured `Graphics` object.

As you saw in the code fragment above, the `paintComponent` method takes a single parameter of type `Graphics`. Measurement on a `Graphics` object for screen display is done in pixels. The (0, 0) coordinate denotes the top-left corner of the component on whose surface you are drawing.

Displaying text is considered a special kind of drawing. The `Graphics` class has a `drawString` method that has the following syntax:

```
g.drawString(text, x, y)
```

In our case, we want to draw the string "Not a Hello, World Program" in our original window, roughly one-quarter of the way across and halfway down. Although we don't yet know how to measure the size of the string, we'll start the string at coordinates (75, 100). This means the first character in the string will start at a position 75 pixels to the right and 100 pixels down. (Actually, it is the baseline for the text that is 100 pixels down—see page 313 for more on how text is measured.) Thus, our `paintComponent` method looks like this:



```

class NotHelloWorldComponent extends JComponent
{
    public void paintComponent(Graphics g)
    {
        g.drawString("Not a Hello, World program", MESSAGE_X, MESSAGE_Y);
    }

    public static final int MESSAGE_X = 75;
    public static final int MESSAGE_Y = 100;
}

```

Listing 7-3 shows the complete code.



NOTE: Instead of extending `JComponent`, some programmers prefer to extend the `JPanel` class. A `JPanel` is intended to be a *container* that can contain other components, but it is also possible to paint on it. There is just one difference. A panel is *opaque*, which means that it is responsible for painting all pixels within its bounds. The easiest way to achieve that is to paint the panel with the background color, by calling `super.paintComponent` in the `paintComponent` method of each panel subclass:

```

class NotHelloWorldPanel extends JPanel
{
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);

        . . . // code for drawing will go here
    }
}

```

#### Listing 7-3 NotHelloWorld.java

```

1. import javax.swing.*;
2. import java.awt.*;
3.
4. /**
5.  * @version 1.32 2007-06-12
6.  * @author Cay Horstmann
7.  */
8. public class NotHelloWorld
9. {
10.     public static void main(String[] args)
11.     {
12.         EventQueue.invokeLater(new Runnable()
13.         {
14.             public void run()
15.             {
16.                 NotHelloWorldFrame frame = new NotHelloWorldFrame();
17.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
18.                 frame.setVisible(true);
19.             }
16.         }

```

**Listing 7-3** NotHelloWorld.java (continued)

```
20.     });
21.   }
22. }
23.
24. /**
25.  * A frame that contains a message panel
26.  */
27. class NotHelloWorldFrame extends JFrame
28. {
29.     public NotHelloWorldFrame()
30.     {
31.         setTitle("NotHelloWorld");
32.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
33.
34.         // add panel to frame
35.
36.         NotHelloWorldPanel panel = new NotHelloWorldPanel();
37.         add(panel);
38.     }
39.
40.     public static final int DEFAULT_WIDTH = 300;
41.     public static final int DEFAULT_HEIGHT = 200;
42. }
43.
44. /**
45.  * A panel that displays a message.
46.  */
47. class NotHelloWorldPanel extends JPanel
48. {
49.     public void paintComponent(Graphics g)
50.     {
51.         g.drawString("Not a Hello, World program", MESSAGE_X, MESSAGE_Y);
52.     }
53.
54.     public static final int MESSAGE_X = 75;
55.     public static final int MESSAGE_Y = 100;
56. }
57.
```

**API** javax.swing.JFrame 1.2

- Container getContentPane()  
returns the content pane object for this JFrame.
- Component add(Component c)  
adds and returns the given component to the content pane of this frame. (Before Java SE 5.0, this method threw an exception.)

**API** `java.awt.Component` 1.0

- `void repaint()`  
causes a repaint of the component “as soon as possible.”
- `public void repaint(int x, int y, int width, int height)`  
causes a repaint of a part of the component “as soon as possible.”

**API** `javax.swing.JComponent` 1.2

- `void paintComponent(Graphics g)`  
overrides this method to describe how your component needs to be painted.

**Working with 2D Shapes**

Starting with Java 1.0, the `Graphics` class had methods to draw lines, rectangles, ellipses, and so on. But those drawing operations are very limited. For example, you cannot vary the line thickness and you cannot rotate the shapes.

Java SE 1.2 introduced the *Java 2D* library, which implements a powerful set of graphical operations. In this chapter, we only look at the basics of the Java 2D library—see the Advanced AWT chapter in Volume II for more information on the advanced features.

To draw shapes in the Java 2D library, you need to obtain an object of the `Graphics2D` class. This class is a subclass of the `Graphics` class. Ever since Java SE 2, methods such as `paintComponent` automatically receive an object of the `Graphics2D` class. Simply use a cast, as follows:


```
public void paintComponent(Graphics g)
{
    Graphics2D g2 = (Graphics2D) g;
    . . .
}
```

The Java 2D library organizes geometric shapes in an object-oriented fashion. In particular, there are classes to represent lines, rectangles, and ellipses:

```
Line2D
Rectangle2D
Ellipse2D
```

These classes all implement the `Shape` interface.

---

 **NOTE:** The Java 2D library supports more complex shapes—in particular, arcs, quadratic and cubic curves, and general paths. See Chapter 7 of Volume II for more information.

---

To draw a shape, you first create an object of a class that implements the `Shape` interface and then call the `draw` method of the `Graphics2D` class. For example:

```
Rectangle2D rect = . . . ;
g2.draw(rect);
```



**NOTE:** Before the Java 2D library appeared, programmers used methods of the `Graphics` class such as `drawRectangle` to draw shapes. Superficially, the old-style method calls look a bit simpler. However, by using the Java 2D library, you keep your options open—you can later enhance your drawings with some of the many tools that the Java 2D library supplies.

Using the Java 2D shape classes introduces some complexity. Unlike the 1.0 draw methods, which used integer pixel coordinates, the Java 2D shapes use floating-point coordinates. In many cases, that is a great convenience because it allows you to specify your shapes in coordinates that are meaningful to you (such as millimeters or inches) and then translate to pixels. The Java 2D library uses single-precision float quantities for many of its internal floating-point calculations. Single precision is sufficient—after all, the ultimate purpose of the geometric computations is to set pixels on the screen or printer. As long as any roundoff errors stay within one pixel, the visual outcome is not affected. Furthermore, float computations are faster on some platforms, and float values require half the storage of double values.

However, manipulating float values is sometimes inconvenient for the programmer because the Java programming language is adamant about requiring casts when converting double values into float values. For example, consider the following statement:

```
float f = 1.2; // Error
```

This statement does not compile because the constant `1.2` has type `double`, and the compiler is nervous about loss of precision. The remedy is to add an `F` suffix to the floating-point constant:

```
float f = 1.2F; // Ok
```

Now consider this statement:

```
Rectangle2D r = . . .
float f = r.getWidth(); // Error
```

This statement does not compile either, for the same reason. The `getWidth` method returns a `double`. This time, the remedy is to provide a cast:

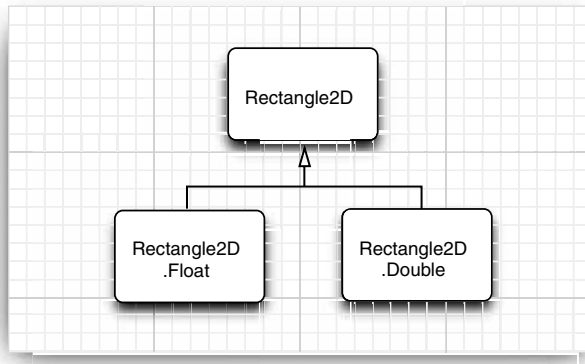
```
float f = (float) r.getWidth(); // Ok
```

Because the suffixes and casts are a bit of a pain, the designers of the 2D library decided to supply *two versions* of each shape class: one with float coordinates for frugal programmers, and one with double coordinates for the lazy ones. (In this book, we fall into the second camp and use double coordinates whenever we can.)

The library designers chose a curious, and initially confusing, method for packaging these choices. Consider the `Rectangle2D` class. This is an abstract class with two concrete subclasses, which are also static inner classes:

```
Rectangle2D.Float
Rectangle2D.Double
```

Figure 7–8 shows the inheritance diagram.



**Figure 7-8** 2D rectangle classes

It is best to try to ignore the fact that the two concrete classes are static inner classes—that is just a gimmick to avoid names such as `FloatRectangle2D` and `DoubleRectangle2D`. (For more information on static inner classes, see Chapter 6.)

When you construct a `Rectangle2D.Float` object, you supply the coordinates as float numbers. For a `Rectangle2D.Double` object, you supply them as double numbers.

```
Rectangle2D.Float floatRect = new Rectangle2D.Float(10.0F, 25.0F, 22.5F, 20.0F);
Rectangle2D.Double doubleRect = new Rectangle2D.Double(10.0, 25.0, 22.5, 20.0);
```

Actually, because both `Rectangle2D.Float` and `Rectangle2D.Double` extend the common `Rectangle2D` class and the methods in the subclasses simply override methods in the `Rectangle2D` superclass, there is no benefit in remembering the exact shape type. You can simply use `Rectangle2D` variables to hold the rectangle references.

```
Rectangle2D floatRect = new Rectangle2D.Float(10.0F, 25.0F, 22.5F, 20.0F);
Rectangle2D doubleRect = new Rectangle2D.Double(10.0, 25.0, 22.5, 20.0);
```

That is, you only need to use the pesky inner classes when you construct the shape objects.

The construction parameters denote the top-left corner, width, and height of the rectangle.



**NOTE:** Actually, the `Rectangle2D.Float` class has one additional method that is not inherited from `Rectangle2D`, namely, `setRect(float x, float y, float h, float w)`. You lose that method if you store the `Rectangle2D.Float` reference in a `Rectangle2D` variable. But it is not a big loss—the `Rectangle2D` class has a `setRect` method with double parameters.

The `Rectangle2D` methods use double parameters and return values. For example, the `getWidth` method returns a double value, even if the width is stored as a float in a `Rectangle2D.Float` object.



TIP: Simply use the `Double` shape classes to avoid dealing with `float` values altogether. However, if you are constructing thousands of shape objects, then you can consider using the `Float` classes to conserve memory.

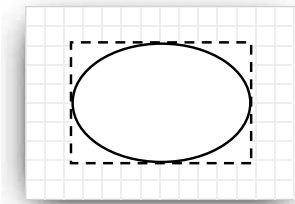
What we just discussed for the `Rectangle2D` classes holds for the other shape classes as well. Furthermore, there is a `Point2D` class with subclasses `Point2D.Float` and `Point2D.Double`. Here is how to make a point object.

```
Point2D p = new Point2D.Double(10, 20);
```



TIP: The `Point2D` class is very useful—it is more object oriented to work with `Point2D` objects than with separate `x`- and `y`- values. Many constructors and methods accept `Point2D` parameters. We suggest that you use `Point2D` objects when you can—they usually make geometric computations easier to understand.

The classes `Rectangle2D` and `Ellipse2D` both inherit from the common superclass `RectangularShape`. Admittedly, ellipses are not rectangular, but they have a *bounding rectangle* (see Figure 7–9).



**Figure 7–9** The bounding rectangle of an ellipse

The `RectangularShape` class defines over 20 methods that are common to these shapes, among them such useful methods as `getWidth`, `getHeight`, `getCenterX`, and `getCenterY` (but sadly, at the time of this writing, not a `getCenter` method that returns the center as a `Point2D` object).

Finally, a couple of legacy classes from Java 1.0 have been fitted into the shape class hierarchy. The `Rectangle` and `Point` classes, which store a rectangle and a point with integer coordinates, extend the `Rectangle2D` and `Point2D` classes.

Figure 7–10 shows the relationships between the shape classes. However, the `Double` and `Float` subclasses are omitted. Legacy classes are marked with a gray fill.

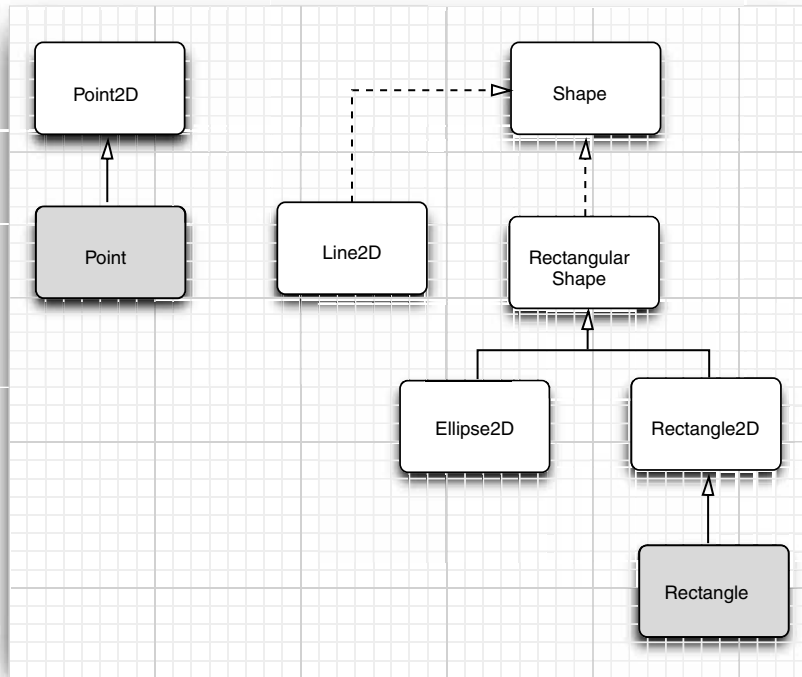
`Rectangle2D` and `Ellipse2D` objects are simple to construct. You need to specify

- The `x`- and `y`-coordinates of the top-left corner; and
- The width and height.

For ellipses, these refer to the bounding rectangle. For example,

```
Ellipse2D e = new Ellipse2D.Double(150, 200, 100, 50);
```

constructs an ellipse that is bounded by a rectangle with the top-left corner at (150, 200), width 100, and height 50.



**Figure 7-10 Relationships between the shape classes**

However, sometimes you don't have the top-left corner readily available. It is quite common to have two diagonal corner points of a rectangle, but perhaps they aren't the top-left and bottom-right corners. You can't simply construct a rectangle as

```
Rectangle2D rect = new Rectangle2D.Double(px, py, qx - px, qy - py); // Error
```

If  $p$  isn't the top-left corner, one or both of the coordinate differences will be negative and the rectangle will come out empty. In that case, first create a blank rectangle and use the `setFrameFromDiagonal` method, as follows:

```
Rectangle2D rect = new Rectangle2D.Double();
rect.setFrameFromDiagonal(px, py, qx, qy);
```

Or, even better, if you know the corner points as `Point2D` objects  $p$  and  $q$ , then

```
rect.setFrameFromDiagonal(p, q);
```

When constructing an ellipse, you usually know the center, width, and height, and not the corner points of the bounding rectangle (which don't even lie on the ellipse). The `setFrameFromCenter` method uses the center point, but it still requires one of the four corner points. Thus, you will usually end up constructing an ellipse as follows:

```
Ellipse2D ellipse = new Ellipse2D.Double(centerX - width / 2, centerY - height / 2, width, height);
```

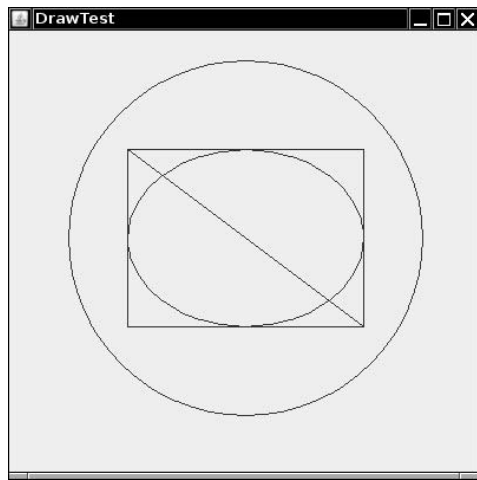
To construct a line, you supply the start and end points, either as `Point2D` objects or as pairs of numbers:

```
Line2D line = new Line2D.Double(start, end);
```

or

```
Line2D line = new Line2D.Double(startX, startY, endX, endY);
```

The program in Listing 7-4 draws a rectangle, the ellipse that is enclosed in the rectangle, a diagonal of the rectangle, and a circle that has the same center as the rectangle. Figure 7-11 shows the result.



**Figure 7-11** Drawing geometric shapes

**Listing 7-4** DrawTest.java

```
1. import java.awt.*;
2. import java.awt.geom.*;
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.32 2007-04-14
7.  * @author Cay Horstmann
8.  */
9. public class DrawTest
10. {
11.     public static void main(String[] args)
12.     {
13.         EventQueue.invokeLater(new Runnable()
14.         {
15.             public void run()
```



**Listing 7-4** DrawTest.java (continued)

```
16.         {
17.             DrawFrame frame = new DrawFrame();
18.             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.             frame.setVisible(true);
20.         }
21.     });
22. }
23. }
24.
25. /**
26.  * A frame that contains a panel with drawings
27.  */
28. class DrawFrame extends JFrame
29. {
30.     public DrawFrame()
31.     {
32.         setTitle("DrawTest");
33.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34.
35.         // add panel to frame
36.
37.         DrawComponent component = new DrawComponent();
38.         add(component);
39.     }
40.
41.     public static final int DEFAULT_WIDTH = 400;
42.     public static final int DEFAULT_HEIGHT = 400;
43. }
44.
45. /**
46.  * A component that displays rectangles and ellipses.
47.  */
48. class DrawComponent extends JComponent
49. {
50.     public void paintComponent(Graphics g)
51.     {
52.         Graphics2D g2 = (Graphics2D) g;
53.
54.         // draw a rectangle
55.
56.         double leftX = 100;
57.         double topY = 100;
58.         double width = 200;
59.         double height = 150;
60.
61.         Rectangle2D rect = new Rectangle2D.Double(leftX, topY, width, height);
62.         g2.draw(rect);
63.
64.         // draw the enclosed ellipse
```

**Listing 7-4** DrawTest.java (continued)

```
65.
66.     Ellipse2D ellipse = new Ellipse2D.Double();
67.     ellipse setFrame(rect);
68.     g2.draw(ellipse);
69.
70.     // draw a diagonal line
71.
72.     g2.draw(new Line2D.Double(leftX, topY, leftX + width, topY + height));
73.
74.     // draw a circle with the same center
75.
76.     double centerX = rect.getCenterX();
77.     double centerY = rect.getCenterY();
78.     double radius = 150;
79.
80.     Ellipse2D circle = new Ellipse2D.Double();
81.     circle.setFrameFromCenter(centerX, centerY, centerX + radius, centerY + radius);
82.     g2.draw(circle);
83. }
84. }
```

**API** java.awt.geom.RectangularShape 1.2

- double getCenterX()
  - double getCenterY()
  - double getMinX()
  - double getMinY()
  - double getMaxX()
  - double getMaxY()
- returns the center, minimum, or maximum *x*- or *y*-value of the enclosing rectangle.
- double getWidth()
  - double getHeight()
- returns the width or height of the enclosing rectangle.
- double getX()
  - double getY()
- returns the *x*- or *y*-coordinate of the top-left corner of the enclosing rectangle.

**API** java.awt.geom.Rectangle2D.Double 1.2

- Rectangle2D.Double(double *x*, double *y*, double *w*, double *h*)  
constructs a rectangle with the given top-left corner, width, and height.

**API** java.awt.geom.Rectangle2D.Float 1.2

- Rectangle2D.Float(float *x*, float *y*, float *w*, float *h*)  
constructs a rectangle with the given top-left corner, width, and height.

**API** `java.awt.geom.Ellipse2D.Double` 1.2

- `Ellipse2D.Double(double x, double y, double w, double h)` constructs an ellipse whose bounding rectangle has the given top-left corner, width, and height.

**API** `java.awt.geom.Point2D.Double` 1.2

- `Point2D.Double(double x, double y)` constructs a point with the given coordinates.

**API** `java.awt.geom.Line2D.Double` 1.2

- `Line2D.Double(Point2D start, Point2D end)`
  - `Line2D.Double(double startX, double startY, double endX, double endY)`
- constructs a line with the given start and end points.

## Using Color

The `setPaint` method of the `Graphics2D` class lets you select a color that is used for all subsequent drawing operations on the graphics context. For example:

```
g2.setPaint(Color.RED);
g2.drawString("Warning!", 100, 100);
```

You can fill the interiors of closed shapes (such as rectangles or ellipses) with a color.

Simply call `fill` instead of `draw`:

```
Rectangle2D rect = . . .;
g2.setPaint(Color.RED);
g2.fill(rect); // fills rect with red color
```

To draw in multiple colors, you select a color, draw or fill, then select another color, and draw or fill again.

You define colors with the `Color` class. The `java.awt.Color` class offers predefined constants for the following 13 standard colors:

```
BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN, LIGHT_GRAY, MAGENTA, ORANGE, PINK, RED, WHITE, YELLOW
```



**NOTE:** Before Java SE 1.4, color constant names were lowercase, such as `Color.red`. This is odd because the standard coding convention is to write constants in uppercase. You can now write the standard color names in uppercase or, for backward compatibility, in lowercase.

You can specify a custom color by creating a `Color` object by its red, green, and blue components. Using a scale of 0–255 (that is, one byte) for the redness, blueness, and greenness, call the `Color` constructor like this:

```
Color(int redness, int greenness, int blueness)
```

Here is an example of setting a custom color:

```
g2.setPaint(new Color(0, 128, 128)); // a dull blue-green
g2.drawString("Welcome!", 75, 125);
```



**NOTE:** In addition to solid colors, you can select more complex “paint” settings, such as varying hues or images. See the Advanced AWT chapter in Volume II for more details. If you use a `Graphics` object instead of a `Graphics2D` object, you need to use the `setColor` method to set colors.

To set the *background color*, you use the `setBackground` method of the `Component` class, an ancestor of `JComponent`.

```
MyComponent p = new MyComponent();
p.setBackground(Color.PINK);
```

There is also a `setForeground` method. It specifies the default color that is used for drawing on the component.



**TIP:** The `brighter()` and `darker()` methods of the `Color` class produce, as their names suggest, either brighter or darker versions of the current color. Using the `brighter` method is also a good way to highlight an item. Actually, `brighter()` is just a little bit brighter. To make a color really stand out, apply it three times: `c.brighter().brighter().brighter()`.

Java gives you predefined names for many more colors in its `SystemColor` class. The constants in this class encapsulate the colors used for various elements of the user’s system. For example,

```
p.setBackground(SystemColor.window)
```

sets the background color of the component to the default used by all windows on the user’s desktop. (The background is filled in whenever the window is repainted.) Using the colors in the `SystemColor` class is particularly useful when you want to draw user interface elements so that the colors match those already found on the user’s desktop. Table 7-1 lists the system color names and their meanings.

**Table 7-1 System Colors**

Name	Purpose
<code>desktop</code>	Background color of desktop
<code>activeCaption</code>	Background color for captions
<code>activeCaptionText</code>	Text color for captions
<code>activeCaptionBorder</code>	Border color for caption text
<code>inactiveCaption</code>	Background color for inactive captions
<code>inactiveCaptionText</code>	Text color for inactive captions
<code>inactiveCaptionBorder</code>	Border color for inactive captions
<code>window</code>	Background for windows
<code>windowBorder</code>	Color of window border frame

**Table 7-1 System Colors (continued)**

Name	Purpose
windowText	Text color inside windows
menu	Background for menus
menuText	Text color for menus
text	Background color for text
textText	Text color for text
textInactiveText	Text color for inactive controls
textHighlight	Background color for highlighted text
textHighlightText	Text color for highlighted text
control	Background color for controls
controlText	Text color for controls
controlLtHighlight	Light highlight color for controls
controlHighlight	Highlight color for controls
controlShadow	Shadow color for controls
controlDkShadow	Dark shadow color for controls
scrollbar	Background color for scrollbars
info	Background color for spot-help text
infoText	Text color for spot-help text

**API** `java.awt.Color` 1.0

- `Color(int r, int g, int b)` creates a color object.

*Parameters:*

<code>r</code>	The red value (0–255)
<code>g</code>	The green value (0–255)
<code>b</code>	The blue value (0–255)

**API** `java.awt.Graphics` 1.0

- `Color getColor()`
- `void setColor(Color c)` gets or sets the current color. All subsequent graphics operations will use the new color.

*Parameters:*

<code>c</code>	The new color
----------------	---------------

**API** `java.awt.Graphics2D` 1.2

- `Paint getPaint()`
- `void setPaint(Paint p)`  
gets or sets the paint property of this graphics context. The `Color` class implements the `Paint` interface. Therefore, you can use this method to set the paint attribute to a solid color.
- `void fill(Shape s)`  
fills the shape with the current paint.

**API** `java.awt.Component` 1.0

- `Color getBackground()`
- `void setBackground(Color c)`  
gets or sets the background color.  
*Parameters:*     `c`                   The new background color
- `Color getForeground()`
- `void setForeground(Color c)`  
gets or sets the foreground color.  
*Parameters:*     `c`                   The new foreground color

**Using Special Fonts for Text**

The “Not a Hello, World” program at the beginning of this chapter displayed a string in the default font. Often, you want to show text in a different font. You specify a font by its *font face name*. A font face name is composed of a *font family name*, such as “Helvetica,” and an optional suffix such as “Bold.” For example, the font faces “Helvetica” and “Helvetica Bold” are both considered to be part of the family named “Helvetica.”

To find out which fonts are available on a particular computer, call the `getAvailableFontFamilyNames` method of the `GraphicsEnvironment` class. The method returns an array of strings that contains the names of all available fonts. To obtain an instance of the `GraphicsEnvironment` class that describes the graphics environment of the user’s system, use the static `getLocalGraphicsEnvironment` method. Thus, the following program prints the names of all fonts on your system:

```
import java.awt.*;

public class ListFonts
{
    public static void main(String[] args)
    {
        String[] fontNames = GraphicsEnvironment
            .getLocalGraphicsEnvironment()
            .getAvailableFontFamilyNames();
        for (String fontName : fontNames)
            System.out.println(fontName);
    }
}
```

On one system, the list starts out like this:

```
Abadi MT Condensed Light
Arial
Arial Black
Arial Narrow
Arioso
Baskerville
Binner Gothic
. . .
```

and goes on for another 70 fonts or so.

Font face names can be trademarked, and font designs can be copyrighted in some jurisdictions. Thus, the distribution of fonts often involves royalty payments to a font foundry. Of course, just as there are inexpensive imitations of famous perfumes, there are lookalikes for name-brand fonts. For example, the Helvetica imitation that is shipped with Windows is called Arial.

To establish a common baseline, the AWT defines five *logical* font names:

```
SansSerif
Serif
Monospaced
Dialog
DialogInput
```

These names are always mapped to fonts that actually exist on the client machine. For example, on a Windows system, SansSerif is mapped to Arial.

In addition, the Sun JDK always includes three font families named “Lucida Sans,” “Lucida Bright,” and “Lucida Sans Typewriter.”

To draw characters in a font, you must first create an object of the class `Font`. You specify the font face name, the font style, and the point size. Here is an example of how you construct a `Font` object:

```
Font sansbold14 = new Font("SansSerif", Font.BOLD, 14);
```

The third argument is the point size. Points are commonly used in typography to indicate the size of a font. There are 72 points per inch.

You can use a logical font name in the place of a font face name in the `Font` constructor. You specify the style (plain, **bold**, *italic*, or **bold italic**) by setting the second `Font` constructor argument to one of the following values:

```
Font.PLAIN
Font.BOLD
Font.ITALIC
Font.BOLD + Font.ITALIC
```



**NOTE:** The mapping from logical to physical font names is defined in the `fontconfig.properties` file in the `jre/lib` subdirectory of the Java installation. See <http://java.sun.com/javase/6/docs/technotes/guides/intl/fontconfig.html> for information on this file.

You can read font files in TrueType or PostScript Type 1 formats. You need an input stream for the font—typically from a file or URL. (See Chapter 1 of Volume II for more information on streams.) Then call the static `Font.createFont` method:

```
URL url = new URL("http://www.fonts.com/Wingbats.ttf");
InputStream in = url.openStream();
Font f1 = Font.createFont(Font.TRUETYPE_FONT, in);
```

The font is plain with a font size of 1 point. Use the `deriveFont` method to get a font of the desired size:

```
Font f = f1.deriveFont(14.0F);
```



**CAUTION:** There are two overloaded versions of the `deriveFont` method. One of them (with a float parameter) sets the font size, the other (with an int parameter) sets the font style. Thus, `f1.deriveFont(14)` sets the style and not the size! (The result is an italic font because it happens that the binary representation of 14 sets the ITALIC bit but not the BOLD bit.)

The Java fonts contain the usual ASCII characters as well as symbols. For example, if you print the character `'\u2297'` in the Dialog font, then you get a  $\otimes$  character. Only those symbols that are defined in the Unicode character set are available.

Here's the code that displays the string "Hello, World!" in the standard sans serif font on your system, using 14-point bold type:

```
Font sansbold14 = new Font("SansSerif", Font.BOLD, 14);
g2.setFont(sansbold14);
String message = "Hello, World!";
g2.drawString(message, 75, 100);
```

Next, let's *center* the string in its component rather than drawing it at an arbitrary position. We need to know the width and height of the string in pixels. These dimensions depend on three factors:

- The font used (in our case, sans serif, bold, 14 point);
- The string (in our case, "Hello, World!"); and
- The device on which the font is drawn (in our case, the user's screen).

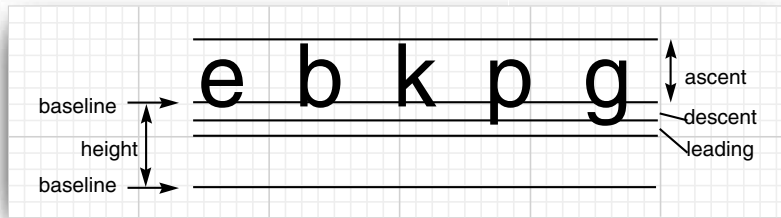
To obtain an object that represents the font characteristics of the screen device, you call the `getFontRenderContext` method of the `Graphics2D` class. It returns an object of the `FontRenderContext` class. You simply pass that object to the `getStringBounds` method of the `Font` class:

```
FontRenderContext context = g2.getFontRenderContext();
Rectangle2D bounds = f.getStringBounds(message, context);
```

The `getStringBounds` method returns a rectangle that encloses the string.

To interpret the dimensions of that rectangle, you should know some basic typesetting terms (see Figure 7-12). The *baseline* is the imaginary line where, for example, the bottom of a character like "e" rests. The *ascent* is the distance from the baseline to the top of an *ascender*, which is the upper part of a letter like "b" or "k," or an uppercase character. The *descent* is the distance from the baseline to a *descender*, which is the lower portion of a letter like "p" or "g."





**Figure 7-12** Typesetting terms illustrated

*Leading* is the space between the descent of one line and the ascent of the next line. (The term has its origin from the strips of lead that typesetters used to separate lines.) The *height* of a font is the distance between successive baselines, which is the same as descent + leading + ascent.

The width of the rectangle that the `getStringBounds` method returns is the horizontal extent of the string. The height of the rectangle is the sum of ascent, descent, and leading. The rectangle has its origin at the baseline of the string. The top *y*-coordinate of the rectangle is negative. Thus, you can obtain string width, height, and ascent as follows:

```
double stringWidth = bounds.getWidth();
double stringHeight = bounds.getHeight();
double ascent = -bounds.getY();
```

If you need to know the descent or leading, you need to use the `getLineMetrics` method of the `Font` class. That method returns an object of the `LineMetrics` class, which has methods to obtain the descent and leading:

```
LineMetrics metrics = f.getLineMetrics(message, context);
float descent = metrics.getDescent();
float leading = metrics.getLeading();
```

The following code uses all this information to center a string in its surrounding component:

```
FontRenderContext context = g2.getFontRenderContext();
Rectangle2D bounds = f.getStringBounds(message, context);

// (x,y) = top left corner of text
double x = (getWidth() - bounds.getWidth()) / 2;
double y = (getHeight() - bounds.getHeight()) / 2;

// add ascent to y to reach the baseline
double ascent = -bounds.getY();
double baseY = y + ascent;
g2.drawString(message, (int) x, (int) baseY);
```

To understand the centering, consider that `getWidth()` returns the width of the component. A portion of that width, namely, `bounds.getWidth()`, is occupied by the message string. The remainder should be equally distributed on both sides. Therefore, the blank space on each side is half the difference. The same reasoning applies to the height.



NOTE: When you need to compute layout dimensions outside the `paintComponent` method, you can't obtain the font render context from the `Graphics2D` object. Instead, call the `getFontMetrics` method of the `JComponent` class and then call `getFontRenderContext`.

```
FontRenderContext context = getFontMetrics(f).getFontRenderContext();
```

To show that the positioning is accurate, the sample program also draws the baseline and the bounding rectangle. Figure 7-13 shows the screen display; Listing 7-5 is the program listing.



Figure 7-13 Drawing the baseline and string bounds

**Listing 7-5** FontTest.java

```

1. import java.awt.*;
2. import java.awt.font.*;
3. import java.awt.geom.*;
4. import javax.swing.*;
5.
6. /**
7.  * @version 1.33 2007-04-14
8.  * @author Cay Horstmann
9.  */
10. public class FontTest
11. {
12.     public static void main(String[] args)
13.     {
14.         EventQueue.invokeLater(new Runnable()
15.         {
16.             public void run()
17.             {
18.                 FontFrame frame = new FontFrame();
19.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20.                 frame.setVisible(true);
21.             }
22.         });
23.     }
24. }
25.

```

**Listing 7-5** FontTest.java (continued)

```
26. /**
27.  * A frame with a text message component
28.  */
29. class FontFrame extends JFrame
30. {
31.     public FontFrame()
32.     {
33.         setTitle("FontTest");
34.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
35.
36.         // add component to frame
37.
38.         FontComponent component = new FontComponent();
39.         add(component);
40.     }
41.
42.     public static final int DEFAULT_WIDTH = 300;
43.     public static final int DEFAULT_HEIGHT = 200;
44. }
45.
46. /**
47.  * A component that shows a centered message in a box.
48.  */
49. class FontComponent extends JComponent
50. {
51.     public void paintComponent(Graphics g)
52.     {
53.         Graphics2D g2 = (Graphics2D) g;
54.
55.         String message = "Hello, World!";
56.
57.         Font f = new Font("Serif", Font.BOLD, 36);
58.         g2.setFont(f);
59.
60.         // measure the size of the message
61.
62.         FontRenderContext context = g2.getFontRenderContext();
63.         Rectangle2D bounds = f.getStringBounds(message, context);
64.
65.         // set (x,y) = top-left corner of text
66.
67.         double x = (getWidth() - bounds.getWidth()) / 2;
68.         double y = (getHeight() - bounds.getHeight()) / 2;
69.
70.         // add ascent to y to reach the baseline
71.
72.         double ascent = -bounds.getY();
73.         double baseY = y + ascent;
74.     }
75. }
```

**Listing 7-5** FontTest.java (continued)

```

75.    // draw the message
76.
77.    g2.drawString(message, (int) x, (int) baseY);
78.
79.    g2.setPaint(Color.LIGHT_GRAY);
80.
81.    // draw the baseline
82.
83.    g2.draw(new Line2D.Double(x, baseY, x + bounds.getWidth(), baseY));
84.
85.    // draw the enclosing rectangle
86.
87.    Rectangle2D rect = new Rectangle2D.Double(x, y, bounds.getWidth(), bounds.getHeight());
88.    g2.draw(rect);
89.  }
90. }
```

**API** java.awt.Font 1.0

- Font(String name, int style, int size)  
creates a new font object.
 

<i>Parameters:</i>	name	The font name. This is either a font face name (such as “Helvetica Bold”) or a logical font name (such as “Serif”, “SansSerif”).
	style	The style (Font.PLAIN, Font.BOLD, Font.ITALIC, or Font.BOLD + Font.ITALIC)
	size	The point size (for example, 12)
- String getFontName()  
gets the font face name (such as “Helvetica Bold”).
- String getFamily()  
gets the font family name (such as “Helvetica”).
- String getName()  
gets the logical name (such as “SansSerif”) if the font was created with a logical font name; otherwise, gets the font face name.
- Rectangle2D getStringBounds(String s, FontRenderContext context) **1.2**  
returns a rectangle that encloses the string. The origin of the rectangle falls on the baseline. The top *y*-coordinate of the rectangle equals the negative of the ascent. The height of the rectangle equals the sum of ascent, descent, and leading. The width equals the string width.
- LineMetrics getLineMetrics(String s, FontRenderContext context) **1.2**  
returns a line metrics object to determine the extent of the string.

- Font `deriveFont(int style)` **1.2**
- Font `deriveFont(float size)` **1.2**
- Font `deriveFont(int style, float size)` **1.2**  
returns a new font that equals this font, except that it has the given size and style.

**API** `java.awt.font.LineMetrics` **1.2**

- float `getAscent()`  
gets the font ascent—the distance from the baseline to the tops of uppercase characters.
- float `getDescent()`  
gets the font descent—the distance from the baseline to the bottoms of descenders.
- float `getLeading()`  
gets the font leading—the space between the bottom of one line of text and the top of the next line.
- float `getHeight()`  
gets the total height of the font—the distance between the two baselines of text (descent + leading + ascent).

**API** `java.awt.Graphics` **1.0**

- Font `getFont()`
- void `setFont(Font font)`  
gets or sets the current font. That font will be used for subsequent text-drawing operations.  
*Parameters:*     font             A font
- void `drawString(String str, int x, int y)`  
draws a string in the current font and color.  
*Parameters:*     str             The string to be drawn  
                  x             The *x*-coordinate of the start of the string  
                  y             The *y*-coordinate of the baseline of the string

**API** `java.awt.Graphics2D` **1.2**

- `FontRenderContext getFontRenderContext()`  
gets a font render context that specifies font characteristics in this graphics context.
- void `drawString(String str, float x, float y)`  
draws a string in the current font and color.  
*Parameters:*     str             The string to be drawn  
                  x             The *x*-coordinate of the start of the string  
                  y             The *y*-coordinate of the baseline of the string

**API** javax.swing.JComponent 1.2

- FontMetrics getFontMetrics(Font f) 5.0  
gets the font metrics for the given font. The FontMetrics class is a precursor to the LineMetrics class.

**API** java.awt.FontMetrics 1.0

- FontRenderContext getFontRenderContext() 1.2  
gets a font render context for the font.

**Displaying Images**

You have already seen how to build up simple drawings by painting lines and shapes. Complex images, such as photographs, are usually generated externally, for example, with a scanner or special image-manipulation software. (As you will see in Volume II, it is also possible to produce an image, pixel by pixel, and store the result in an array. This procedure is common for fractal images, for example.)

Once images are stored in local files or someplace on the Internet, you can read them into a Java application and display them on Graphics objects. As of Java SE 1.4, reading an image is very simple. If the image is stored in a local file, call

```
String filename = "...";
Image image = ImageIO.read(new File(filename));
```

Otherwise, you can supply a URL:

```
String urlName = "...";
Image image = ImageIO.read(new URL(urlName));
```

The read method throws an IOException if the image is not available. We discuss the general topic of exception handling in Chapter 11. For now, our sample program just catches that exception and prints a stack trace if it occurs.

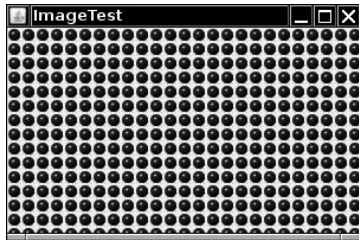
Now the variable `image` contains a reference to an object that encapsulates the image data. You can display the image with the `drawImage` method of the `Graphics` class.

```
public void paintComponent(Graphics g)
{
    . . .
    g.drawImage(image, x, y, null);
}
```

Listing 7-6 takes this a little bit further and *tiles* the window with the graphics image. The result looks like the screen shown in Figure 7-14. We do the tiling in the `paintComponent` method. We first draw one copy of the image in the top-left corner and then use the `copyArea` call to copy it into the entire window:

```
for (int i = 0; i * imageWidth <= getWidth(); i++)
    for (int j = 0; j * imageHeight <= getHeight(); j++)
        if (i + j > 0)
            g.copyArea(0, 0, imageWidth, imageHeight, i * imageWidth, j * imageHeight);
```

Listing 7-6 shows the full source code of the image display program.

**Figure 7-14** Window with tiled graphics image**Listing 7-6** ImageTest.java

```
1. import java.awt.*;
2. import java.io.*;
3. import javax.imageio.*;
4. import javax.swing.*;
5.
6. /**
7.  * @version 1.33 2007-04-14
8.  * @author Cay Horstmann
9.  */
10. public class ImageTest
11. {
12.     public static void main(String[] args)
13.     {
14.         EventQueue.invokeLater(new Runnable()
15.         {
16.             public void run()
17.             {
18.                 ImageFrame frame = new ImageFrame();
19.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20.                 frame.setVisible(true);
21.             }
22.         });
23.     }
24. }
25.
26. /**
27.  * A frame with an image component
28.  */
29. class ImageFrame extends JFrame
30. {
31.     public ImageFrame()
32.     {
33.         setTitle("ImageTest");
34.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
35.     }
36. }
```

**Listing 7-6** ImageTest.java (continued)

```
36.     // add component to frame
37.
38.     ImageComponent component = new ImageComponent();
39.     add(component);
40. }
41.
42. public static final int DEFAULT_WIDTH = 300;
43. public static final int DEFAULT_HEIGHT = 200;
44. }
45.
46. /**
47.  * A component that displays a tiled image
48.  */
49. class ImageComponent extends JComponent
50. {
51.     public ImageComponent()
52.     {
53.         // acquire the image
54.         try
55.         {
56.             image = ImageIO.read(new File("blue-ball.gif"));
57.         }
58.         catch (IOException e)
59.         {
60.             e.printStackTrace();
61.         }
62.     }
63.
64.     public void paintComponent(Graphics g)
65.     {
66.         if (image == null) return;
67.
68.         int imageWidth = image.getWidth(this);
69.         int imageHeight = image.getHeight(this);
70.
71.         // draw the image in the top-left corner
72.
73.         g.drawImage(image, 0, 0, null);
74.         // tile the image across the component
75.
76.         for (int i = 0; i * imageWidth <= getWidth(); i++)
77.             for (int j = 0; j * imageHeight <= getHeight(); j++)
78.                 if (i + j > 0) g.copyArea(0, 0, imageWidth, imageHeight, i * imageWidth, j
79.                     * imageHeight);
80.     }
81.
82.     private Image image;
83. }
```



**API** javax.imageio.ImageIO 1.4

- static BufferedImage read(File f)
- static BufferedImage read(URL u)  
reads an image from the given file or URL.

**API** java.awt.Graphics 1.0

- boolean drawImage(Image img, int x, int y, ImageObserver observer)  
draws an unscaled image. Note: This call may return before the image is drawn.  
*Parameters:*

img	The image to be drawn
x	The <i>x</i> -coordinate of the top-left corner
y	The <i>y</i> -coordinate of the top-left corner
observer	The object to notify of the progress of the rendering process (may be null)
- boolean drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)  
draws a scaled image. The system scales the image to fit into a region with the given width and height. Note: This call may return before the image is drawn.  
*Parameters:*

img	The image to be drawn
x	The <i>x</i> -coordinate of the top-left corner
y	The <i>y</i> -coordinate of the top-left corner
width	The desired width of image
height	The desired height of image
observer	The object to notify of the progress of the rendering process (may be null)
- void copyArea(int x, int y, int width, int height, int dx, int dy)  
copies an area of the screen.  
*Parameters:*

x	The <i>x</i> -coordinate of the top-left corner of the source area
y	The <i>y</i> -coordinate of the top-left corner of the source area
width	The width of the source area
height	The height of the source area
dx	The horizontal distance from the source area to the target area
dy	The vertical distance from the source area to the target area

This concludes our introduction to Java graphics programming. For more advanced techniques, you can turn to the discussion about 2D graphics and image manipulation in Volume II. In the next chapter, you will learn how your programs react to user input.

---

*Chapter*

8

# EVENT HANDLING

- ▼ BASICS OF EVENT HANDLING
- ▼ ACTIONS
- ▼ MOUSE EVENTS
- ▼ THE AWT EVENT HIERARCHY

**E**vent handling is of fundamental importance to programs with a graphical user interface. To implement user interfaces, you must master the way in which Java handles events. This chapter explains how the Java AWT event model works. You will see how to capture events from user interface components and input devices. We also show you how to work with *actions*, a more structured approach for processing action events.

### Basics of Event Handling

Any operating environment that supports GUIs constantly monitors events such as keystrokes or mouse clicks. The operating environment reports these events to the programs that are running. Each program then decides what, if anything, to do in response to these events. In languages like Visual Basic, the correspondence between events and code is obvious. One writes code for each specific event of interest and places the code in what is usually called an *event procedure*. For example, a Visual Basic button named “HelpButton” would have a `HelpButton_Click` event procedure associated with it. The code in this procedure executes whenever that button is clicked. Each Visual Basic GUI component responds to a fixed set of events, and it is impossible to change the events to which a Visual Basic component responds.

On the other hand, if you use a language like raw C to do event-driven programming, you need to write the code that constantly checks the event queue for what the operating environment is reporting. (You usually do this by encasing your code in a loop with a massive switch statement!) This technique is obviously rather ugly, and, in any case, it is much more difficult to code. The advantage is that the events you can respond to are not as limited as in languages, like Visual Basic, that go to great lengths to hide the event queue from the programmer.

The Java programming environment takes an approach somewhat between the Visual Basic approach and the raw C approach in terms of power and, therefore, in resulting complexity. Within the limits of the events that the AWT knows about, you completely control how events are transmitted from the *event sources* (such as buttons or scrollbars) to *event listeners*. You can designate *any* object to be an event listener—in practice, you pick an object that can conveniently carry out the desired response to the event. This *event delegation model* gives you much more flexibility than is possible with Visual Basic, in which the listener is predetermined.

Event sources have methods that allow you to register event listeners with them. When an event happens to the source, the source sends a notification of that event to all the listener objects that were registered for that event.

As one would expect in an object-oriented language like Java, the information about the event is encapsulated in an *event object*. In Java, all event objects ultimately derive from the class `java.util.EventObject`. Of course, there are subclasses for each event type, such as `ActionEvent` and `WindowEvent`.

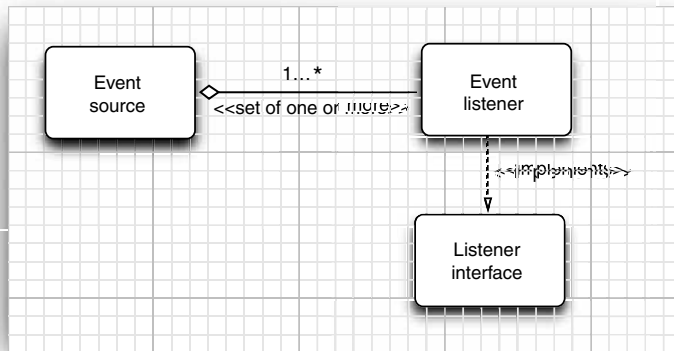
Different event sources can produce different kinds of events. For example, a button can send `ActionEvent` objects, whereas a window can send `WindowEvent` objects.

To sum up, here’s an overview of how event handling in the AWT works:

- A listener object is an instance of a class that implements a special interface called (naturally enough) a *listener interface*.
- An event source is an object that can register listener objects and send them event objects.

- The event source sends out event objects to all registered listeners when that event occurs.
- The listener objects will then use the information in the event object to determine their reaction to the event.

Figure 8-1 shows the relationship between the event handling classes and interfaces.



**Figure 8-1 Relationship between event sources and listeners**

Here is an example for specifying a listener:

```

ActionListener listener = . . . ;
JButton button = new JButton("Ok");
button.addActionListener(listener);
  
```

Now the listener object is notified whenever an “action event” occurs in the button. For buttons, as you might expect, an action event is a button click.

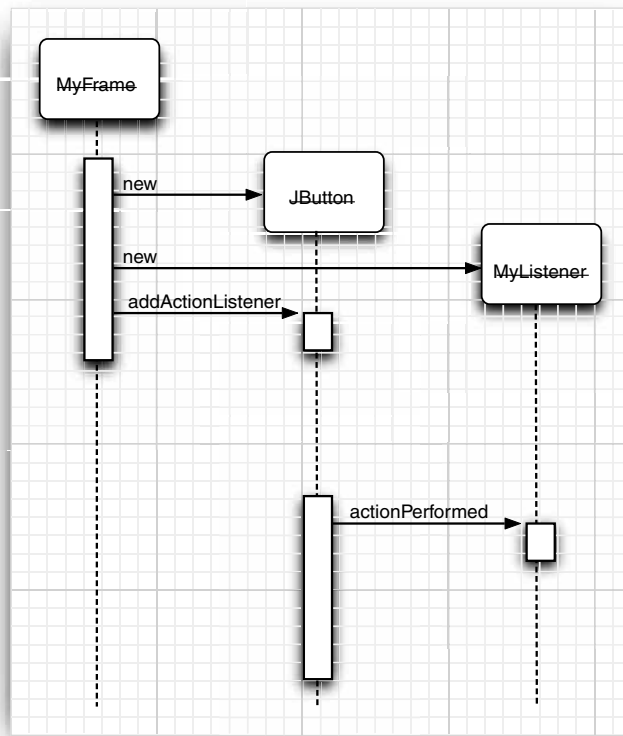
To implement the ActionListener interface, the listener class must have a method called `actionPerformed` that receives an `ActionEvent` object as a parameter.

```

class MyListener implements ActionListener
{
    . . .
    public void actionPerformed(ActionEvent event)
    {
        // reaction to button click goes here
        . . .
    }
}
  
```

Whenever the user clicks the button, the `JButton` object creates an `ActionEvent` object and calls `listener.actionPerformed(event)`, passing that event object. An event source such as a button can have multiple listeners. In that case, the button calls the `actionPerformed` methods of all listeners whenever the user clicks the button.

Figure 8-2 shows the interaction between the event source, event listener, and event object.



**Figure 8-2** Event notification

***Example: Handling a Button Click***

As a way of getting comfortable with the event delegation model, let's work through all details needed for the simple example of responding to a button click. For this example, we will show a panel populated with three buttons. Three listener objects are added as action listeners to the buttons.

With this scenario, each time a user clicks on any of the buttons on the panel, the associated listener object then receives an `ActionEvent` that indicates a button click. In our sample program, the listener object will then change the background color of the panel.

Before we can show you the program that listens to button clicks, we first need to explain how to create buttons and how to add them to a panel. (For more on GUI elements, see Chapter 9.)

You create a button by specifying a label string, an icon, or both in the button constructor. Here are two examples:

```

JButton yellowButton = new JButton("Yellow");
JButton blueButton = new JButton(new ImageIcon("blue-ball.gif"));

```

Call the `add` method to add the buttons to a panel:

```

JButton yellowButton = new JButton("Yellow");
JButton blueButton = new JButton("Blue");
JButton redButton = new JButton("Red");

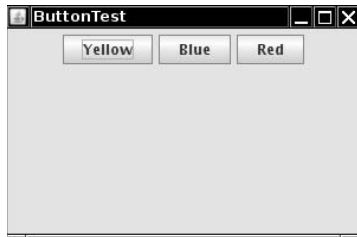
```

```

buttonPanel.add(yellowButton);
buttonPanel.add(blueButton);
buttonPanel.add(redButton);

```

Figure 8–3 shows the result.



**Figure 8–3** A panel filled with buttons

Next, we need to add code that listens to these buttons. This requires classes that implement the `ActionListener` interface, which, as we just mentioned, has one method: `actionPerformed`, whose signature looks like this:

```
public void actionPerformed(ActionEvent event)
```



**NOTE:** The `ActionListener` interface we used in the button example is not restricted to button clicks. It is used in many separate situations:

- When an item is selected from a list box with a double click
- When a menu item is selected
- When the `ENTER` key is clicked in a text field
- When a certain amount of time has elapsed for a `Timer` component

You will see more details in this chapter and the next.

The way to use the `ActionListener` interface is the same in all situations: the `actionPerformed` method (which is the only method in `ActionListener`) takes an object of type `ActionEvent` as a parameter. This event object gives you information about the event that happened.

When a button is clicked, we want the background color of the panel to change to a particular color. We store the desired color in our listener class.

```

class ColorAction implements ActionListener
{
    public ColorAction(Color c)
    {

```

```

        backgroundColor = c;
    }

    public void actionPerformed(ActionEvent event)
    {
        // set panel background color
        . . .
    }

    private Color backgroundColor;
}

```

We then construct one object for each color and set the objects as the button listeners.

```

ColorAction yellowAction = new ColorAction(Color.YELLOW);
ColorAction blueAction = new ColorAction(Color.BLUE);
ColorAction redAction = new ColorAction(Color.RED);

yellowButton.addActionListener(yellowAction);
blueButton.addActionListener(blueAction);
redButton.addActionListener(redAction);

```

For example, if a user clicks on the button marked “Yellow,” then the `actionPerformed` method of the `yellowAction` object is called. Its `backgroundColor` instance field is set to `Color.YELLOW`, and it can now proceed to set the panel’s background color.

Just one issue remains. The `ColorAction` object doesn’t have access to the `buttonPanel` variable. You can solve this problem in two ways. You can store the panel in the `ColorAction` object and set it in the `ColorAction` constructor. Or, more conveniently, you can make `ColorAction` into an inner class of the `ButtonFrame` class. Its methods can then access the outer panel automatically. (For more information on inner classes, see Chapter 6.)

We follow the latter approach. Here is how you place the `ColorAction` class inside the `ButtonFrame` class:

```

class ButtonPanel extends JFrame
{
    . . .

    private class ColorAction implements ActionListener
    {
        . . .

        public void actionPerformed(ActionEvent event)
        {
            buttonPanel.setBackground(backgroundColor);
        }

        private Color backgroundColor;
    }
    private JPanel buttonPanel;
}

```

Look closely at the `actionPerformed` method. The `ColorAction` class doesn’t have a `buttonPanel` field. But the outer `ButtonFrame` class does.



This situation is very common. Event listener objects usually need to carry out some action that affects other objects. You can often strategically place the listener class inside the class whose state the listener should modify.

Listing 8-1 contains the complete program. Whenever you click one of the buttons, the appropriate action listener changes the background color of the panel.

**Listing 8-1** ButtonTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.33 2007-06-12
7.  * @author Cay Horstmann
8.  */
9. public class ButtonTest
10. {
11.     public static void main(String[] args)
12.     {
13.         EventQueue.invokeLater(new Runnable()
14.         {
15.             public void run()
16.             {
17.                 ButtonFrame frame = new ButtonFrame();
18.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.                 frame.setVisible(true);
20.             }
21.         });
22.     }
23. }
24.
25. /**
26.  * A frame with a button panel
27.  */
28. class ButtonFrame extends JFrame
29. {
30.     public ButtonFrame()
31.     {
32.         setTitle("ButtonTest");
33.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34.
35.         // create buttons
36.         JButton yellowButton = new JButton("Yellow");
37.         JButton blueButton = new JButton("Blue");
38.         JButton redButton = new JButton("Red");
39.
40.         buttonPanel = new JPanel();
41.
```

**Listing 8-1** ButtonTest.java (continued)

```
42.    // add buttons to panel
43.    buttonPanel.add(yellowButton);
44.    buttonPanel.add(blueButton);
45.    buttonPanel.add(redButton);
46.
47.    // add panel to frame
48.    add(buttonPanel);
49.
50.    // create button actions
51.    ColorAction yellowAction = new ColorAction(Color.YELLOW);
52.    ColorAction blueAction = new ColorAction(Color.BLUE);
53.    ColorAction redAction = new ColorAction(Color.RED);
54.
55.    // associate actions with buttons
56.    yellowButton.addActionListener(yellowAction);
57.    blueButton.addActionListener(blueAction);
58.    redButton.addActionListener(redAction);
59. }
60.
61. /**
62.  * An action listener that sets the panel's background color.
63.  */
64. private class ColorAction implements ActionListener
65. {
66.     public ColorAction(Color c)
67.     {
68.         backgroundColor = c;
69.     }
70.
71.     public void actionPerformed(ActionEvent event)
72.     {
73.         buttonPanel.setBackground(backgroundColor);
74.     }
75.
76.     private Color backgroundColor;
77. }
78.
79. private JPanel buttonPanel;
80.
81. public static final int DEFAULT_WIDTH = 300;
82. public static final int DEFAULT_HEIGHT = 200;
83. }
```

---

**API** javax.swing.JButton 1.2

- JButton(String label)
- JButton(Icon icon)
- JButton(String label, Icon icon)  
constructs a button. The label string can be plain text or, starting with Java SE 1.3, HTML; for example, "<html><b>0k</b></html>".

**API** java.awt.Container 1.0

- Component add(Component c)  
adds the component c to this container.

**API** javax.swing.ImageIcon 1.2

- ImageIcon(String filename)  
constructs an icon whose image is stored in a file.

***Becoming Comfortable with Inner Classes***

Some people dislike inner classes because they feel that a proliferation of classes and objects makes their programs slower. Let's have a look at that claim. You don't need a new class for every user interface component. In our example, all three buttons share the same listener class. Of course, each of them has a separate listener object. But these objects aren't large. They each contain a color value and a reference to the panel. And the traditional solution, with if . . . else statements, also references the same color objects that the action listeners store, just as local variables and not as instance fields.

Here is a good example of how anonymous inner classes can actually simplify your code. If you look at the code of Listing 8-1, you will note that each button requires the same treatment:

1. Construct the button with a label string.
2. Add the button to the panel.
3. Construct an action listener with the appropriate color.
4. Add that action listener.

Let's implement a helper method to simplify these tasks:

```
public void makeButton(String name, Color backgroundColor)
{
    JButton button = new JButton(name);
    buttonPanel.add(button);
    ColorAction action = new ColorAction(backgroundColor);
    button.addActionListener(action);
}
```

Then we simply call

```
makeButton("yellow", Color.YELLOW);
makeButton("blue", Color.BLUE);
makeButton("red", Color.RED);
```

Now you can make a further simplification. Note that the `ColorAction` class is only needed *once*: in the `makeButton` method. Therefore, you can make it into an anonymous class:

```
public void makeButton(String name, final Color backgroundColor)
{
    JButton button = new JButton(name);
    buttonPanel.add(button);
    button.addActionListener(new ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            buttonPanel.setBackground(backgroundColor);
        }
    });
}
```

The action listener code has become quite a bit simpler. The `actionPerformed` method simply refers to the parameter variable `backgroundColor`. (As with all local variables that are accessed in the inner class, the parameter needs to be declared as `final`.)

No explicit constructor is needed. As you saw in Chapter 6, the inner class mechanism automatically generates a constructor that stores all local `final` variables that are used in one of the methods of the inner class.



**TIP:** Anonymous inner classes can look confusing. But you can get used to deciphering them if you train your eyes to glaze over the routine code, like this:

```
button.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        buttonPanel.setBackground(backgroundColor);
    }
});
```

That is, the button action sets the background color. As long as the event handler consists of just a few statements, we think this can be quite readable, particularly if you don't worry about the inner class mechanics.



**NOTE:** You are completely free to designate *any* object of a class that implements the `ActionListener` interface as a button listener. We prefer to use objects of a new class that was expressly created for carrying out the desired button actions. However, some programmers are not comfortable with inner classes and choose a different strategy. They make the container of the event sources implement the `ActionListener` interface. Then, the container sets *itself* as the listener, like this:

```
yellowButton.addActionListener(this);
blueButton.addActionListener(this);
redButton.addActionListener(this);
```

Now the three buttons no longer have individual listeners. They share a single listener object, namely, the button frame. Therefore, the `actionPerformed` method must figure out which button was clicked.

```

class ButtonFrame extends JFrame implements ActionListener
{
    . . .
    public void actionPerformed(ActionEvent event)
    {
        Object source = event.getSource();
        if (source == yellowButton) . . .
        else if (source == blueButton) . . .
        else if (source == redButton) . . .
        else . . .
    }
}

```

As you can see, this gets quite messy, and we do not recommend it.

#### **API** java.util.EventObject 1.1

- Object getSource()  
returns a reference to the object where the event occurred.

#### **API** java.awt.event.ActionEvent 1.1

- String getActionCommand()  
returns the command string associated with this action event. If the action event originated from a button, the command string equals the button label, unless it has been changed with the setActionCommand method.

#### **API** java.beans.EventHandler 1.4

- static Object create(Class listenerInterface, Object target, String action)
  - static Object create(Class listenerInterface, Object target, String action, String eventProperty)
  - static Object create(Class listenerInterface, Object target, String action, String eventProperty, String listenerMethod)
- constructs an object of a proxy class that implements the given interface. Either the named method or all methods of the interface carry out the given action on the target object.

The action can be a method name or a property of the target. If it is a property, its setter method is executed. For example, an action "text" is turned into a call of the setText method.

The event property consists of one or more dot-separated property names. The first property is read from the parameter of the listener method. The second property is read from the resulting object, and so on. The final result becomes the parameter of the action. For example, the property "source.text" is turned into calls to the getSource and getText methods.

**Creating Listeners Containing a Single Method Call**

Java SE 1.4 introduces a mechanism that lets you specify simple event listeners without programming inner classes. For example, suppose you have a button labeled “Load” whose event handler contains a single method call:

```
frame.loadData();
```

Of course, you can use an anonymous inner class:

```
loadButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        frame.loadData();
    }
});
```

But the `EventHandler` class can create such a listener automatically, with the call

```
EventHandler.create(ActionListener.class, frame, "loadData")
```

Of course, you still need to install the handler:

```
loadButton.addActionListener(
    EventHandler.create(ActionListener.class, frame, "loadData"));
```

If the listener calls a method with a single parameter that can be obtained from the event parameter, you can use another form of the `create` method. For example, the call

```
EventHandler.create(ActionListener.class, frame, "loadData", "source.text")
```

is equivalent to

```
new ActionListener()
{
    public void actionPerformed(ActionEvent event)
    {
        frame.loadData(((JTextField) event.getSource()).getText());
    }
}
```

The property names `source` and `text` turn into method calls `getSource` and `getText`.

**Example: Changing the Look and Feel**

By default, Swing programs use the Metal look and feel. There are two ways to change to a different look and feel. The first way is to supply a file `swing.properties` in the `jrre/lib` subdirectory of your Java installation. In that file, set the property `swing.defaultlaf` to the class name of the look and feel that you want. For example:

```
swing.defaultlaf=com.sun.java.swing.plaf.motif.MotifLookAndFeel
```

Note that the Metal look and feel is located in the `javax.swing` package. The other look-and-feel packages are located in the `com.sun.java` package and need not be present in every Java implementation. Currently, for copyright reasons, the Windows and Macintosh look-and-feel packages are only shipped with the Windows and Macintosh versions of the Java runtime environment.



**TIP:** Because lines starting with a # character are ignored in property files, you can supply several look and feel selections in the `swing.properties` file and move around the # to select one of them:

```
#swing.defaultlaf=javax.swing.plaf.meta1.Meta1LookAndFeel
swing.defaultlaf=com.sun.java.swing.plaf.motif.MotifLookAndFeel
#swing.defaultlaf=com.sun.java.swing.plaf.windows.WindowsLookAndFeel
```

You must restart your program to switch the look and feel in this way. A Swing program reads the `swing.properties` file only once, at startup.

The second way is to change the look and feel dynamically. Call the static `UIManager.setLookAndFeel` method and give it the name of the look-and-feel class that you want. Then call the static method `SwingUtilities.updateComponentTreeUI` to refresh the entire set of components. You need to supply one component to that method; it will find all others. The `UIManager.setLookAndFeel` method may throw a number of exceptions when it can't find the look and feel that you request, or when there is an error loading it. As always, we ask you to gloss over the exception handling code and wait until Chapter 11 for a full explanation.

Here is an example showing how you can switch to the Motif look and feel in your program:

```
String plaf = "com.sun.java.swing.plaf.motif.MotifLookAndFeel";
try
{
    UIManager.setLookAndFeel(plaf);
    SwingUtilities.updateComponentTreeUI(panel);
}
catch(Exception e) { e.printStackTrace(); }
```

To enumerate all installed look and feel implementations, call

```
UIManager.LookAndFeelInfo[] infos = UIManager.getInstalledLookAndFeels();
```

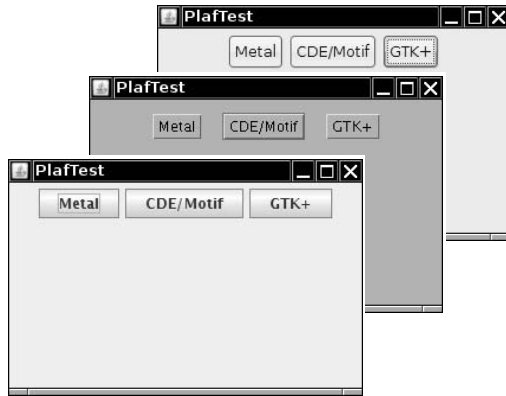
Then you can get the name and class name for each look and feel as

```
String name = infos[i].getName();
String className = infos[i].getClassName();
```

Listing 8-2 is a complete program that demonstrates how to switch the look and feel (see Figure 8-4). The program is similar to Listing 8-1. Following the advice of the preceding section, we use a helper method `makeButton` and an anonymous inner class to specify the button action, namely, to switch the look and feel.

There is one fine point to this program. The `actionPerformed` method of the inner action listener class needs to pass the `this` reference of the outer `PlafFrame` class to the `updateComponentTreeUI` method. Recall from Chapter 6 that the outer object's `this` pointer must be prefixed by the outer class name:

```
SwingUtilities.updateComponentTreeUI(PlafPanel.this);
```



**Figure 8-4** Switching the look and feel

**Listing 8-2** PlafTest.java

```

1. import java.awt.EventQueue;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.32 2007-06-12
7.  * @author Cay Horstmann
8.  */
9. public class PlafTest
10. {
11.     public static void main(String[] args)
12.     {
13.         EventQueue.invokeLater(new Runnable()
14.         {
15.             public void run()
16.             {
17.                 PlafFrame frame = new PlafFrame();
18.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.                 frame.setVisible(true);
20.             }
21.         });
22.     }
23. }
24.
25. /**
26.  * A frame with a button panel for changing look and feel
27.  */
28. class PlafFrame extends JFrame
29. {

```



**Listing 8-2** PlafTest.java (continued)

```
30. public PlafFrame()
31. {
32.     setTitle("PlafTest");
33.     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34.
35.     buttonPanel = new JPanel();
36.
37.     UIManager.LookAndFeelInfo[] infos = UIManager.getInstalledLookAndFeels();
38.     for (UIManager.LookAndFeelInfo info : infos)
39.         makeButton(info.getName(), info.getClassName());
40.
41.     add(buttonPanel);
42. }
43.
44. /**
45.  * Makes a button to change the pluggable look and feel.
46.  * @param name the button name
47.  * @param plafName the name of the look and feel class
48.  */
49. void makeButton(String name, final String plafName)
50. {
51.     // add button to panel
52.
53.     JButton button = new JButton(name);
54.     buttonPanel.add(button);
55.
56.     // set button action
57.
58.     button.addActionListener(new ActionListener()
59.     {
60.         public void actionPerformed(ActionEvent event)
61.         {
62.             // button action: switch to the new look and feel
63.             try
64.             {
65.                 UIManager.setLookAndFeel(plafName);
66.                 SwingUtilities.updateComponentTreeUI(PlafFrame.this);
67.             }
68.             catch (Exception e)
69.             {
70.                 e.printStackTrace();
71.             }
72.         }
73.     });
74. }
75.
76. private JPanel buttonPanel;
77.
78. public static final int DEFAULT_WIDTH = 300;
79. public static final int DEFAULT_HEIGHT = 200;
80. }
```

**API** javax.swing.UIManager 1.2

- static UIManager.LookAndFeelInfo[] getInstalledLookAndFeels() gets an array of objects that describe the installed look-and-feel implementations.
- static setLookAndFeel(String className) sets the current look and feel, using the given class name (such as "javax.swing.plaf.metal.MetalLookAndFeel").

**API** javax.swing.UIManager.LookAndFeelInfo 1.2

- String getName() returns the display name for the look and feel.
- String getClassName() returns the name of the implementation class for the look and feel.

**Adapter Classes**

Not all events are as simple to handle as button clicks. In a non-toy program, you will want to monitor when the user tries to close the main frame because you don't want your users to lose unsaved work. When the user closes the frame, you want to put up a dialog and exit the program only when the user agrees.

When the program user tries to close a frame window, the JFrame object is the source of a WindowEvent. If you want to catch that event, you must have an appropriate listener object and add it to the frame's list of window listeners.

```
WindowListener listener = . . . ;
frame.addWindowListener(listener);
```

The window listener must be an object of a class that implements the WindowListener interface. There are actually seven methods in the WindowListener interface. The frame calls them as the responses to seven distinct events that could happen to a window. The names are self-explanatory, except that "iconified" is usually called "minimized" under Windows. Here is the complete WindowListener interface:

```
public interface WindowListener
{
    void windowOpened(WindowEvent e);
    void windowClosing(WindowEvent e);
    void windowClosed(WindowEvent e);
    void windowIconified(WindowEvent e);
    void windowDeiconified(WindowEvent e);
    void windowActivated(WindowEvent e);
    void windowDeactivated(WindowEvent e);
}
```



**NOTE:** To find out whether a window has been maximized, install a WindowStateListener. See the API notes on page 341 for details.

As is always the case in Java, any class that implements an interface must implement all its methods; in this case, that means implementing *seven* methods. Recall that we are only interested in one of these seven methods, namely, the windowClosing method.

Of course, we can define a class that implements the interface, add a call to `System.exit(0)` in the `windowClosing` method, and write do-nothing functions for the other six methods:

```
class Terminator implements WindowListener
{
    public void windowClosing(WindowEvent e)
    {
        if (user agrees)
            System.exit(0);
    }

    public void windowOpened(WindowEvent e) {}
    public void windowClosed(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
}
```

Typing code for six methods that don't do anything is the kind of tedious busywork that nobody likes. To simplify this task, each of the AWT listener interfaces that has more than one method comes with a companion *adapter* class that implements all the methods in the interface but does nothing with them. For example, the `WindowAdapter` class has seven do-nothing methods. This means the adapter class automatically satisfies the technical requirements that Java imposes for implementing the associated listener interface. You can extend the adapter class to specify the desired reactions to some, but not all, of the event types in the interface. (An interface such as `ActionListener` that has only a single method does not need an adapter class.)

Let us make use of the window adapter. We can extend the `WindowAdapter` class, inherit six of the do-nothing methods, and override the `windowClosing` method:

```
class Terminator extends WindowAdapter
{
    public void windowClosing(WindowEvent e)
    {
        if (user agrees)
            System.exit(0);
    }
}
```

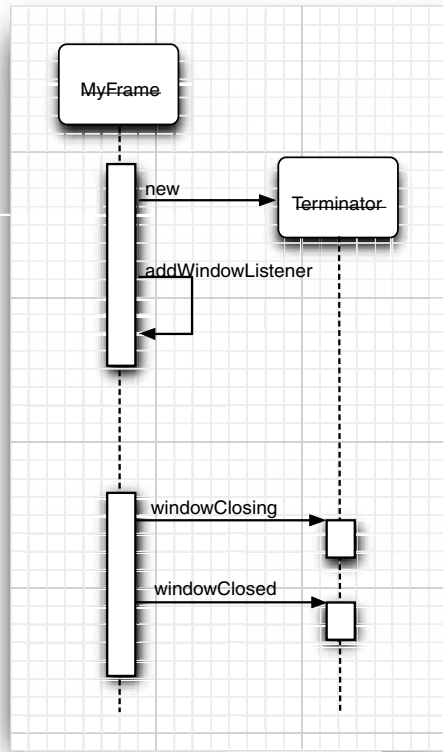
Now you can register an object of type `Terminator` as the event listener:

```
WindowListener listener = new Terminator();
frame.addWindowListener(listener);
```

Whenever the frame generates a window event, it passes it to the listener object by calling one of its seven methods (see Figure 8-5). Six of those methods do nothing; the `windowClosing` method calls `System.exit(0)`, terminating the application.



**CAUTION:** If you misspell the name of a method when extending an adapter class, then the compiler won't catch your error. For example, if you define a method `windowIsClosing` in a `WindowAdapter` class, then you get a class with eight methods, and the `windowClosing` method does nothing.

**Figure 8-5 A window listener**

Creating a listener class that extends the `WindowAdapter` is an improvement, but we can go even further. There is no need to give a name to the listener object. Simply write

```
frame.addWindowListener(new Terminator());
```

But why stop there? We can make the listener class into an anonymous inner class of the frame.

```
frame.addWindowListener(new
    WindowAdapter()
    {
        public void windowClosing(WindowEvent e)
        {
            if (user agrees)
                System.exit(0);
        }
    });
```

This code does the following:

- Defines a class without a name that extends the `WindowAdapter` class
- Adds a `windowClosing` method to that anonymous class (as before, this method exits the program)
- Inherits the remaining six do-nothing methods from `WindowAdapter`
- Creates an object of this class; that object does not have a name, either
- Passes that object to the `addWindowListener` method

We say again that the syntax for using anonymous inner classes takes some getting used to. The payoff is that the resulting code is as short as possible.

**API** `java.awt.event.WindowListener` 1.1

- `void windowOpened(WindowEvent e)`  
is called after the window has been opened.
- `void windowClosing(WindowEvent e)`  
is called when the user has issued a window manager command to close the window. Note that the window will close only if its `hide` or `dispose` method is called.
- `void windowClosed(WindowEvent e)`  
is called after the window has closed.
- `void windowIconified(WindowEvent e)`  
is called after the window has been iconified.
- `void windowDeiconified(WindowEvent e)`  
is called after the window has been deiconified.
- `void windowActivated(WindowEvent e)`  
is called after the window has become active. Only a frame or dialog can be active. Typically, the window manager decorates the active window, for example, by highlighting the title bar.
- `void windowDeactivated(WindowEvent e)`  
is called after the window has become deactivated.

**API** `java.awt.event.WindowStateListener` 1.4

- `void windowStateChanged(WindowEvent event)`  
is called after the window has been maximized, iconified, or restored to normal size.

**API** `java.awt.event.WindowEvent` 1.1

- `int getNewState()` 1.4
  - `int getOldState()` 1.4
- returns the new and old state of a window in a window state change event. The returned integer is one of the following values:

```
Frame.NORMAL  
Frame.ICONIFIED  
Frame.MAXIMIZED_HORIZ  
Frame.MAXIMIZED_VERT  
Frame.MAXIMIZED_BOTH
```

## Actions

It is common to have multiple ways to activate the same command. The user can choose a certain function through a menu, a keystroke, or a button on a toolbar. This is easy to achieve in the AWT event model: link all events to the same listener. For example, suppose `blueAction` is an action listener whose `actionPerformed` method changes the background color to blue. You can attach the same object as a listener to several event sources:

- A toolbar button labeled “Blue”
- A menu item labeled “Blue”
- A keystroke `CTRL+B`

Then the color change command is handled in a uniform way, no matter whether it was caused by a button click, a menu selection, or a key press.

The Swing package provides a very useful mechanism to encapsulate commands and to attach them to multiple event sources: the `Action` interface. An *action* is an object that encapsulates

- A description of the command (as a text string and an optional icon); and
- Parameters that are necessary to carry out the command (such as the requested color in our example).

The `Action` interface has the following methods:

```
void actionPerformed(ActionEvent event)
void setEnabled(boolean b)
boolean isEnabled()
void putValue(String key, Object value)
Object getValue(String key)
void addPropertyChangeListener(PropertyChangeListener listener)
void removePropertyChangeListener(PropertyChangeListener listener)
```

The first method is the familiar method in the `ActionListener` interface: in fact, the `Action` interface extends the `ActionListener` interface. Therefore, you can use an `Action` object whenever an `ActionListener` object is expected.

The next two methods let you enable or disable the action and check whether the action is currently enabled. When an action is attached to a menu or toolbar and the action is disabled, then the option is grayed out.

The `putValue` and `getValue` methods let you store and retrieve arbitrary name/value pairs in the action object. A couple of important predefined strings, namely, `Action.NAME` and `Action.SMALL_ICON`, store action names and icons into an action object:

```
action.putValue(Action.NAME, "Blue");
action.putValue(Action.SMALL_ICON, new ImageIcon("blue-ball.gif"));
```

Table 8–1 shows all predefined action table names.

If the action object is added to a menu or toolbar, then the name and icon are automatically retrieved and displayed in the menu item or toolbar button. The `SHORT_DESCRIPTION` value turns into a tooltip.

The final two methods of the `Action` interface allow other objects, in particular menus or toolbars that trigger the action, to be notified when the properties of the action object change. For example, if a menu is added as a property change listener of an action object and the action object is subsequently disabled, then the menu is called

and can gray out the action name. Property change listeners are a general construct that is a part of the “JavaBeans” component model. You can find out more about beans and their properties in Volume II.

**Table 8-1 Predefined Action Table Names**

Name	Value
NAME	The name of the action; displayed on buttons and menu items.
SMALL_ICON	A place to store a small icon; for display in a button, menu item, or toolbar.
SHORT_DESCRIPTION	A short description of the icon; for display in a tooltip.
LONG_DESCRIPTION	A long description of the icon; for potential use in on-line help. No Swing component uses this value.
MNEMONIC_KEY	A mnemonic abbreviation; for display in menu items (see Chapter 9).
ACCELERATOR_KEY	A place to store an accelerator keystroke. No Swing component uses this value.
ACTION_COMMAND_KEY	Historically, used in the now obsolete <code>registerKeyboardAction</code> method.
DEFAULT	Potentially useful catch-all property. No Swing component uses this value.

Note that `Action` is an *interface*, not a class. Any class implementing this interface must implement the seven methods we just discussed. Fortunately, a friendly soul has provided a class `AbstractAction` that implements all methods except for `actionPerformed`. That class takes care of storing all name/value pairs and managing the property change listeners. You simply extend `AbstractAction` and supply an `actionPerformed` method.

Let’s build an action object that can execute color change commands. We store the name of the command, an icon, and the desired color. We store the color in the table of name/value pairs that the `AbstractAction` class provides. Here is the code for the `ColorAction` class. The constructor sets the name/value pairs, and the `actionPerformed` method carries out the color change action.

```
public class ColorAction extends AbstractAction
{
    public ColorAction(String name, Icon icon, Color c)
    {
        putValue(Action.NAME, name);
        putValue(Action.SMALL_ICON, icon);
        putValue("color", c);
        putValue(Action.SHORT_DESCRIPTION, "Set panel color to " + name.toLowerCase());
    }

    public void actionPerformed(ActionEvent event)
    {
        Color c = (Color) getValue("color");
        buttonPanel.setBackground(c);
    }
}
```

Our test program creates three objects of this class, such as

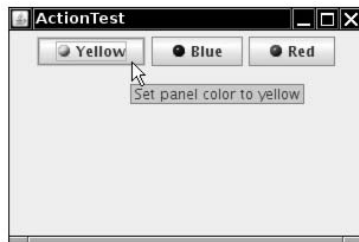
```
Action blueAction = new ColorAction("Blue", new ImageIcon("blue-ball.gif"), Color.BLUE);
```

Next, let's associate this action with a button. That is easy because we can use a  `JButton`  constructor that takes an  `Action`  object.

```
JButton blueButton = new JButton(blueAction);
```

That constructor reads the name and icon from the action, sets the short description as the tooltip, and sets the action as the listener. You can see the icons and a tooltip in Figure 8-6.

As we demonstrate in the next chapter, it is just as easy to add the same action to a menu.



**Figure 8-6** Buttons display the icons from the action objects

Finally, we want to add the action objects to keystrokes so that the actions are carried out when the user types keyboard commands. To associate actions with keystrokes, you first need to generate objects of the  `KeyStroke`  class. This is a convenience class that encapsulates the description of a key. To generate a  `KeyStroke`  object, you don't call a constructor but instead use the static  `getKeyStroke`  method of the  `KeyStroke`  class.

```
KeyStroke ctrlBKey = KeyStroke.getKeyStroke("ctrl B");
```

To understand the next step, you need to know the concept of *keyboard focus*. A user interface can have many buttons, menus, scrollbars, and other components. When you hit a key, it is sent to the component that has focus. That component is usually (but not always) visually distinguished. For example, in the Java Look and Feel, a button with focus has a thin rectangular border around the button text. You can use the  `TAB`  key to move the focus between components. When you press the  `SPACE`  key, the button with focus is clicked. Other keys carry out different actions; for example, the arrow keys can move a scrollbar.

However, in our case, we do not want to send the keystroke to the component that has focus. Otherwise, each of the buttons would need to know how to handle the  `CTRL+Y` ,  `CTRL+B` , and  `CTRL+R`  keys.

This is a common problem, and the Swing designers came up with a convenient solution for solving it. Every  `JComponent`  has three *input maps*, each mapping  `KeyStroke`  objects to associated actions. The three input maps correspond to three different conditions (see Table 8-2).



Keystroke processing checks these maps in the following order:

1. Check the `WHEN_FOCUSED` map of the component with input focus. If the keystroke exists, execute the corresponding action. If the action is enabled, stop processing.
2. Starting from the component with input focus, check the `WHEN_ANCESTOR_OF_FOCUSED_COMPONENT` maps of its parent components. As soon as a map with the keystroke is found, execute the corresponding action. If the action is enabled, stop processing.
3. Look at all *visible* and *enabled* components in the window with input focus that have this keystroke registered in a `WHEN_IN_FOCUSED_WINDOW` map. Give these components (in the order of their keystroke registration) a chance to execute the corresponding action. As soon as the first enabled action is executed, stop processing. This part of the process is somewhat fragile if a keystroke appears in more than one `WHEN_IN_FOCUSED_WINDOW` map.

**Table 8–2 Input Map Conditions**

Flag	Invoke Action
<code>WHEN_FOCUSED</code>	When this component has keyboard focus
<code>WHEN_ANCESTOR_OF_FOCUSED_COMPONENT</code>	When this component contains the component that has keyboard focus
<code>WHEN_IN_FOCUSED_WINDOW</code>	When this component is contained in the same window as the component that has keyboard focus

You obtain an input map from the component with the `getInputMap` method. Here is an example:

```
InputMap imap = panel.getInputMap(JComponent.WHEN_FOCUSED);
```

The `WHEN_FOCUSED` condition means that this map is consulted when the current component has the keyboard focus. In our situation, that isn't the map we want. One of the buttons, not the panel, has the input focus. Either of the other two map choices works fine for inserting the color change keystrokes. We use `WHEN_ANCESTOR_OF_FOCUSED_COMPONENT` in our example program.

The `InputMap` doesn't directly map `KeyStroke` objects to `Action` objects. Instead, it maps to arbitrary objects, and a second map, implemented by the `ActionMap` class, maps objects to actions. That makes it easier to share the same actions among keystrokes that come from different input maps.

Thus, each component has three input maps and one action map. To tie them together, you need to come up with names for the actions. Here is how you can tie a key to an action:

```
imap.put(KeyStroke.getKeyStroke("ctrl Y"), "panel.yellow");
ActionMap amap = panel.getActionMap();
amap.put("panel.yellow", yellowAction);
```

It is customary to use the string "none" for a do-nothing action. That makes it easy to deactivate a key:

```
imap.put(KeyStroke.getKeyStroke("ctrl C"), "none");
```

**X** CAUTION: The JDK documentation suggests using the action name as the action's key. We don't think that is a good idea. The action name is displayed on buttons and menu items; thus, it can change at the whim of the UI designer and it may be translated into multiple languages. Such unstable strings are poor choices for lookup keys. Instead, we recommend that you come up with action names that are independent of the displayed names.

To summarize, here is what you do to carry out the same action in response to a button, a menu item, or a keystroke:

1. Implement a class that extends the `AbstractAction` class. You may be able to use the same class for multiple related actions.
2. Construct an object of the action class.
3. Construct a button or menu item from the action object. The constructor will read the label text and icon from the action object.
4. For actions that can be triggered by keystrokes, you have to carry out additional steps. First locate the top-level component of the window, such as a panel that contains all other components.
5. Then get the `WHEN_ANCESTOR_OF_FOCUSED_COMPONENT` input map of the top-level component. Make a `KeyStroke` object for the desired keystroke. Make an action key object, such as a string that describes your action. Add the pair (keystroke, action key) into the input map.
6. Finally, get the action map of the top-level component. Add the pair (action key, action object) into the map.

Listing 8-3 shows the complete code of the program that maps both buttons and keystrokes to action objects. Try it out—clicking either the buttons or pressing `CTRL+Y`, `CTRL+B`, or `CTRL+R` changes the panel color.

**Listing 8-3** ActionTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.33 2007-06-12
7.  * @author Cay Horstmann
8.  */
9. public class ActionTest
10. {
11.     public static void main(String[] args)
12.     {
13.         EventQueue.invokeLater(new Runnable()
14.         {
15.             public void run()
16.             {
```

**Listing 8-3** ActionTest.java (continued)

```
17.         ActionFrame frame = new ActionFrame();
18.         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.         frame.setVisible(true);
20.     }
21. });
22. }
23. }
24.
25. /**
26.  * A frame with a panel that demonstrates color change actions.
27.  */
28. class ActionFrame extends JFrame
29. {
30.     public ActionFrame()
31.     {
32.         setTitle("ActionTest");
33.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34.
35.         buttonPanel = new JPanel();
36.
37.         // define actions
38.         Action yellowAction = new ColorAction("Yellow", new ImageIcon("yellow-ball.gif"),
39.             Color.YELLOW);
40.         Action blueAction = new ColorAction("Blue", new ImageIcon("blue-ball.gif"), Color.BLUE);
41.         Action redAction = new ColorAction("Red", new ImageIcon("red-ball.gif"), Color.RED);
42.
43.         // add buttons for these actions
44.         buttonPanel.add(new JButton(yellowAction));
45.         buttonPanel.add(new JButton(blueAction));
46.         buttonPanel.add(new JButton(redAction));
47.
48.         // add panel to frame
49.         add(buttonPanel);
50.
51.         // associate the Y, B, and R keys with names
52.         InputMap imap = buttonPanel.getInputMap(JComponent.WHEN_ANCESTOR_OF_FOCUSED_COMPONENT);
53.         imap.put(KeyStroke.getKeyStroke("ctrl Y"), "panel.yellow");
54.         imap.put(KeyStroke.getKeyStroke("ctrl B"), "panel.blue");
55.         imap.put(KeyStroke.getKeyStroke("ctrl R"), "panel.red");
56.
57.         // associate the names with actions
58.         ActionMap amap = buttonPanel.getActionMap();
59.         amap.put("panel.yellow", yellowAction);
60.         amap.put("panel.blue", blueAction);
61.         amap.put("panel.red", redAction);
62.     }
63.
64.     public class ColorAction extends AbstractAction
65.     {
```

**Listing 8-3** ActionTest.java (continued)

```
66.  /**
67.   * Constructs a color action.
68.   * @param name the name to show on the button
69.   * @param icon the icon to display on the button
70.   * @param c the background color
71.   */
72.  public ColorAction(String name, Icon icon, Color c)
73.  {
74.      putValue(Action.NAME, name);
75.      putValue(Action.SMALL_ICON, icon);
76.      putValue(Action.SHORT_DESCRIPTION, "Set panel color to " + name.toLowerCase());
77.      putValue("color", c);
78.  }
79.
80.  public void actionPerformed(ActionEvent event)
81.  {
82.      Color c = (Color) getValue("color");
83.      buttonPanel.setBackground(c);
84.  }
85. }
86.
87. private JPanel buttonPanel;
88.
89. public static final int DEFAULT_WIDTH = 300;
90. public static final int DEFAULT_HEIGHT = 200;
91. }
```

**API** javax.swing.Action 1.2

- boolean isEnabled()
- void setEnabled(boolean b)  
gets or sets the enabled property of this action.
- void putValue(String key, Object value)  
places a name/value pair inside the action object.  

<i>Parameters:</i>	key	The name of the feature to store with the action object. This can be any string, but several names have predefined meanings—see Table 8-1 on page 343.
	value	The object associated with the name.
- Object getValue(String key)  
returns the value of a stored name/value pair.

**API** `javax.swing.KeyStroke` 1.2

- static `KeyStroke getKeyStroke(String description)`  
constructs a keystroke from a humanly readable description (a sequence of whitespace-delimited strings). The description starts with zero or more modifiers `shift control ctrl meta alt altGraph` and ends with either the string `typed`, followed by a one-character string (for example, "typed a"), or an optional event specifier (`pressed`—the default—or `released`), followed by a key code. The key code, when prefixed with `VK_`, should correspond to a `KeyEvent` constant; for example, "INSERT" corresponds to `KeyEvent.VK_INSERT`.

**API** `javax.swing.JComponent` 1.2

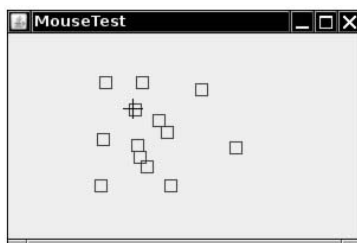
- `ActionMap getActionMap()` 1.3  
returns the map that associates action map keys (which can be arbitrary objects) with Action objects.
- `InputMap getInputMap(int flag)` 1.3  
gets the input map that maps key strokes to action map keys.

*Parameters:*    `flag`            A condition on the keyboard focus to trigger the action, one of the values in Table 8–2 on page 345

**Mouse Events**

You do not need to handle mouse events explicitly if you just want the user to be able to click on a button or menu. These mouse operations are handled internally by the various components in the user interface. However, if you want to enable the user to draw with the mouse, you will need to trap mouse move, click, and drag events.

In this section, we show you a simple graphics editor application that allows the user to place, move, and erase squares on a canvas (see Figure 8–7).



**Figure 8–7** A mouse test program

When the user clicks a mouse button, three listener methods are called: `mousePressed` when the mouse is first pressed, `mouseReleased` when the mouse is released, and, finally, `mouseClicked`. If you are only interested in complete clicks, you can ignore the first two methods. By using the `getX` and `getY` methods on the `MouseEvent` argument, you can obtain the  $x$ - and  $y$ -coordinates of the mouse pointer when the mouse was clicked. To distinguish between single, double, and triple (!) clicks, use the `getClickCount` method.

Some user interface designers inflict mouse click and keyboard modifier combinations, such as CONTROL + SHIFT + CLICK, on their users. We find this practice reprehensible, but if you disagree, you will find that checking for mouse buttons and keyboard modifiers is a mess.

You use bit masks to test which modifiers have been set. In the original API, two of the button masks equal two keyboard modifier masks, namely

```
BUTTON2_MASK == ALT_MASK
BUTTON3_MASK == META_MASK
```

This was done so that users with a one-button mouse could simulate the other mouse buttons by holding down modifier keys instead. However, as of Java SE 1.4, a different approach is recommended. There are now masks

```
BUTTON1_DOWN_MASK
BUTTON2_DOWN_MASK
BUTTON3_DOWN_MASK
SHIFT_DOWN_MASK
CTRL_DOWN_MASK
ALT_DOWN_MASK
ALT_GRAPH_DOWN_MASK
META_DOWN_MASK
```

The `getModifiersEx` method accurately reports the mouse buttons and keyboard modifiers of a mouse event.

Note that `BUTTON3_DOWN_MASK` tests for the right (nonprimary) mouse button under Windows. For example, you can use code like this to detect whether the right mouse button is down:

```
if ((event.getModifiersEx() & InputEvent.BUTTON3_DOWN_MASK) != 0)
    . . . // code for right click
```

In our sample program, we supply both a `mousePressed` and a `mouseClicked` method. When you click onto a pixel that is not inside any of the squares that have been drawn, a new square is added. We implemented this in the `mousePressed` method so that the user receives immediate feedback and does not have to wait until the mouse button is released. When a user double-clicks inside an existing square, it is erased. We implemented this in the `mouseClicked` method because we need the click count.














```
public void mousePressed(MouseEvent event)
{
    current = find(event.getPoint());
    if (current == null) // not inside a square
        add(event.getPoint());
}

public void mouseClicked(MouseEvent event)
{
    current = find(event.getPoint());
    if (current != null && event.getClickCount() >= 2)
        remove(current);
}
```

As the mouse moves over a window, the window receives a steady stream of mouse movement events. Note that there are separate `MouseListener` and `MouseMotionListener` interfaces. This is done for efficiency—there are a lot of mouse events as the user moves the mouse around, and a listener that just cares about mouse *clicks* will not be bothered with unwanted mouse *moves*.

Our test application traps mouse motion events to change the cursor to a different shape (a cross hair) when it is over a square. This is done with the `getPredefinedCursor` method of the `Cursor` class. Table 8–3 lists the constants to use with this method along with what the cursors look like under Windows.

**Table 8–3 Sample Cursor Shapes**

Icon	Constant	Icon	Constant
	DEFAULT_CURSOR		NE_RESIZE_CURSOR
	CROSSHAIR_CURSOR		E_RESIZE_CURSOR
	HAND_CURSOR		SE_RESIZE_CURSOR
	MOVE_CURSOR		S_RESIZE_CURSOR
	TEXT_CURSOR		SW_RESIZE_CURSOR
	WAIT_CURSOR		W_RESIZE_CURSOR
	N_RESIZE_CURSOR		NW_RESIZE_CURSOR

Here is the `mouseMoved` method of the `MouseMotionListener` in our example program:

```
public void mouseMoved(MouseEvent event)
{
    if (find(event.getPoint()) == null)
        setCursor(Cursor.getDefaultCursor());
    else
        setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
}
```



NOTE: You can also define your own cursor types through the use of the `createCustomCursor` method in the `Toolkit` class:

```
Toolkit tk = Toolkit.getDefaultToolkit();
Image img = tk.getImage("dynamite.gif");
Cursor dynamiteCursor = tk.createCustomCursor(img, new Point(10, 10), "dynamite stick");
```

The first parameter of the `createCustomCursor` points to the cursor image. The second parameter gives the offset of the "hot spot" of the cursor. The third parameter is a string that describes the cursor. This string can be used for accessibility support. For example, a screen reader program can read the cursor shape description to a user who is visually impaired or who simply is not facing the screen.

If the user presses a mouse button while the mouse is in motion, `mouseDragged` calls are generated instead of `mouseMoved` calls. Our test application lets a user drag the square under the cursor. We simply update the currently dragged rectangle to be centered under the mouse position. Then, we repaint the canvas to show the new mouse position.

```
public void mouseDragged(MouseEvent event)
{
    if (current != null)
    {
        int x = event.getX();
        int y = event.getY();

        current setFrame(x - SIDELENGTH / 2, y - SIDELENGTH / 2, SIDELENGTH, SIDELENGTH);
        repaint();
    }
}
```



NOTE: The `mouseMoved` method is only called as long as the mouse stays inside the component. However, the `mouseDragged` method keeps getting called even when the mouse is being dragged outside the component.

There are two other mouse event methods: `mouseEntered` and `mouseExited`. These methods are called when the mouse enters or exits a component.

Finally, we explain how to listen to mouse events. Mouse clicks are reported through the `mouseClicked` procedure, which is part of the `MouseListener` interface. Because many applications are interested only in mouse clicks and not in mouse moves and because mouse move events occur so frequently, the mouse move and drag events are defined in a separate interface called `MouseMotionListener`.

In our program we are interested in both types of mouse events. We define two inner classes: `MouseHandler` and `MouseMotionHandler`. The `MouseHandler` class extends the `MouseAdapter` class because it defines only two of the five `MouseListener` methods. The `MouseMotionHandler` implements the `MouseMotionListener` and defines both methods of that interface. Listing 8-4 is the program listing.



**Listing 8-4** MouseTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.util.*;
4. import java.awt.geom.*;
5. import javax.swing.*;
6.
7. /**
8.  * @version 1.32 2007-06-12
9.  * @author Cay Horstmann
10. */
11. public class MouseTest
12. {
13.     public static void main(String[] args)
14.     {
15.         EventQueue.invokeLater(new Runnable()
16.         {
17.             public void run()
18.             {
19.                 MouseFrame frame = new MouseFrame();
20.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21.                 frame.setVisible(true);
22.             }
23.         });
24.     }
25. }
26.
27. /**
28.  * A frame containing a panel for testing mouse operations
29.  */
30. class MouseFrame extends JFrame
31. {
32.     public MouseFrame()
33.     {
34.         setTitle("MouseTest");
35.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
36.
37.         // add component to frame
38.
39.         MouseComponent component = new MouseComponent();
40.         add(component);
41.     }
42.
43.     public static final int DEFAULT_WIDTH = 300;
44.     public static final int DEFAULT_HEIGHT = 200;
45. }
46.
47. /**
48.  * A component with mouse operations for adding and removing squares.
49.  */
```

**Listing 8-4** MouseTest.java (continued)

```
50. class MouseComponent extends JComponent
51. {
52.     public MouseComponent()
53.     {
54.         squares = new ArrayList<Rectangle2D>();
55.         current = null;
56.
57.         addMouseListener(new MouseHandler());
58.         addMouseMotionListener(new MouseMotionHandler());
59.     }
60.
61.     public void paintComponent(Graphics g)
62.     {
63.         Graphics2D g2 = (Graphics2D) g;
64.
65.         // draw all squares
66.         for (Rectangle2D r : squares)
67.             g2.draw(r);
68.     }
69.
70.     /**
71.      * Finds the first square containing a point.
72.      * @param p a point
73.      * @return the first square that contains p
74.      */
75.     public Rectangle2D find(Point2D p)
76.     {
77.         for (Rectangle2D r : squares)
78.         {
79.             if (r.contains(p)) return r;
80.         }
81.         return null;
82.     }
83.
84.     /**
85.      * Adds a square to the collection.
86.      * @param p the center of the square
87.      */
88.     public void add(Point2D p)
89.     {
90.         double x = p.getX();
91.         double y = p.getY();
92.
93.         current = new Rectangle2D.Double(x - SIDELENGTH / 2, y - SIDELENGTH / 2, SIDELENGTH,
94.             SIDELENGTH);
95.         squares.add(current);
96.         repaint();
97.     }
98.
```

**Listing 8-4** MouseTest.java (continued)

```
99.  /**
100.   * Removes a square from the collection.
101.   * @param s the square to remove
102.   */
103.  public void remove(Rectangle2D s)
104.  {
105.      if (s == null) return;
106.      if (s == current) current = null;
107.      squares.remove(s);
108.      repaint();
109.  }
110.
111.  private static final int SIDELENGTH = 10;
112.  private ArrayList<Rectangle2D> squares;
113.  private Rectangle2D current;
114.
115.  // the square containing the mouse cursor
116.
117.  private class MouseHandler extends MouseAdapter
118.  {
119.      public void mousePressed(MouseEvent event)
120.      {
121.          // add a new square if the cursor isn't inside a square
122.          current = find(event.getPoint());
123.          if (current == null) add(event.getPoint());
124.      }
125.
126.      public void mouseClicked(MouseEvent event)
127.      {
128.          // remove the current square if double clicked
129.          current = find(event.getPoint());
130.          if (current != null && event.getClickCount() >= 2) remove(current);
131.      }
132.  }
133.
134.  private class MouseMotionHandler implements MouseMotionListener
135.  {
136.      public void mouseMoved(MouseEvent event)
137.      {
138.          // set the mouse cursor to cross hairs if it is inside
139.          // a rectangle
140.
141.          if (find(event.getPoint()) == null) setCursor(Cursor.getDefaultCursor());
142.          else setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR));
143.      }
144.
145.      public void mouseDragged(MouseEvent event)
146.      {
```

**Listing 8-4** MouseTest.java (continued)

```

147.     if (current != null)
148.     {
149.         int x = event.getX();
150.         int y = event.getY();
151.
152.         // drag the current rectangle to center it at (x, y)
153.         current setFrame(x - SIDELENGTH / 2, y - SIDELENGTH / 2, SIDELENGTH, SIDELENGTH);
154.         repaint();
155.     }
156. }
157. }
158. }

```

**API** java.awt.event.MouseEvent 1.1

- int getX()
- int getY()
- Point getPoint()  
returns the *x*- (horizontal) and *y*- (vertical) coordinate, or point where the event happened, measured from the top-left corner of the component that is the event source.
- int getClickCount()  
returns the number of consecutive mouse clicks associated with this event. (The time interval for what constitutes “consecutive” is system dependent.)

**API** java.awt.event.InputEvent 1.1

- int getModifiersEx() 1.4  
returns the extended or “down” modifiers for this event. Use the following mask values to test the returned value:  
 BUTTON1\_DOWN\_MASK  
 BUTTON2\_DOWN\_MASK  
 BUTTON3\_DOWN\_MASK  
 SHIFT\_DOWN\_MASK  
 CTRL\_DOWN\_MASK  
 ALT\_DOWN\_MASK  
 ALT\_GRAPH\_DOWN\_MASK  
 META\_DOWN\_MASK
- static String getModifiersExText(int modifiers) 1.4  
returns a string such as “Shift+Button1” describing the extended or “down” modifiers in the given flag set.

**API** `java.awt.Toolkit` 1.0

- `public Cursor createCustomCursor(Image image, Point hotSpot, String name)` 1.2  
creates a new custom cursor object.

<i>Parameters:</i>	<code>image</code>	The image to display when the cursor is active
	<code>hotSpot</code>	The cursor's hot spot (such as the tip of an arrow or the center of cross hairs)
	<code>name</code>	A description of the cursor, to support special accessibility environments

**API** `java.awt.Component` 1.0

- `public void setCursor(Cursor cursor)` 1.1  
sets the cursor image to the specified cursor.

**The AWT Event Hierarchy**

Having given you a taste of how event handling works, we finish this chapter with an overview of the AWT event handling architecture.

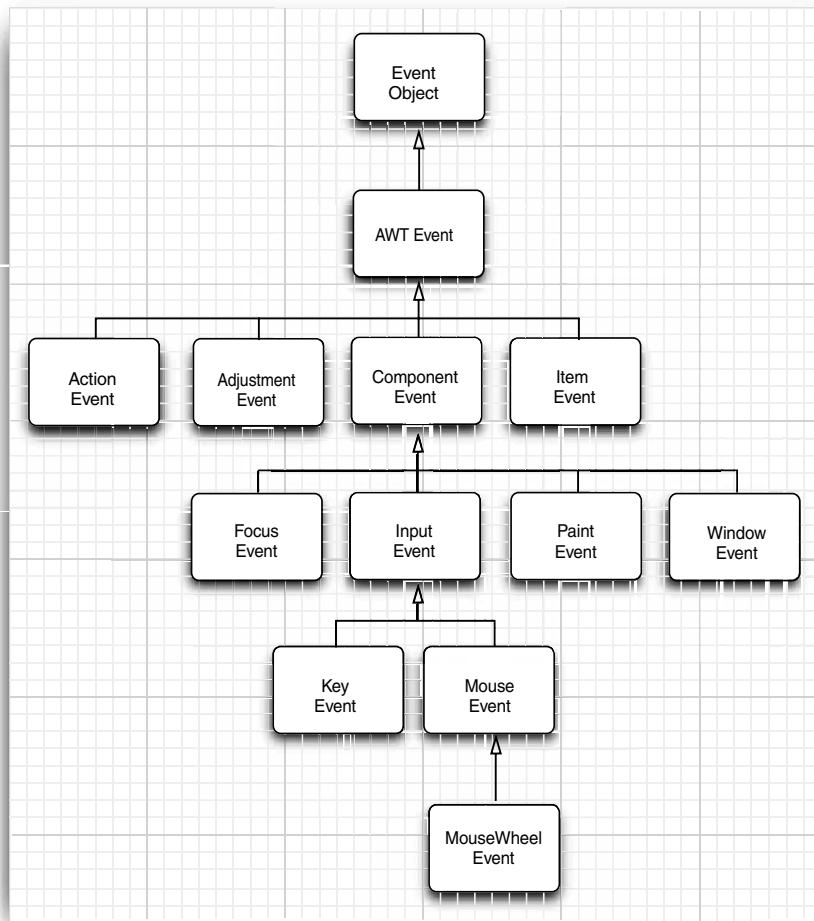
As we briefly mentioned earlier, event handling in Java is object oriented, with all events descending from the `EventObject` class in the `java.util` package. (The common superclass is not called `Event` because that is the name of the event class in the old event model. Although the old model is now deprecated, its classes are still a part of the Java library.)

The `EventObject` class has a subclass `AWTEvent`, which is the parent of all AWT event classes. Figure 8–8 shows the inheritance diagram of the AWT events.

Some of the Swing components generate event objects of yet more event types; these directly extend `EventObject`, not `AWTEvent`.

The event objects encapsulate information about the event that the event source communicates to its listeners. When necessary, you can then analyze the event objects that were passed to the listener object, as we did in the button example with the `getSource` and `getActionCommand` methods.

Some of the AWT event classes are of no practical use for the Java programmer. For example, the AWT inserts `PaintEvent` objects into the event queue, but these objects are not delivered to listeners. Java programmers don't listen to paint events; they override the `paintComponent` method to control repainting. The AWT also generates a number of events that are needed only by system programmers, to provide input systems for ideographic languages, automated testing robots, and so on. We do not discuss these specialized event types.



**Figure 8-8** Inheritance diagram of AWT event classes

### **Semantic and Low-Level Events**

The AWT makes a useful distinction between *low-level* and *semantic* events. A semantic event is one that expresses what the user is doing, such as “clicking that button”; hence, an `ActionEvent` is a semantic event. Low-level events are those events that make this possible. In the case of a button click, this is a mouse down, a series of mouse moves, and a mouse up (but only if the mouse up is inside the button area). Or it might be a keystroke, which happens if the user selects the button with the `TAB` key and then activates it with the space bar. Similarly, adjusting a scrollbar is a semantic event, but dragging the mouse is a low-level event.

Here are the most commonly used semantic event classes in the `java.awt.event` package:

- `ActionEvent` (for a button click, a menu selection, selecting a list item, or `ENTER` typed in a text field)
- `AdjustmentEvent` (the user adjusted a scrollbar)
- `ItemEvent` (the user made a selection from a set of checkbox or list items)

Five low-level event classes are commonly used:

- `KeyEvent` (a key was pressed or released)
- `MouseEvent` (the mouse button was pressed, released, moved, or dragged)
- `MouseWheelEvent` (the mouse wheel was rotated)
- `FocusEvent` (a component got focus or lost focus)
- `WindowEvent` (the window state changed)

The following interfaces listen to these events:

<code>ActionListener</code>	<code>MouseMotionListener</code>
<code>AdjustmentListener</code>	<code>MouseWheelListener</code>
<code>FocusListener</code>	<code>WindowListener</code>
<code>ItemListener</code>	<code>WindowFocusListener</code>
<code>KeyListener</code>	<code>WindowStateListener</code>
<code>MouseListener</code>	

Several of the AWT listener interfaces, namely, those that have more than one method, come with a companion adapter class that implements all the methods in the interface to do nothing. (The other interfaces have only a single method each, so there is no benefit in having adapter classes for these interfaces.) Here are the commonly used adapter classes:

<code>FocusAdapter</code>	<code>MouseMotionAdapter</code>
<code>KeyAdapter</code>	<code>WindowAdapter</code>
<code>MouseAdapter</code>	

Table 8–4 shows the most important AWT listener interfaces, events, and event sources. The `javax.swing.event` package contains additional events that are specific to Swing components. We cover some of them in the next chapter.

**Table 8–4 Event Handling Summary**

Interface	Methods	Parameter/ Accessors	Events Generated By
<code>ActionListener</code>	<code>actionPerformed</code>	<code>ActionEvent</code> • <code>getActionCommand</code> • <code>getModifiers</code>	<code>AbstractButton</code> <code>JComboBox</code> <code>JTextField</code> <code>Timer</code>
<code>AdjustmentListener</code>	<code>adjustmentValueChanged</code>	<code>AdjustmentEvent</code> • <code>getAdjustable</code> • <code>getAdjustmentType</code> • <code>getValue</code>	<code>JScrollbar</code>

**Table 8-4 Event Handling Summary (continued)**

<b>Interface</b>	<b>Methods</b>	<b>Parameter/ Accessors</b>	<b>Events Generated By</b>
ItemListener	itemStateChanged	ItemEvent • getItem • getItemSelectable • getStateChange	AbstractButton JComboBox
FocusListener	focusGained focusLost	FocusEvent • isTemporary	Component
KeyListener	keyPressed keyReleased keyTyped	KeyEvent • getKeyChar • getKeyCode • getKeyModifiersText • getKeyText • isActionKey	Component
MouseListener	mousePressed mouseReleased mouseEntered mouseExited mouseClicked	MouseEvent • getClickCount • getX • getY • getPoint • translatePoint	Component
MouseMotionListener	mouseDragged mouseMoved	MouseEvent	Component
MouseWheelListener	mouseWheelMoved	MouseWheelEvent • getWheelRotation • getScrollAmount	Component
WindowListener	windowClosing windowOpened windowIconified windowDeiconified windowClosed windowActivated windowDeactivated	WindowEvent • getWindow	Window
WindowFocusListener	windowGainedFocus windowLostFocus	WindowEvent • getOppositeWindow	Window
WindowStateListener	windowStateChanged	WindowEvent • getOldState • getNewState	Window

This concludes our discussion of AWT event handling. The next chapter shows you how to put together the most common of the components that Swing offers, along with a detailed coverage of the events they generate.



# *Chapter*

# 9

## USER INTERFACE COMPONENTS WITH SWING

- ▼ SWING AND THE MODEL-VIEW-CONTROLLER DESIGN PATTERN
- ▼ INTRODUCTION TO LAYOUT MANAGEMENT
- ▼ TEXT INPUT
- ▼ CHOICE COMPONENTS
- ▼ MENUS
- ▼ SOPHISTICATED LAYOUT MANAGEMENT
- ▼ DIALOG BOXES

**T**he last chapter was primarily designed to show you how to use the event model in Java. In the process you took the first steps toward learning how to build a graphical user interface. This chapter shows you the most important tools you'll need to build more full-featured GUIs.

We start out with a tour of the architectural underpinnings of Swing. Knowing what goes on “under the hood” is important in understanding how to use some of the more advanced components effectively. We then show you how to use the most common user interface components in Swing such as text fields, radio buttons, and menus. Next, you

will learn how to use the nifty layout manager features of Java to arrange these components in a window, regardless of the look and feel of a particular user interface. Finally, you'll see how to implement dialog boxes in Swing.

This chapter covers basic Swing components such as text components, buttons, and sliders. These are the essential user interface components that you will need most frequently. We cover advanced Swing components in Volume II.

### Swing and the Model-View-Controller Design Pattern

As promised, we start this chapter with a section describing the architecture of Swing components. We first discuss the concept of *design patterns* and then look at the “model-view-controller” pattern that has greatly influenced the design of the Swing framework.

#### Design Patterns

When solving a problem, you don't usually figure out a solution from first principles. Instead, you are likely to be guided by past experience, or you may ask other experts for advice on what has worked for them. Design patterns are a method for presenting this expertise in a structured way.

In recent years, software engineers have begun to assemble catalogs of such patterns.

The pioneers in this area were inspired by the architectural design patterns of the architect Christopher Alexander. In his book, *The Timeless Way of Building* (Oxford University Press, 1979), Alexander gives a catalog of patterns for designing public and private living spaces. Here is a typical example:

#### Window Place

Everybody loves window seats, bay windows, and big windows with low sills and comfortable chairs drawn up to them . . . A room which does not have a place like this seldom allows you to feel comfortable or perfectly at ease . . .

If the room contains no window which is a “place,” a person in the room will be torn between two forces: (1) He wants to sit down and be comfortable, and (2) he is drawn toward the light.

Obviously, if the comfortable places—those places in the room where you most want to sit—are away from the windows, there is no way of overcoming this conflict . . .

Therefore: In every room where you spend any length of time during the day, make at least one window into a “window place.”

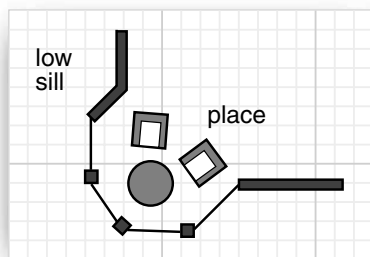


Figure 9-1 A window place

Each pattern in Alexander's catalog, as well as those in the catalogs of software patterns, follows a particular format. The pattern first describes a context, a situation that gives rise to a design problem. Then, the problem is explained, usually as a set of conflicting forces. Finally, the solution shows a configuration that balances these forces.

In the "window place" pattern, the context is a room in which you spend any length of time during the day. The conflicting forces are that you want to sit down and be comfortable and that you are drawn to the light. The solution is to make a "window place."

In the "model-view-controller" pattern, which we will describe in the next section, the context is a user interface system that presents information and receives user input.

There are several forces. There may be multiple visual representations of the same data that need to be updated together. The visual representation may change, for example, to accommodate various look-and-feel standards. The interaction mechanisms may change, for example, to support voice commands. The solution is to distribute responsibilities into three separate interacting components: the model, view, and controller.

The model-view-controller pattern is not the only pattern used in the design of AWT and Swing. Here are several additional examples:

- Containers and components are examples of the "composite" pattern.
- The scroll pane is a "decorator."
- Layout managers follow the "strategy" pattern.

One important aspect of design patterns is that they become part of the culture. Programmers all over the world know what you mean when you talk about the model-view-controller pattern or the "decorator" pattern. Thus, patterns become an efficient way of talking about design problems.

You will find a formal description of numerous useful software patterns in the seminal book of the pattern movement, *Design Patterns—Elements of Reusable Object-Oriented Software*, by Erich Gamma et al. (Addison-Wesley, 1995). We also highly recommend the excellent book *A System of Patterns* by Frank Buschmann et al. (John Wiley & Sons, 1996), which we find less seminal and more approachable.

### **The Model-View-Controller Pattern**

Let's step back for a minute and think about the pieces that make up a user interface component such as a button, a checkbox, a text field, or a sophisticated tree control.

Every component has three characteristics:

- Its *content*, such as the state of a button (pushed in or not), or the text in a text field
- Its *visual appearance* (color, size, and so on)
- Its *behavior* (reaction to events)

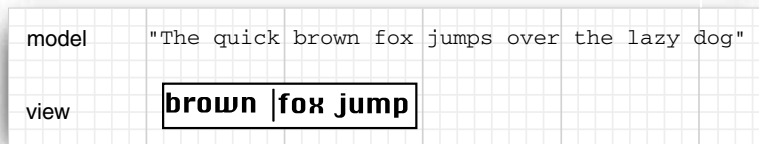
Even a seemingly simple component such as a button exhibits some moderately complex interaction among these characteristics. Obviously, the visual appearance of a button depends on the look and feel. A Metal button looks different from a Windows button or a Motif button. In addition, the appearance depends on the button state: When a button is pushed in, it needs to be redrawn to look different. The state depends on the events that the button receives. When the user depresses the mouse inside the button, the button is pushed in.

Of course, when you use a button in your programs, you simply consider it as a *button*, and you don't think too much about the inner workings and characteristics. That, after all, is the job of the programmer who implemented the button. However, those programmers who implement buttons are motivated to think a little harder about them. After all, they have to implement buttons, and all other user interface components, so that they work well no matter what look and feel is installed.

To do this, the Swing designers turned to a well-known design pattern: the *model-view-controller* pattern. This pattern, like many other design patterns, goes back to one of the principles of object-oriented design that we mentioned way back in Chapter 5: Don't make one object responsible for too much. Don't have a single button class do everything. Instead, have the look and feel of the component associated with one object and store the content in *another* object. The model-view-controller (MVC) design pattern teaches how to accomplish this. Implement three separate classes:

- The *model*, which stores the content
- The *view*, which displays the content
- The *controller*, which handles user input

The pattern specifies precisely how these three objects interact. The model stores the content and has *no user interface*. For a button, the content is pretty trivial—just a small set of flags that tells whether the button is currently pushed in or out, whether it is active or inactive, and so on. For a text field, the content is a bit more interesting. It is a string object that holds the current text. This is *not the same* as the view of the content—if the content is larger than the text field, the user sees only a portion of the text displayed (see Figure 9-2).



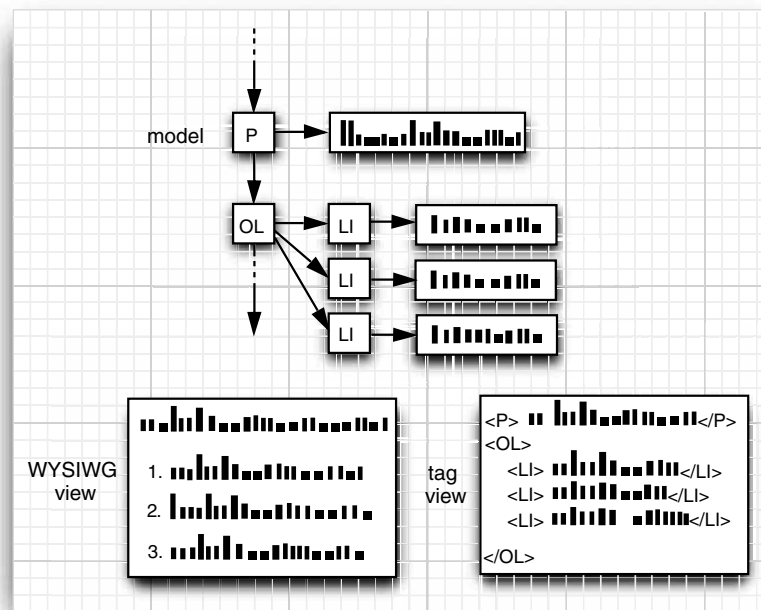
**Figure 9-2 Model and view of a text field**

The model must implement methods to change the content and to discover what the content is. For example, a text model has methods to add or remove characters in the current text and to return the current text as a string. Again, keep in mind that the model is completely nonvisual. It is the job of a view to draw the data that is stored in the model.



**NOTE:** The term "model" is perhaps unfortunate because we often think of a model as a representation of an abstract concept. Car and airplane designers build models to simulate real cars and planes. But that analogy really leads you astray when thinking about the model-view-controller pattern. In the design pattern, the model stores the complete content, and the view gives a (complete or incomplete) visual representation of the content. A better analogy might be the model who poses for an artist. It is up to the artist to look at the model and create a view. Depending on the artist, that view might be a formal portrait, an impressionist painting, or a cubist drawing that shows the limbs in strange contortions.

One of the advantages of the model-view-controller pattern is that a model can have multiple views, each showing a different part or aspect of the full content. For example, an HTML editor can offer two *simultaneous* views of the same content: a WYSIWYG view and a “raw tag” view (see Figure 9–3). When the model is updated through the controller of one of the views, it tells both attached views about the change. When the views are notified, they refresh themselves automatically. Of course, for a simple user interface component such as a button, you won’t have multiple views of the same model.



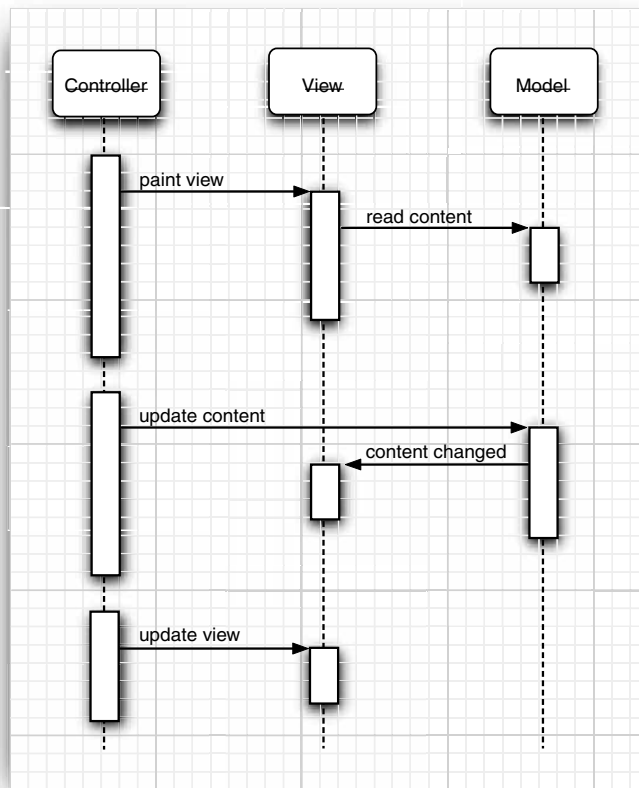
**Figure 9–3** Two separate views of the same model

The controller handles the user-input events such as mouse clicks and keystrokes. It then decides whether to translate these events into changes in the model or the view. For example, if the user presses a character key in a text box, the controller calls the “insert character” command of the model. The model then tells the view to update itself. The view never knows why the text changed. But if the user presses a cursor key, then the controller may tell the view to scroll. Scrolling the view has no effect on the underlying text, so the model never knows that this event happened.

Figure 9–4 shows the interactions among model, view, and controller objects.

As a programmer using Swing components, you generally don’t need to think about the model-view-controller architecture. Each user interface component has a wrapper class (such as `JButton` or `JTextField`) that stores the model and the view. When you want to inquire about the content (for example, the text in a text field), the wrapper class asks the model

and returns the answer to you. When you want to change the view (for example, move the caret position in a text field), the wrapper class forwards that request to the view. However, occasionally the wrapper class doesn't work hard enough on forwarding commands. Then, you have to ask it to retrieve the model and work directly with the model. (You don't have to work directly with the view—that is the job of the look-and-feel code.)



**Figure 9-4 Interactions among model, view, and controller objects**

Besides being “the right thing to do,” the model-view-controller pattern was attractive for the Swing designers because it allowed them to implement pluggable look and feel. The model of a button or text field is independent of the look and feel. But of course the visual representation is completely dependent on the user interface design of a particular look and feel. The controller can vary as well. For example, in a voice-controlled device, the controller must cope with an entirely different set of events than in a standard computer with a keyboard and a mouse. By separating out the underlying model

from the user interface, the Swing designers can reuse the code for the models and can even switch the look and feel in a running program.

Of course, patterns are only intended as guidance, not as religion. No pattern is applicable in all situations. For example, you may find it difficult to follow the “window places” pattern to rearrange your cubicle. Similarly, the Swing designers found that the harsh reality of pluggable look-and-feel implementation does not always allow for a neat realization of the model-view-controller pattern. Models are easy to separate, and each user interface component has a model class. But the responsibilities of the view and controller are not always clearly separated and are distributed over a number of different classes. Of course, as a user of these classes, you won’t be concerned about this. In fact, as we pointed out before, you often won’t have to worry about the models either—you can just use the component wrapper classes.

### ***A Model-View-Controller Analysis of Swing Buttons***

You already learned how to use buttons in the previous chapter, without having to worry about the controller, model, or view for them. Still, buttons are about the simplest user interface elements, so they are a good place to become comfortable with the model-view-controller pattern. You will encounter similar kinds of classes and interfaces for the more sophisticated Swing components.

For most components, the model class implements an interface whose name ends in `Model`; thus the interface called `ButtonModel`. Classes implementing that interface can define the state of the various kinds of buttons. Actually, buttons aren’t all that complicated, and the Swing library contains a single class, called `DefaultButtonModel`, that implements this interface.

You can get a sense of what sort of data are maintained by a button model by looking at the properties of the `ButtonModel` interface—see Table 9–1.

**Table 9–1 Properties of the `ButtonModel` Interface**

Property Name	Value
<code>actionCommand</code>	The action command string associated with this button
<code>mnemonic</code>	The keyboard mnemonic for this button
<code>armed</code>	true if the button was pressed and the mouse is still over the button
<code>enabled</code>	true if the button is selectable
<code>pressed</code>	true if the button was pressed but the mouse button hasn’t yet been released
<code>rollover</code>	true if the mouse is over the button
<code>selected</code>	true if the button has been toggled on (used for checkboxes and radio buttons)

Each `JButton` object stores a button model object, which you can retrieve.

```
JButton button = new JButton("Blue");  
ButtonModel model = button.getModel();
```

In practice, you won't care—the minutiae of the button state are only of interest to the view that draws it. And the important information—such as whether a button is enabled—is available from the `JButton` class. (The `JButton` then asks its model, of course, to retrieve that information.)

Have another look at the `ButtonModel` interface to see what *isn't* there. The model does *not* store the button label or icon. There is no way to find out what's on the face of a button just by looking at its model. (Actually, as you will see in "Radio Buttons" on page 388, purity of design is the source of some grief for the programmer.)

It is also worth noting that the *same* model (namely, `DefaultButtonModel`) is used for push buttons, radio buttons, checkboxes, and even menu items. Of course, each of these button types has different views and controllers. When using the Metal look and feel, the `JButton` uses a class called `BasicButtonUI` for the view and a class called `ButtonUIListener` as controller. In general, each Swing component has an associated view object that ends in `UI`. But not all Swing components have dedicated controller objects.

So, having read this short introduction to what is going on under the hood in a `JButton`, you may be wondering: Just what is a `JButton` really? It is simply a wrapper class inheriting from `JComponent` that holds the `DefaultButtonModel` object, some view data (such as the button label and icons), and a `BasicButtonUI` object that is responsible for the button view.

### Introduction to Layout Management

Before we go on to discussing individual Swing components, such as text fields and radio buttons, we briefly cover how to arrange these components inside a frame. Unlike Visual Basic, the JDK has no form designer. You need to write code to position (lay out) the user interface components where you want them to be.

Of course, if you have a Java-enabled development environment, it will probably have a layout tool that automates some or all of these tasks. Nevertheless, it is important to know exactly what goes on "under the hood" because even the best of these tools will usually require hand-tweaking.

Let's start by reviewing the program from Chapter 8 that used buttons to change the background color of a frame (see Figure 9-5).



Figure 9-5 A panel with three buttons



The buttons are contained in a `JPanel` object and are managed by the *flow layout manager*, the default layout manager for a panel. Figure 9-6 shows what happens when you add more buttons to the panel. As you can see, a new row is started when there is no more room.



**Figure 9-6** A panel with six buttons managed by a flow layout

Moreover, the buttons stay centered in the panel, even when the user resizes the frame (see Figure 9-7).



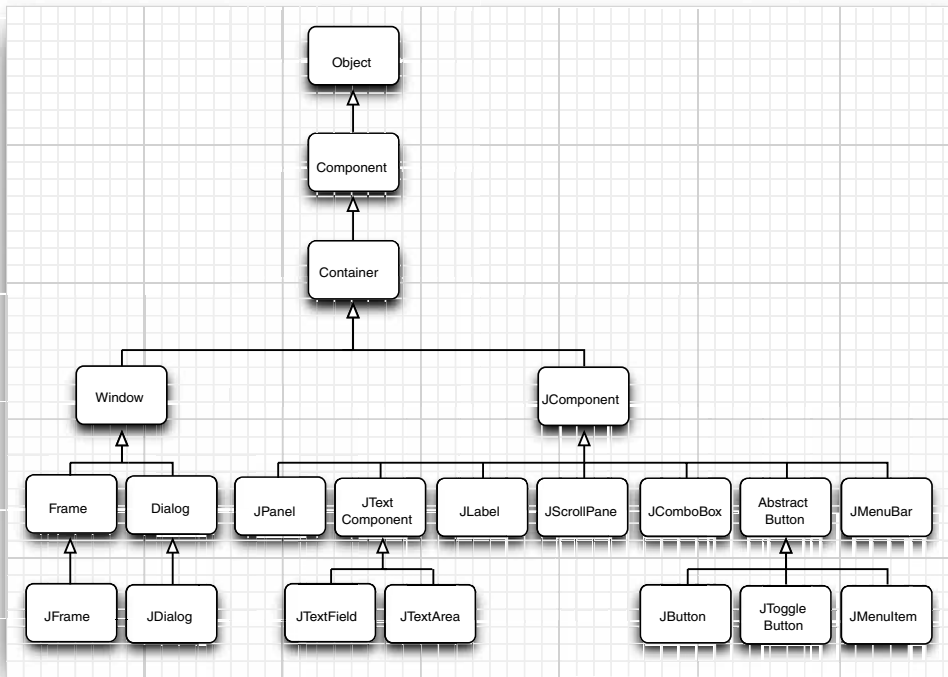
**Figure 9-7** Changing the panel size rearranges the buttons automatically

In general, *components* are placed inside *containers*, and a *layout manager* determines the positions and sizes of the components in the container.

Buttons, text fields, and other user interface elements extend the class `Component`. Components can be placed inside containers such as panels. Because containers can themselves be put inside other containers, the class `Container` extends `Component`. Figure 9-8 shows the inheritance hierarchy for `Component`.



**NOTE:** Unfortunately, the inheritance hierarchy is somewhat unclear in two respects. First, top-level windows such as `JFrame` are subclasses of `Container` and hence `Component`, but they cannot be placed inside other containers. Moreover, `JComponent` is a subclass of `Container` not `Component`, and therefore one can add other components into a `JButton`. (However, those components would not be displayed.)



**Figure 9-8 Inheritance hierarchy for the Component class**

Each container has a default layout manager, but you can always set your own. For example, the statement

```
panel.setLayout(new GridLayout(4, 4));
```

uses the `GridLayout` class to lay out the components in the panel. You add components to the container. The `add` method of the container passes the component and any placement directions to the layout manager.

**API** `java.awt.Container` 1.0

- `void setLayout(LayoutManager m)`  
sets the layout manager for this container.
- `Component add(Component c)`
- `Component add(Component c, Object constraints)` 1.1  
adds a component to this container and returns the component reference.

*Parameters:*

<code>c</code>	The component to add
<code>constraints</code>	An identifier understood by the layout manager

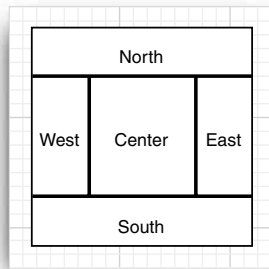
**API** java.awt.FlowLayout 1.0

- FlowLayout()
- FlowLayout(int align)
- FlowLayout(int align, int hgap, int vgap)  
constructs a new FlowLayout.

<i>Parameters:</i>	align	One of LEFT, CENTER, or RIGHT
	hgap	The horizontal gap to use in pixels (negative values force an overlap)
	vgap	The vertical gap to use in pixels (negative values force an overlap)

**Border Layout**

The *border layout manager* is the default layout manager of the content pane of every JFrame. Unlike the flow layout manager, which completely controls the position of each component, the border layout manager lets you choose where you want to place each component. You can choose to place the component in the center, north, south, east, or west of the content pane (see Figure 9–9).

**Figure 9–9** Border layout

For example:

```
frame.add(component, BorderLayout.SOUTH);
```

The edge components are laid out first, and the remaining available space is occupied by the center. When the container is resized, the dimensions of the edge components are unchanged, but the center component changes its size. You add components by specifying a constant CENTER, NORTH, SOUTH, EAST, or WEST of the BorderLayout class. Not all of the positions need to be occupied. If you don't supply any value, CENTER is assumed.



**NOTE:** The BorderLayout constants are defined as strings. For example, BorderLayout.SOUTH is defined as the string "South". Many programmers prefer to use the strings directly because they are shorter, for example, frame.add(component, "South"). However, if you accidentally misspell a string, the compiler won't catch that error.

Unlike the flow layout, the border layout grows all components to fill the available space. (The flow layout leaves each component at its preferred size.) This is a problem when you add a button:

```
frame.add(yellowButton, BorderLayout.SOUTH); // don't
```

Figure 9–10 shows what happens when you use the preceding code fragment. The button has grown to fill the entire southern region of the frame. And, if you were to add another button to the southern region, it would just displace the first button.



**Figure 9–10** A single button managed by a border layout

You solve this problem by using additional panels. For example, look at Figure 9–11. The three buttons at the bottom of the screen are all contained in a panel. The panel is put into the southern region of the content pane.



**Figure 9–11** Panel placed at the southern region of the frame

To achieve this configuration, first create a new `JPanel` object, then add the individual buttons to the panel. The default layout manager for a panel is a `FlowLayout`, which is a good choice for this situation. You add the individual buttons to the panel, using the `add` method you have seen before. The position and size of the buttons is under the control of the `FlowLayout` manager. This means the buttons stay centered within the panel, and they do not expand to fill the entire panel area. Finally, you add the panel to the content pane of the frame.

```
JPanel panel = new JPanel();
panel.add(yellowButton);
panel.add(blueButton);
panel.add(redButton);
frame.add(panel, BorderLayout.SOUTH);
```

The border layout expands the size of the panel to fill the entire southern region.

**API** `java.awt.BorderLayout` 1.0

- `BorderLayout()`
- `BorderLayout(int hgap, int vgap)`  
constructs a new `BorderLayout`.

*Parameters:*

<code>hgap</code>	The horizontal gap to use in pixels (negative values force an overlap)
<code>vgap</code>	The vertical gap to use in pixels (negative values force an overlap)

**Grid Layout**

The grid layout arranges all components in rows and columns like a spreadsheet. All components are given the same size. The calculator program in Figure 9–12 uses a grid layout to arrange the calculator buttons. When you resize the window, the buttons grow and shrink, but all buttons have identical sizes.

**Figure 9–12** A calculator

In the constructor of the grid layout object, you specify how many rows and columns you need.

```
panel.setLayout(new GridLayout(5, 4));
```

You add the components, starting with the first entry in the first row, then the second entry in the first row, and so on.

```
panel.add(new JButton("1"));
panel.add(new JButton("2"));
```

Listing 9–1 is the source listing for the calculator program. This is a regular calculator, not the “reverse Polish” variety that is so oddly popular in Java tutorials. In this program, we call the `pack` method after adding the component to the frame. This method uses the preferred sizes of all components to compute the width and height of the frame.

Of course, few applications have as rigid a layout as the face of a calculator. In practice, small grids (usually with just one row or one column) can be useful to organize partial areas of a window. For example, if you want to have a row of buttons with identical size, then you can put the buttons inside a panel that is governed by a grid layout with a single row.

**Listing 9-1** Calculator.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.33 2007-06-12
7.  * @author Cay Horstmann
8.  */
9. public class Calculator
10. {
11.     public static void main(String[] args)
12.     {
13.         EventQueue.invokeLater(new Runnable()
14.         {
15.             public void run()
16.             {
17.                 CalculatorFrame frame = new CalculatorFrame();
18.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.                 frame.setVisible(true);
20.             }
21.         });
22.     }
23. }
24.
25. /**
26.  * A frame with a calculator panel.
27.  */
28. class CalculatorFrame extends JFrame
29. {
30.     public CalculatorFrame()
31.     {
32.         setTitle("Calculator");
33.         CalculatorPanel panel = new CalculatorPanel();
34.         add(panel);
35.         pack();
36.     }
37. }
38.
39. /**
40.  * A panel with calculator buttons and a result display.
41.  */
42. class CalculatorPanel extends JPanel
43. {
44.     public CalculatorPanel()
45.     {
46.         setLayout(new BorderLayout());
47.
48.         result = 0;
49.         lastCommand = "=";
50.         start = true;
```

**Listing 9-1** Calculator.java (continued)

```
51.
52.     // add the display
53.
54.     display = new JButton("0");
55.     display.setEnabled(false);
56.     add(display, BorderLayout.NORTH);
57.
58.     ActionListener insert = new InsertAction();
59.     ActionListener command = new CommandAction();
60.
61.     // add the buttons in a 4 x 4 grid
62.
63.     panel = new JPanel();
64.     panel.setLayout(new GridLayout(4, 4));
65.
66.     addButton("7", insert);
67.     addButton("8", insert);
68.     addButton("9", insert);
69.     addButton("/", command);
70.
71.     addButton("4", insert);
72.     addButton("5", insert);
73.     addButton("6", insert);
74.     addButton("*", command);
75.
76.     addButton("1", insert);
77.     addButton("2", insert);
78.     addButton("3", insert);
79.     addButton("-", command);
80.
81.     addButton("0", insert);
82.     addButton(".", insert);
83.     addButton("=", command);
84.     addButton("+", command);
85.
86.     add(panel, BorderLayout.CENTER);
87. }
88.
89. /**
90.  * Adds a button to the center panel.
91.  * @param label the button label
92.  * @param listener the button listener
93.  */
94. private void addButton(String label, ActionListener listener)
95. {
96.     JButton button = new JButton(label);
97.     button.addActionListener(listener);
98.     panel.add(button);
99. }
100.
```

**Listing 9-1** Calculator.java (continued)

```
101. /**
102.  * This action inserts the button action string to the end of the display text.
103.  */
104. private class InsertAction implements ActionListener
105. {
106.     public void actionPerformed(ActionEvent event)
107.     {
108.         String input = event.getActionCommand();
109.         if (start)
110.         {
111.             display.setText("");
112.             start = false;
113.         }
114.         display.setText(display.getText() + input);
115.     }
116. }
117.
118. /**
119.  * This action executes the command that the button action string denotes.
120.  */
121. private class CommandAction implements ActionListener
122. {
123.     public void actionPerformed(ActionEvent event)
124.     {
125.         String command = event.getActionCommand();
126.
127.         if (start)
128.         {
129.             if (command.equals("-"))
130.             {
131.                 display.setText(command);
132.                 start = false;
133.             }
134.             else lastCommand = command;
135.         }
136.         else
137.         {
138.             calculate(Double.parseDouble(display.getText()));
139.             lastCommand = command;
140.             start = true;
141.         }
142.     }
143. }
144.
145. /**
146.  * Carries out the pending calculation.
147.  * @param x the value to be accumulated with the prior result.
148.  */
149. public void calculate(double x)
150. {
```



**Listing 9-1** Calculator.java (continued)

```

151.     if (lastCommand.equals("+")) result += x;
152.     else if (lastCommand.equals("-")) result -= x;
153.     else if (lastCommand.equals("*")) result *= x;
154.     else if (lastCommand.equals("/")) result /= x;
155.     else if (lastCommand.equals("=")) result = x;
156.     display.setText("" + result);
157. }
158.
159. private JButton display;
160. private JPanel panel;
161. private double result;
162. private String lastCommand;
163. private boolean start;
164. }

```

**API** java.awt.GridLayout 1.0

- GridLayout(int rows, int columns)
- GridLayout(int rows, int columns, int hgap, int vgap)  
constructs a new GridLayout. One of rows and columns (but not both) may be zero, denoting an arbitrary number of components per row or column.

<i>Parameters:</i>	rows	The number of rows in the grid
	columns	The number of columns in the grid
	hgap	The horizontal gap to use in pixels (negative values force an overlap)
	vgap	The vertical gap to use in pixels (negative values force an overlap)

**API** java.awt.Window 1.0

- void pack()  
resizes this window, taking into account the preferred sizes of its components.

**Text Input**

We are finally ready to start introducing the Swing user interface components. We start with components that let a user input and edit text. You can use the `JTextField` and `JTextArea` components for gathering text input. A text field can accept only one line of text; a text area can accept multiple lines of text. A `JPasswordField` accepts one line of text without showing the contents.

All three of these classes inherit from a class called `JTextComponent`. You will not be able to construct a `JTextComponent` yourself because it is an abstract class. On the other hand, as is so often the case in Java, when you go searching through the API documentation, you may find that the methods you are looking for are actually in the parent class `JTextComponent` rather than in the derived class. For example, the methods that get or set the text in a text field or text area are actually methods in `JTextComponent`.

**API** javax.swing.text.JTextComponent 1.2

- String getText()
- void setText(String text)  
gets or sets the text of this text component.
- boolean isEditable()
- void setEditable(boolean b)  
gets or sets the editable property that determines whether the user can edit the content of this text component.

**Text Fields**

The usual way to add a text field to a window is to add it to a panel or other container—just as you would a button:

```
JPanel panel = new JPanel();
JTextField textField = new JTextField("Default input", 20);
panel.add(textField);
```

This code adds a text field and initializes the text field by placing the string "Default input" inside it. The second parameter of this constructor sets the width. In this case, the width is 20 "columns." Unfortunately, a column is a rather imprecise measurement. One column is the expected width of one character in the font you are using for the text. The idea is that if you expect the inputs to be *n* characters or less, you are supposed to specify *n* as the column width. In practice, this measurement doesn't work out too well, and you should add 1 or 2 to the maximum input length to be on the safe side. Also, keep in mind that the number of columns is only a hint to the AWT that gives the *preferred* size. If the layout manager needs to grow or shrink the text field, it can adjust its size. The column width that you set in the `JTextField` constructor is not an upper limit on the number of characters the user can enter. The user can still type in longer strings, but the input scrolls when the text exceeds the length of the field. Users tend to find scrolling text fields irritating, so you should size the fields generously. If you need to reset the number of columns at runtime, you can do that with the `setColumns` method.



**TIP:** After changing the size of a text box with the `setColumns` method, call the `revalidate` method of the surrounding container.

```
textField.setColumns(10);
panel.revalidate();
```

The `revalidate` method recomputes the size and layout of all components in a container. After you use the `revalidate` method, the layout manager resizes the container, and the changed size of the text field will be visible.

The `revalidate` method belongs to the `JComponent` class. It doesn't immediately resize the component but merely marks it for resizing. This approach avoids repetitive calculations if multiple components request to be resized. However, if you want to recompute all components inside a `JFrame`, you have to call the `validate` method—`JFrame` doesn't extend `JComponent`.

In general, you want to let the user add text (or edit the existing text) in a text field. Quite often these text fields start out blank. To make a blank text field, just leave out the string as a parameter for the `JTextField` constructor:

```
JTextField textField = new JTextField(20);
```

You can change the content of the text field at any time by using the `setText` method from the `JTextComponent` parent class mentioned in the previous section. For example:

```
textField.setText("Hello!");
```

And, as was also mentioned in the previous section, you can find out what the user typed by calling the `getText` method. This method returns the exact text that the user typed. To trim any extraneous leading and trailing spaces from the data in a text field, apply the `trim` method to the return value of `getText`:

```
String text = textField.getText().trim();
```

To change the font in which the user text appears, use the `setFont` method.

**API** `javax.swing.JTextField` 1.2

- `JTextField(int cols)`  
constructs an empty `JTextField` with a specified number of columns.
- `JTextField(String text, int cols)`  
constructs a new `JTextField` with an initial string and the specified number of columns.
- `int getColumns()`
- `void setColumns(int cols)`  
gets or sets the number of columns that this text field should use.

**API** `javax.swing.JComponent` 1.2

- `void revalidate()`  
causes the position and size of a component to be recomputed.
- `void setFont(Font f)`  
sets the font of this component.

**API** `java.awt.Component` 1.0

- `void validate()`  
recomputes the position and size of a component. If the component is a container, the positions and sizes of its components are recomputed.
- `Font getFont()`  
gets the font of this component.

**Labels and Labeling Components**

Labels are components that hold text. They have no decorations (for example, no boundaries). They also do not react to user input. You can use a label to identify components. For example, unlike buttons, text fields have no label to identify them. To label a component that does not itself come with an identifier:

1. Construct a `JLabel` component with the correct text.
2. Place it close enough to the component you want to identify so that the user can see that the label identifies the correct component.

The constructor for a `JLabel` lets you specify the initial text or icon, and optionally, the alignment of the content. You use constants from the `SwingConstants` interface to specify alignment. That interface defines a number of useful constants such as `LEFT`, `RIGHT`, `CENTER`, `NORTH`, `EAST`, and so on. The `JLabel` class is one of several Swing classes that implement this interface. Therefore, you can specify a right-aligned label either as

```
JLabel label = new JLabel("User name: ", SwingConstants.RIGHT);
```

or

```
JLabel label = new JLabel("User name: ", JLabel.RIGHT);
```

The `setText` and `setIcon` methods let you set the text and icon of the label at runtime.



**TIP:** Beginning with Java SE 1.3, you can use both plain and HTML text in buttons, labels, and menu items. We don't recommend HTML in buttons—it interferes with the look and feel. But HTML in labels can be very effective. Simply surround the label string with `<html>...</html>`, like this:

```
label = new JLabel("<html><b>Required</b> entry:</html>");
```

Fair warning—the first component with an HTML label takes some time to be displayed because the rather complex HTML rendering code must be loaded.

Labels can be positioned inside a container like any other component. This means you can use the techniques you have seen before to place labels where you need them.

#### API `javax.swing.JLabel` 1.2

- `JLabel(String text)`
- `JLabel(Icon icon)`
- `JLabel(String text, int align)`
- `JLabel(String text, Icon icon, int align)`  
constructs a label.

<i>Parameters:</i>	<code>text</code>	The text in the label
	<code>icon</code>	The icon in the label
	<code>align</code>	One of the <code>SwingConstants</code> constants <code>LEFT</code> (default), <code>CENTER</code> , or <code>RIGHT</code>

- `String getText()`
- `void setText(String text)`  
gets or sets the text of this label.
- `Icon getIcon()`
- `void setIcon(Icon icon)`  
gets or sets the icon of this label.

### Password Fields

Password fields are a special kind of text field. To avoid nosy bystanders being able to glance at a password, the characters that the user entered are not actually displayed. Instead, each typed character is represented by an *echo character*, typically an asterisk (\*). Swing supplies a `JPasswordField` class that implements such a text field.

The password field is another example of the power of the model-view-controller architecture pattern. The password field uses the same model to store the data as a regular text field, but its view has been changed to display all characters as echo characters.

#### API `javax.swing.JPasswordField` 1.2

- `JPasswordField(String text, int columns)`  
constructs a new password field.
- `void setEchoChar(char echo)`  
sets the echo character for this password field. This is advisory; a particular look and feel may insist on its own choice of echo character. A value of 0 resets the echo character to the default.
- `char[] getPassword()`  
returns the text contained in this password field. For stronger security, you should overwrite the content of the returned array after use. (The password is not returned as a `String` because a string would stay in the virtual machine until it is garbage-collected.)

### Text Areas

Sometimes, you need to collect user input that is more than one line long. As mentioned earlier, you use the `JTextArea` component for this collection. When you place a text area component in your program, a user can enter any number of lines of text, using the ENTER key to separate them. Each line ends with a '\n'. Figure 9-13 shows a text area at work.



Figure 9-13 Text components

In the constructor for the `JTextArea` component, you specify the number of rows and columns for the text area. For example,

```
textArea = new JTextArea(8, 40); // 8 lines of 40 columns each
```

where the `columns` parameter works as before—and you still need to add a few more columns for safety’s sake. Also, as before, the user is not restricted to the number of rows and columns; the text simply scrolls when the user inputs too much. You can also use the `setColumns` method to change the number of columns, and the `setRows` method to change the number of rows. These numbers only indicate the preferred size—the layout manager can still grow or shrink the text area.

If there is more text than the text area can display, then the remaining text is simply clipped. You can avoid clipping long lines by turning on line wrapping:

```
textArea.setLineWrap(true); // long lines are wrapped
```

This wrapping is a visual effect only; the text in the document is not changed—no `'\n'` characters are inserted into the text.

### Scroll Panes

In Swing, a text area does not have scrollbars. If you want scrollbars, you have to insert the text area inside a *scroll pane*.

```
textArea = new JTextArea(8, 40);
JScrollPane scrollPane = new JScrollPane(textArea);
```

The scroll pane now manages the view of the text area. Scrollbars automatically appear if there is more text than the text area can display, and they vanish again if text is deleted and the remaining text fits inside the area. The scrolling is handled internally in the scroll pane—your program does not need to process scroll events.

This is a general mechanism that works for any component, not just text areas. To add scrollbars to a component, put them inside a scroll pane.

Listing 9–2 demonstrates the various text components. This program simply shows a text field, a password field, and a text area with scrollbars. The text field and password field are labeled. Click on “Insert” to insert the field contents into the text area.



**NOTE:** The `JTextArea` component displays plain text only, without special fonts or formatting. To display formatted text (such as HTML), you can use the `JEditorPane` class that is discussed in Volume II.

#### Listing 9–2 TextComponentTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.40 2007-04-27
7.  * @author Cay Horstmann
8.  */
```

**Listing 9-2** TextComponentTest.java (continued)

```
9. public class TextComponentTest
10. {
11.     public static void main(String[] args)
12.     {
13.         EventQueue.invokeLater(new Runnable()
14.             {
15.                 public void run()
16.                 {
17.                     TextComponentFrame frame = new TextComponentFrame();
18.                     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.                     frame.setVisible(true);
20.                 }
21.             });
22.     }
23. }
24.
25. /**
26.  * A frame with sample text components.
27.  */
28. class TextComponentFrame extends JFrame
29. {
30.     public TextComponentFrame()
31.     {
32.         setTitle("TextComponentTest");
33.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34.
35.         final JTextField textField = new JTextField();
36.         final JPasswordField passwordField = new JPasswordField();
37.
38.         JPanel northPanel = new JPanel();
39.         northPanel.setLayout(new GridLayout(2, 2));
40.         northPanel.add(new JLabel("User name: ", SwingConstants.RIGHT));
41.         northPanel.add(textField);
42.         northPanel.add(new JLabel("Password: ", SwingConstants.RIGHT));
43.         northPanel.add(passwordField);
44.
45.         add(northPanel, BorderLayout.NORTH);
46.
47.         final JTextArea textArea = new JTextArea(8, 40);
48.         JScrollPane scrollPane = new JScrollPane(textArea);
49.
50.         add(scrollPane, BorderLayout.CENTER);
51.
52.         // add button to append text into the text area
53.
54.         JPanel southPanel = new JPanel();
55.
56.         JButton insertButton = new JButton("Insert");
57.         southPanel.add(insertButton);
```

**Listing 9-2** TextComponentTest.java (continued)

```
58.     insertButton.addActionListener(new ActionListener()
59.     {
60.         public void actionPerformed(ActionEvent event)
61.         {
62.             textArea.append("User name: " + textField.getText() + " Password: "
63.                 + new String(passwordField.getPassword()) + "\n");
64.         }
65.     });
66.
67.     add(southPanel, BorderLayout.SOUTH);
68.
69.     // add a text area with scrollbars
70.
71. }
72.
73. public static final int DEFAULT_WIDTH = 300;
74. public static final int DEFAULT_HEIGHT = 300;
75. }
```

**API** javax.swing.JTextArea 1.2

- JTextArea()
- JTextArea(int rows, int cols)
- JTextArea(String text, int rows, int cols)  
constructs a new text area.
- void setColumns(int cols)  
tells the text area the preferred number of columns it should use.
- void setRows(int rows)  
tells the text area the preferred number of rows it should use.
- void append(String newText)  
appends the given text to the end of the text already in the text area.
- void setLineWrap(boolean wrap)  
turns line wrapping on or off.
- void setWrapStyleWord(boolean word)  
If word is true, then long lines are wrapped at word boundaries. If it is false, then long lines are broken without taking word boundaries into account.
- void setTabSize(int c)  
sets tab stops every c columns. Note that the tabs aren't converted to spaces but cause alignment with the next tab stop.

**API** javax.swing.JScrollPane 1.2

- JScrollPane(Component c)  
creates a scroll pane that displays the content of the specified component. Scrollbars are supplied when the component is larger than the view.



## Choice Components

You now know how to collect text input from users, but there are many occasions for which you would rather give users a finite set of choices than have them enter the data in a text component. Using a set of buttons or a list of items tells your users what choices they have. (It also saves you the trouble of error checking.) In this section, you learn how to program checkboxes, radio buttons, lists of choices, and sliders.

### Checkboxes

If you want to collect just a “yes” or “no” input, use a checkbox component. Checkboxes automatically come with labels that identify them. The user usually checks the box by clicking inside it and turns off the check mark by clicking inside the box again. To toggle the check mark, the user can also press the space bar when the focus is in the checkbox.

Figure 9–14 shows a simple program with two checkboxes, one to turn on or off the italic attribute of a font, and the other for boldface. Note that the second checkbox has focus, as indicated by the rectangle around the label. Each time the user clicks one of the checkboxes, the screen is refreshed, using the new font attributes.



**Figure 9–14** Checkboxes

Checkboxes need a label next to them to identify their purpose. You give the label text in the constructor.

```
bold = new JCheckBox("Bold");
```

You use the `setSelected` method to turn a checkbox on or off. For example:

```
bold.setSelected(true);
```

The `isSelected` method then retrieves the current state of each checkbox. It is `false` if unchecked; `true` if checked.

When the user clicks on a checkbox, this triggers an action event. As always, you attach an action listener to the checkbox. In our program, the two checkboxes share the same action listener.

```
ActionListener listener = . . .
bold.addActionListener(listener);
italic.addActionListener(listener);
```

The `actionPerformed` method queries the state of the `bold` and `italic` checkboxes and sets the font of the panel to plain, bold, italic, or both bold and italic.

```

public void actionPerformed(ActionEvent event)
{
    int mode = 0;
    if (bold.isSelected()) mode += Font.BOLD;
    if (italic.isSelected()) mode += Font.ITALIC;
    label.setFont(new Font("Serif", mode, FONTSIZE));
}

```

Listing 9-3 is the complete program listing for the checkbox example.

**Listing 9-3** CheckBoxTest.java

```

1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.33 2007-06-12
7.  * @author Cay Horstmann
8.  */
9. public class CheckBoxTest
10. {
11.     public static void main(String[] args)
12.     {
13.         EventQueue.invokeLater(new Runnable()
14.         {
15.             public void run()
16.             {
17.                 CheckBoxFrame frame = new CheckBoxFrame();
18.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.                 frame.setVisible(true);
20.             }
21.         });
22.     }
23. }
24.
25. /**
26.  * A frame with a sample text label and check boxes for selecting font attributes.
27.  */
28. class CheckBoxFrame extends JFrame
29. {
30.     public CheckBoxFrame()
31.     {
32.         setTitle("CheckBoxTest");
33.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34.
35.         // add the sample text label
36.
37.         label = new JLabel("The quick brown fox jumps over the lazy dog.");
38.         label.setFont(new Font("Serif", Font.PLAIN, FONTSIZE));
39.         add(label, BorderLayout.CENTER);
40.

```

**Listing 9-3** CheckBoxTest.java (continued)

```
41. // this listener sets the font attribute of
42. // the label to the check box state
43.
44. ActionListener listener = new ActionListener()
45. {
46.     public void actionPerformed(ActionEvent event)
47.     {
48.         int mode = 0;
49.         if (bold.isSelected()) mode += Font.BOLD;
50.         if (italic.isSelected()) mode += Font.ITALIC;
51.         label.setFont(new Font("Serif", mode, FONTSIZE));
52.     }
53. };
54.
55. // add the check boxes
56.
57. JPanel buttonPanel = new JPanel();
58.
59. bold = new JCheckBox("Bold");
60. bold.addActionListener(listener);
61. buttonPanel.add(bold);
62.
63. italic = new JCheckBox("Italic");
64. italic.addActionListener(listener);
65. buttonPanel.add(italic);
66.
67. add(buttonPanel, BorderLayout.SOUTH);
68. }
69.
70. public static final int DEFAULT_WIDTH = 300;
71. public static final int DEFAULT_HEIGHT = 200;
72.
73. private JLabel label;
74. private JCheckBox bold;
75. private JCheckBox italic;
76.
77. private static final int FONTSIZE = 12;
78. }
```

**API** javax.swing.JCheckBox 1.2

- JCheckBox(String label)
- JCheckBox(String label, Icon icon)  
constructs a checkbox that is initially unselected.
- JCheckBox(String label, boolean state)  
constructs a checkbox with the given label and initial state.
- boolean isSelected ()
- void setSelected(boolean state)  
gets or sets the selection state of the checkbox.

### Radio Buttons

In the previous example, the user could check either, both, or neither of the two checkboxes. In many cases, we want to require the user to check only one of several boxes. When another box is checked, the previous box is automatically unchecked. Such a group of boxes is often called a *radio button group* because the buttons work like the station selector buttons on a radio. When you push in one button, the previously depressed button pops out. Figure 9–15 shows a typical example. We allow the user to select a font size from among the choices—Small, Medium, Large, and Extra large—but, of course, we will allow the user to select only one size at a time.



**Figure 9–15** A radio button group

Implementing radio button groups is easy in Swing. You construct one object of type `ButtonGroup` for every group of buttons. Then, you add objects of type `JRadioButton` to the button group. The button group object is responsible for turning off the previously set button when a new button is clicked.

```
ButtonGroup group = new ButtonGroup();

JRadioButton smallButton = new JRadioButton("Small", false);
group.add(smallButton);

JRadioButton mediumButton = new JRadioButton("Medium", true);
group.add(mediumButton);
. . .
```

The second argument of the constructor is `true` for the button that should be checked initially and `false` for all others. Note that the button group controls only the *behavior* of the buttons; if you want to group the buttons for layout purposes, you also need to add them to a container such as a `JPanel`.

If you look again at Figures 9–14 and 9–15, you will note that the appearance of the radio buttons is different from that of checkboxes. Checkboxes are square and contain a check mark when selected. Radio buttons are round and contain a dot when selected.

The event notification mechanism for radio buttons is the same as for any other buttons. When the user checks a radio button, the radio button generates an action event. In our example program, we define an action listener that sets the font size to a particular value:

```
ActionListener listener = new
    ActionListener()
    {
```

```

    public void actionPerformed(ActionEvent event)
    {
        // size refers to the final parameter of the addRadioButton method
        label.setFont(new Font("Serif", Font.PLAIN, size));
    }
};

```

Compare this listener setup with that of the checkbox example. Each radio button gets a different listener object. Each listener object knows exactly what it needs to do—set the font size to a particular value. In the case of the checkboxes, we used a different approach. Both checkboxes have the same action listener. It called a method that looked at the current state of both checkboxes.

Could we follow the same approach here? We could have a single listener that computes the size as follows:

```

    if (smallButton.isSelected()) size = 8;
    else if (mediumButton.isSelected()) size = 12;
    . . .

```

However, we prefer to use separate action listener objects because they tie the size values more closely to the buttons.



**NOTE:** If you have a group of radio buttons, you know that only one of them is selected. It would be nice to be able to quickly find out which one without having to query all the buttons in the group. Because the `ButtonGroup` object controls all buttons, it would be convenient if this object could give us a reference to the selected button. Indeed, the `ButtonGroup` class has a `getSelection` method, but that method doesn't return the radio button that is selected. Instead, it returns a `ButtonModel` reference to the model attached to the button. Unfortunately, none of the `ButtonModel` methods are very helpful. The `ButtonModel` interface inherits a method `getSelectedObjects` from the `ItemSelectable` interface that, rather uselessly, returns `null`. The `getActionCommand` method looks promising because the "action command" of a radio button is its text label. But the action command of its model is `null`. Only if you explicitly set the action commands of all radio buttons with the `setActionCommand` method do the models' action command values also get set. Then you can retrieve the action command of the currently selected button with `buttonGroup.getSelection().getActionCommand()`.

Listing 9-4 is the complete program for font size selection that puts a set of radio buttons to work.

#### Listing 9-4 RadioButtonTest.java

```

1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.33 2007-06-12
7.  * @author Cay Horstmann
8.  */
9. public class RadioButtonTest
10. {

```

**Listing 9-4** RadioButtonTest.java (continued)

```
11. public static void main(String[] args)
12. {
13.     EventQueue.invokeLater(new Runnable()
14.     {
15.         public void run()
16.         {
17.             RadioButtonFrame frame = new RadioButtonFrame();
18.             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.             frame.setVisible(true);
20.         }
21.     });
22. }
23. }
24.
25. /**
26.  * A frame with a sample text label and radio buttons for selecting font sizes.
27.  */
28. class RadioButtonFrame extends JFrame
29. {
30.     public RadioButtonFrame()
31.     {
32.         setTitle("RadioButtonTest");
33.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34.
35.         // add the sample text label
36.
37.         label = new JLabel("The quick brown fox jumps over the lazy dog.");
38.         label.setFont(new Font("Serif", Font.PLAIN, DEFAULT_SIZE));
39.         add(label, BorderLayout.CENTER);
40.
41.         // add the radio buttons
42.
43.         buttonPanel = new JPanel();
44.         group = new ButtonGroup();
45.
46.         addRadioButton("Small", 8);
47.         addRadioButton("Medium", 12);
48.         addRadioButton("Large", 18);
49.         addRadioButton("Extra large", 36);
50.
51.         add(buttonPanel, BorderLayout.SOUTH);
52.     }
53.
54.     /**
55.     * Adds a radio button that sets the font size of the sample text.
56.     * @param name the string to appear on the button
57.     * @param size the font size that this button sets
58.     */
```

**Listing 9-4** RadioButtonTest.java (continued)

```
59. public void addRadioButton(String name, final int size)
60. {
61.     boolean selected = size == DEFAULT_SIZE;
62.     JRadioButton button = new JRadioButton(name, selected);
63.     group.add(button);
64.     buttonPanel.add(button);
65.
66.     // this listener sets the label font size
67.
68.     ActionListener listener = new ActionListener()
69.     {
70.         public void actionPerformed(ActionEvent event)
71.         {
72.             // size refers to the final parameter of the addRadioButton
73.             // method
74.             label.setFont(new Font("Serif", Font.PLAIN, size));
75.         }
76.     };
77.
78.     button.addActionListener(listener);
79. }
80.
81. public static final int DEFAULT_WIDTH = 400;
82. public static final int DEFAULT_HEIGHT = 200;
83.
84. private JPanel buttonPanel;
85. private ButtonGroup group;
86. private JLabel label;
87.
88. private static final int DEFAULT_SIZE = 12;
89. }
```

**API** javax.swing.JRadioButton 1.2

- JRadioButton(String label, Icon icon)  
constructs a radio button that is initially unselected.
- JRadioButton(String label, boolean state)  
constructs a radio button with the given label and initial state.

**API** javax.swing.ButtonGroup 1.2

- void add(AbstractButton b)  
adds the button to the group.
- ButtonModel getSelection()  
returns the button model of the selected button.

**API** javax.swing.ButtonModel 1.2

- String getActionCommand()  
returns the action command for this button model.

**API** javax.swing.AbstractButton 1.2

- void setActionCommand(String s)  
sets the action command for this button and its model.

**Borders**

If you have multiple groups of radio buttons in a window, you will want to visually indicate which buttons are grouped. Swing provides a set of useful *borders* for this purpose. You can apply a border to any component that extends `JComponent`. The most common usage is to place a border around a panel and fill that panel with other user interface elements such as radio buttons.

You can choose from quite a few borders, but you follow the same steps for all of them.

1. Call a static method of the `BorderFactory` to create a border. You can choose among the following styles (see Figure 9–16):
  - Lowered bevel
  - Raised bevel
  - Etched
  - Line
  - Matte
  - Empty (just to create some blank space around the component)
2. If you like, add a title to your border by passing your border to `BorderFactory.createTitledBorder`.
3. If you really want to go all out, combine several borders with a call to `BorderFactory.createCompoundBorder`.
4. Add the resulting border to your component by calling the `setBorder` method of the `JComponent` class.

For example, here is how you add an etched border with a title to a panel:

```
Border etched = BorderFactory.createEtchedBorder()
Border titled = BorderFactory.createTitledBorder(etched, "A Title");
panel.setBorder(titled);
```

Run the program in Listing 9–5 to get an idea what the various borders look like.

The various borders have different options for setting border widths and colors. See the API notes for details. True border enthusiasts will appreciate that there is also a `SoftBevelBorder` class for beveled borders with softened corners and that a `LineBorder` can have rounded corners as well. You can construct these borders only by using one of the class constructors—there is no `BorderFactory` method for them.



**Figure 9-16** Testing border types**Listing 9-5** BorderTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4. import javax.swing.border.*;
5.
6. /**
7.  * @version 1.33 2007-06-12
8.  * @author Cay Horstmann
9.  */
10. public class BorderTest
11. {
12.     public static void main(String[] args)
13.     {
14.         EventQueue.invokeLater(new Runnable()
15.         {
16.             public void run()
17.             {
18.                 BorderFrame frame = new BorderFrame();
19.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20.                 frame.setVisible(true);
21.             }
22.         });
23.     }
24. }
25.
26. /**
27.  * A frame with radio buttons to pick a border style.
28.  */
29. class BorderFrame extends JFrame
30. {
31.     public BorderFrame()
32.     {
33.         setTitle("BorderTest");
34.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
35.     }
36. }
```

**Listing 9-5** BorderTest.java (continued)

```

36.     demoPanel = new JPanel();
37.     buttonPanel = new JPanel();
38.     group = new ButtonGroup();
39.
40.     addRadioButton("Lowered bevel", BorderFactory.createLoweredBevelBorder());
41.     addRadioButton("Raised bevel", BorderFactory.createRaisedBevelBorder());
42.     addRadioButton("Etched", BorderFactory.createEtchedBorder());
43.     addRadioButton("Line", BorderFactory.createLineBorder(Color.BLUE));
44.     addRadioButton("Matte", BorderFactory.createMatteBorder(10, 10, 10, 10, Color.BLUE));
45.     addRadioButton("Empty", BorderFactory.createEmptyBorder());
46.
47.     Border etched = BorderFactory.createEtchedBorder();
48.     Border titled = BorderFactory.createTitledBorder(etched, "Border types");
49.     buttonPanel.setBorder(titled);
50.
51.     setLayout(new GridLayout(2, 1));
52.     add(buttonPanel);
53.     add(demoPanel);
54. }
55.
56. public void addRadioButton(String buttonName, final Border b)
57. {
58.     JRadioButton button = new JRadioButton(buttonName);
59.     button.addActionListener(new ActionListener()
60.     {
61.         public void actionPerformed(ActionEvent event)
62.         {
63.             demoPanel.setBorder(b);
64.         }
65.     });
66.     group.add(button);
67.     buttonPanel.add(button);
68. }
69.
70. public static final int DEFAULT_WIDTH = 600;
71. public static final int DEFAULT_HEIGHT = 200;
72.
73. private JPanel demoPanel;
74. private JPanel buttonPanel;
75. private ButtonGroup group;
76. }

```

**API** javax.swing.BorderFactory 1.2

- static Border createLineBorder(Color color)
- static Border createLineBorder(Color color, int thickness)  
creates a simple line border.

- static `MatteBorder createMatteBorder(int top, int left, int bottom, int right, Color color)`
- static `MatteBorder createMatteBorder(int top, int left, int bottom, int right, Icon tileIcon)`  
creates a thick border that is filled with a color or a repeating icon.
- static `Border createEmptyBorder()`
- static `Border createEmptyBorder(int top, int left, int bottom, int right)`  
creates an empty border.
- static `Border createEtchedBorder()`
- static `Border createEtchedBorder(Color highlight, Color shadow)`
- static `Border createEtchedBorder(int type)`
- static `Border createEtchedBorder(int type, Color highlight, Color shadow)`  
creates a line border with a 3D effect.  
*Parameters:*

highlight, shadow	Colors for 3D effect
type	One of <code>EtchedBorder.RAISED</code> , <code>EtchedBorder.LOWERED</code>
- static `Border createBevelBorder(int type)`
- static `Border createBevelBorder(int type, Color highlight, Color shadow)`
- static `Border createLoweredBevelBorder()`
- static `Border createRaisedBevelBorder()`  
creates a border that gives the effect of a lowered or raised surface.  
*Parameters:*

highlight, shadow	Colors for 3D effect
type	One of <code>EtchedBorder.RAISED</code> , <code>EtchedBorder.LOWERED</code>
- static `TitledBorder createTitledBorder(String title)`
- static `TitledBorder createTitledBorder(Border border)`
- static `TitledBorder createTitledBorder(Border border, String title)`
- static `TitledBorder createTitledBorder(Border border, String title, int justification, int position)`
- static `TitledBorder createTitledBorder(Border border, String title, int justification, int position, Font font)`
- static `TitledBorder createTitledBorder(Border border, String title, int justification, int position, Font font, Color color)`  
creates a titled border with the specified properties.  
*Parameters:*

title	The title string
border	The border to decorate with the title
justification	One of the <code>TitledBorder</code> constants <code>LEFT</code> , <code>CENTER</code> , <code>RIGHT</code> , <code>LEADING</code> , <code>TRAILING</code> , or <code>DEFAULT_JUSTIFICATION</code> (left)
position	One of the <code>TitledBorder</code> constants <code>ABOVE_TOP</code> , <code>TOP</code> , <code>BELOW_TOP</code> , <code>ABOVE_BOTTOM</code> , <code>BOTTOM</code> , <code>BELOW_BOTTOM</code> , or <code>DEFAULT_POSITION</code> (top)
font	The font for the title
color	The color of the title
- static `CompoundBorder createCompoundBorder(Border outsideBorder, Border insideBorder)`  
combines two borders to a new border.

**API** `javax.swing.border.SoftBevelBorder` 1.2

- `SoftBevelBorder(int type)`
- `SoftBevelBorder(int type, Color highlight, Color shadow)`  
creates a bevel border with softened corners.

*Parameters:*    highlight, shadow    Colors for 3D effect  
                         type                    One of `EtchedBorder.RAISED`, `EtchedBorder.LOWERED`

**API** `javax.swing.border.LineBorder` 1.2

- `public LineBorder(Color color, int thickness, boolean roundedCorners)`  
creates a line border with the given color and thickness. If `roundedCorners` is true, the border has rounded corners.

**API** `javax.swing.JComponent` 1.2

- `void setBorder(Border border)`  
sets the border of this component.

**Combo Boxes**

If you have more than a handful of alternatives, radio buttons are not a good choice because they take up too much screen space. Instead, you can use a combo box. When the user clicks on the component, a list of choices drops down, and the user can then select one of them (see Figure 9–17).



**Figure 9–17** A combo box

If the drop-down list box is set to be *editable*, then you can edit the current selection as if it were a text field. For that reason, this component is called a *combo box*—it combines the flexibility of a text field with a set of predefined choices. The `JComboBox` class provides a combo box component.

You call the `setEditable` method to make the combo box editable. Note that editing affects only the current item. It does not change the content of the list.

You can obtain the current selection or edited text by calling the `getSelectedItem` method.

In the example program, the user can choose a font style from a list of styles (Serif, Sans-Serif, Monospaced, etc.). The user can also type in another font.

You add the choice items with the `addItem` method. In our program, `addItem` is called only in the constructor, but you can call it any time.

```
faceCombo = new JComboBox();
faceCombo.setEditable(true);
faceCombo.addItem("Serif");
faceCombo.addItem("SansSerif");
. . .
```

This method adds the string at the end of the list. You can add new items anywhere in the list with the `insertItemAt` method:

```
faceCombo.insertItemAt("Monospaced", 0); // add at the beginning
```

You can add items of any type—the combo box invokes each item's `toString` method to display it.

If you need to remove items at runtime, you use the `removeItem` or `removeItemAt` method, depending on whether you supply the item to be removed or its position.

```
faceCombo.removeItem("Monospaced");
faceCombo.removeItemAt(0); // remove first item
```

The `removeAllItems` method removes all items at once.



**TIP:** If you need to add a large number of items to a combo box, the `addItem` method will perform poorly. Instead, construct a `DefaultComboBoxModel`, populate it by calling `addElement`, and then call the `setModel` method of the `JComboBox` class.

When the user selects an item from a combo box, the combo box generates an action event. To find out which item was selected, call `getSource` on the event parameter to get a reference to the combo box that sent the event. Then call the `getSelectedItem` method to retrieve the currently selected item. You need to cast the returned value to the appropriate type, usually `String`.

```
public void actionPerformed(ActionEvent event)
{
    label.setFont(new Font(
        (String) faceCombo.getSelectedItem(),
        Font.PLAIN,
        DEFAULT_SIZE));
}
```

Listing 9-6 shows the complete program.



**NOTE:** If you want to show a permanently displayed list instead of a dropdown list, use the `JList` component. We cover `JList` in Chapter 6 of Volume II.

**Listing 9-6** ComboBoxTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.33 2007-06-12
7.  * @author Cay Horstmann
8.  */
9. public class ComboBoxTest
10. {
11.     public static void main(String[] args)
12.     {
13.         EventQueue.invokeLater(new Runnable()
14.         {
15.             public void run()
16.             {
17.
18.                 ComboBoxFrame frame = new ComboBoxFrame();
19.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20.                 frame.setVisible(true);
21.             }
22.         });
23.     }
24. }
25.
26. /**
27.  * A frame with a sample text label and a combo box for selecting font faces.
28.  */
29. class ComboBoxFrame extends JFrame
30. {
31.     public ComboBoxFrame()
32.     {
33.         setTitle("ComboBoxTest");
34.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
35.
36.         // add the sample text label
37.
38.         label = new JLabel("The quick brown fox jumps over the lazy dog.");
39.         label.setFont(new Font("Serif", Font.PLAIN, DEFAULT_SIZE));
40.         add(label, BorderLayout.CENTER);
41.
42.         // make a combo box and add face names
43.
44.         faceCombo = new JComboBox();
45.         faceCombo.setEditable(true);
46.         faceCombo.addItem("Serif");
47.         faceCombo.addItem("SansSerif");
48.         faceCombo.addItem("Monospaced");
49.         faceCombo.addItem("Dialog");
50.         faceCombo.addItem("DialogInput");
```

**Listing 9-6** ComboBoxTest.java (continued)

```
51.
52.     // the combo box listener changes the label font to the selected face name
53.
54.     faceCombo.addActionListener(new ActionListener()
55.     {
56.         public void actionPerformed(ActionEvent event)
57.         {
58.             label.setFont(new Font((String) faceCombo.getSelectedItem(), Font.PLAIN,
59.                 DEFAULT_SIZE));
60.         }
61.     });
62.
63.     // add combo box to a panel at the frame's southern border
64.
65.     JPanel comboPanel = new JPanel();
66.     comboPanel.add(faceCombo);
67.     add(comboPanel, BorderLayout.SOUTH);
68. }
69.
70. public static final int DEFAULT_WIDTH = 300;
71. public static final int DEFAULT_HEIGHT = 200;
72.
73. private JComboBox faceCombo;
74. private JLabel label;
75. private static final int DEFAULT_SIZE = 12;
76. }
```

**API** javax.swing.JComboBox 1.2

- boolean isEditable()
- void setEditable(boolean b)  
gets or sets the editable property of this combo box.
- void addItem(Object item)  
adds an item to the item list.
- void insertItemAt(Object item, int index)  
inserts an item into the item list at a given index.
- void removeItem(Object item)  
removes an item from the item list.
- void removeItemAt(int index)  
removes the item at an index.
- void removeAllItems()  
removes all items from the item list.
- Object getSelectedItem()  
returns the currently selected item.

### Sliders

Combo boxes let users choose from a discrete set of values. Sliders offer a choice from a continuum of values, for example, any number between 1 and 100.

The most common way of constructing a slider is as follows:

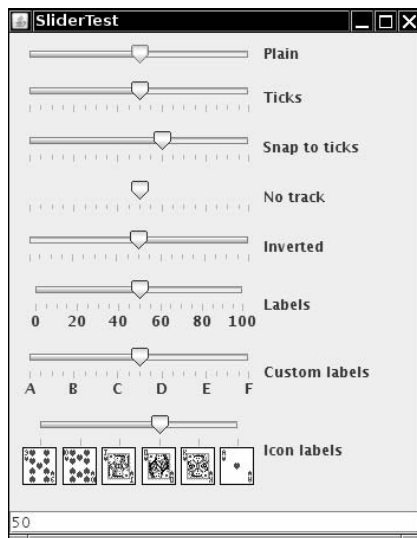
```
JSlider slider = new JSlider(min, max, initialValue);
```

If you omit the minimum, maximum, and initial values, they are initialized with 0, 100, and 50, respectively.

Or if you want the slider to be vertical, then use the following constructor call:

```
JSlider slider = new JSlider(SwingConstants.VERTICAL, min, max, initialValue);
```

These constructors create a plain slider, such as the top slider in Figure 9–18. You will see presently how to add decorations to a slider.



**Figure 9–18 Sliders**

As the user slides the slider bar, the *value* of the slider moves between the minimum and the maximum values. When the value changes, a `ChangeEvent` is sent to all change listeners. To be notified of the change, you call the `addChangeListener` method and install an object that implements the `ChangeListener` interface. That interface has a single method, `stateChanged`. In that method, you should retrieve the slider value:

```
public void stateChanged(ChangeEvent event)
{
    JSlider slider = (JSlider) event.getSource();
    int value = slider.getValue();
    . . .
}
```



You can embellish the slider by showing *ticks*. For example, in the sample program, the second slider uses the following settings:

```
slider.setMajorTickSpacing(20);  
slider.setMinorTickSpacing(5);
```

The slider is decorated with large tick marks every 20 units and small tick marks every 5 units. The units refer to slider values, not pixels.

These instructions only set the units for the tick marks. To actually have the tick marks appear, you also call

```
slider.setPaintTicks(true);
```

The major and minor tick marks are independent. For example, you can set major tick marks every 20 units and minor tick marks every 7 units, but you'll get a very messy scale.

You can force the slider to *snap to ticks*. Whenever the user has finished dragging a slider in snap mode, it is immediately moved to the closest tick. You activate this mode with the call

```
slider.setSnapToTicks(true);
```



**CAUTION:** The “snap to ticks” behavior doesn’t work as well as you might imagine. Until the slider has actually snapped, the change listener still reports slider values that don’t correspond to ticks. And if you click next to the slider—an action that normally advances the slider a bit in the direction of the click—a slider with “snap to ticks” does not move to the next tick.

You can ask for *tick mark labels* for the major tick marks by calling

```
slider.setPaintLabels(true);
```

For example, with a slider ranging from 0 to 100 and major tick spacing of 20, the ticks are labeled 0, 20, 40, 60, 80, and 100.

You can also supply other tick marks, such as strings or icons (see Figure 9–18). The process is a bit convoluted. You need to fill a hash table with keys of type `Integer` and values of type `Component`. (Autoboxing makes this simple in Java SE 5.0 and beyond.) You then call the `setLabelTable` method. The components are placed under the tick marks. Usually, you use `JLabel` objects. Here is how you can label ticks as A, B, C, D, E, and F:

```
Hashtable<Integer, Component> labelTable = new Hashtable<Integer, Component>();  
labelTable.put(0, new JLabel("A"));  
labelTable.put(20, new JLabel("B"));  
...  
labelTable.put(100, new JLabel("F"));  
slider.setLabelTable(labelTable);
```

See Chapter 2 of Volume II for more information about hash tables.

Listing 9–7 also shows a slider with icons as tick labels.



**TIP:** If your tick marks or labels don’t show, double-check that you called `setPaintTicks(true)` and `setPaintLabels(true)`.

The fourth slider in Figure 9–18 has no track. To suppress the “track” in which the slider moves, call

```
slider.setPaintTrack(false);
```

The fifth slider has its direction reversed by a call to

```
slider.setInverted(true);
```

The example program shows all these visual effects with a collection of sliders. Each slider has a change event listener installed that places the current slider value into the text field at the bottom of the frame.

#### Listing 9–7 SliderTest.java

```

1. import java.awt.*;
2. import java.util.*;
3. import javax.swing.*;
4. import javax.swing.event.*;
5.
6. /**
7.  * @version 1.13 2007-06-12
8.  * @author Cay Horstmann
9.  */
10. public class SliderTest
11. {
12.     public static void main(String[] args)
13.     {
14.         EventQueue.invokeLater(new Runnable()
15.         {
16.             public void run()
17.             {
18.                 SliderTestFrame frame = new SliderTestFrame();
19.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20.                 frame.setVisible(true);
21.             }
22.         });
23.     }
24. }
25.
26. /**
27.  * A frame with many sliders and a text field to show slider values.
28.  */
29. class SliderTestFrame extends JFrame
30. {
31.     public SliderTestFrame()
32.     {
33.         setTitle("SliderTest");
34.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
35.
36.         sliderPanel = new JPanel();
37.         sliderPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
38.

```

**Listing 9-7** SliderTest.java (continued)

```
39. // common listener for all sliders
40. listener = new ChangeListener()
41. {
42.     public void stateChanged(ChangeEvent event)
43.     {
44.         // update text field when the slider value changes
45.         JSlider source = (JSlider) event.getSource();
46.         textField.setText("" + source.getValue());
47.     }
48. };
49.
50. // add a plain slider
51.
52. JSlider slider = new JSlider();
53. addSlider(slider, "Plain");
54.
55. // add a slider with major and minor ticks
56.
57. slider = new JSlider();
58. slider.setPaintTicks(true);
59. slider.setMajorTickSpacing(20);
60. slider.setMinorTickSpacing(5);
61. addSlider(slider, "Ticks");
62.
63. // add a slider that snaps to ticks
64.
65. slider = new JSlider();
66. slider.setPaintTicks(true);
67. slider.setSnapToTicks(true);
68. slider.setMajorTickSpacing(20);
69. slider.setMinorTickSpacing(5);
70. addSlider(slider, "Snap to ticks");
71.
72. // add a slider with no track
73.
74. slider = new JSlider();
75. slider.setPaintTicks(true);
76. slider.setMajorTickSpacing(20);
77. slider.setMinorTickSpacing(5);
78. slider.setPaintTrack(false);
79. addSlider(slider, "No track");
80.
81. // add an inverted slider
82.
83. slider = new JSlider();
84. slider.setPaintTicks(true);
85. slider.setMajorTickSpacing(20);
86. slider.setMinorTickSpacing(5);
87. slider.setInverted(true);
88. addSlider(slider, "Inverted");
```

**Listing 9-7** SliderTest.java (continued)

```
89.
90.     // add a slider with numeric labels
91.
92.     slider = new JSlider();
93.     slider.setPaintTicks(true);
94.     slider.setPaintLabels(true);
95.     slider.setMajorTickSpacing(20);
96.     slider.setMinorTickSpacing(5);
97.     addSlider(slider, "Labels");
98.
99.     // add a slider with alphabetic labels
100.
101.     slider = new JSlider();
102.     slider.setPaintLabels(true);
103.     slider.setPaintTicks(true);
104.     slider.setMajorTickSpacing(20);
105.     slider.setMinorTickSpacing(5);
106.
107.     Dictionary<Integer, Component> labelTable = new Hashtable<Integer, Component>();
108.     labelTable.put(0, new JLabel("A"));
109.     labelTable.put(20, new JLabel("B"));
110.     labelTable.put(40, new JLabel("C"));
111.     labelTable.put(60, new JLabel("D"));
112.     labelTable.put(80, new JLabel("E"));
113.     labelTable.put(100, new JLabel("F"));
114.
115.     slider.setLabelTable(labelTable);
116.     addSlider(slider, "Custom labels");
117.
118.     // add a slider with icon labels
119.
120.     slider = new JSlider();
121.     slider.setPaintTicks(true);
122.     slider.setPaintLabels(true);
123.     slider.setSnapToTicks(true);
124.     slider.setMajorTickSpacing(20);
125.     slider.setMinorTickSpacing(20);
126.
127.     labelTable = new Hashtable<Integer, Component>();
128.
129.     // add card images
130.
131.     labelTable.put(0, new JLabel(new ImageIcon("nine.gif")));
132.     labelTable.put(20, new JLabel(new ImageIcon("ten.gif")));
133.     labelTable.put(40, new JLabel(new ImageIcon("jack.gif")));
134.     labelTable.put(60, new JLabel(new ImageIcon("queen.gif")));
135.     labelTable.put(80, new JLabel(new ImageIcon("king.gif")));
136.     labelTable.put(100, new JLabel(new ImageIcon("ace.gif")));
137.
```

**Listing 9-7** SliderTest.java (continued)

```

138.     slider.setLabelTable(labelTable);
139.     addSlider(slider, "Icon labels");
140.
141.     // add the text field that displays the slider value
142.
143.     textField = new JTextField();
144.     add(sliderPanel, BorderLayout.CENTER);
145.     add(textField, BorderLayout.SOUTH);
146. }
147.
148. /**
149.  * Adds a slider to the slider panel and hooks up the listener
150.  * @param s the slider
151.  * @param description the slider description
152.  */
153. public void addSlider(JSlider s, String description)
154. {
155.     s.addChangeListener(listener);
156.     JPanel panel = new JPanel();
157.     panel.add(s);
158.     panel.add(new JLabel(description));
159.     sliderPanel.add(panel);
160. }
161.
162. public static final int DEFAULT_WIDTH = 350;
163. public static final int DEFAULT_HEIGHT = 450;
164.
165. private JPanel sliderPanel;
166. private JTextField textField;
167. private ChangeListener listener;
168. }

```

**API** javax.swing.JSlider 1.2

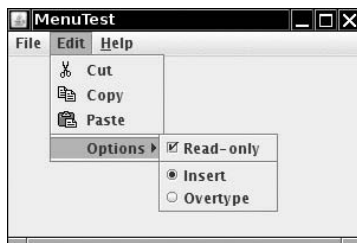
- JSlider()
  - JSlider(int direction)
  - JSlider(int min, int max)
  - JSlider(int min, int max, int initialValue)
  - JSlider(int direction, int min, int max, int initialValue)
- constructs a horizontal slider with the given direction and minimum, maximum, and initial values.
- |                    |              |   |
|--------------------|--------------|---|
| <i>Parameters:</i> | direction    | One of SwingConstants.HORIZONTAL or SwingConstants.VERTICAL. The default is horizontal. |
|                    | min, max     | The minimum and maximum for the slider values. Defaults are 0 and 100.                  |
|                    | initialValue | The initial value for the slider. The default is 50.                                    |

- `void setPaintTicks(boolean b)`  
displays ticks if `b` is true.
- `void setMajorTickSpacing(int units)`
- `void setMinorTickSpacing(int units)`  
sets major or minor ticks at multiples of the given slider units.
- `void setPaintLabels(boolean b)`  
displays tick labels if `b` is true.
- `void setLabelTable(Dictionary table)`  
sets the components to use for the tick labels. Each key/value pair in the table has the form `new Integer(value)/component`.
- `void setSnapToTicks(boolean b)`  
if `b` is true, then the slider snaps to the closest tick after each adjustment.
- `void setPaintTrack(boolean b)`  
if `b` is true, then a track is displayed in which the slider runs.

## Menus

We started this chapter by introducing the most common components that you might want to place into a window, such as various kinds of buttons, text fields, and combo boxes. Swing also supports another type of user interface element, the pull-down menu that are familiar from GUI applications.

A *menu bar* on top of the window contains the names of the pull-down menus. Clicking on a name opens the menu containing *menu items* and *submenus*. When the user clicks on a menu item, all menus are closed and a message is sent to the program. Figure 9–19 shows a typical menu with a submenu.



**Figure 9–19** A menu with a submenu

## Menu Building

Building menus is straightforward. You first create a menu bar:

```
JMenuBar menuBar = new JMenuBar();
```

A menu bar is just a component that you can add anywhere you like. Normally, you want it to appear at the top of a frame. You can add it there with the `setJMenuBar` method:

```
frame.setJMenuBar(menuBar);
```

For each menu, you create a menu object:

```
JMenu editMenu = new JMenu("Edit");
```

You add the top-level menus to the menu bar:

```
menuBar.add(editMenu);
```

You add menu items, separators, and submenus to the menu object:

```
JMenuItem pasteItem = new JMenuItem("Paste");
editMenu.add(pasteItem);
editMenu.addSeparator();
JMenu optionsMenu = . . .; // a submenu
editMenu.add(optionsMenu);
```

You can see separators in Figure 9–19 below the “Paste” and “Read-only” menu items.

When the user selects a menu, an action event is triggered. You need to install an action listener for each menu item:

```
ActionListener listener = . . .;
pasteItem.addActionListener(listener);
```

The method `JMenu.add(String s)` conveniently adds a menu item to the end of a menu. For example:

```
editMenu.add("Paste");
```

The `add` method returns the created menu item, so you can capture it and then add the listener, as follows:

```
JMenuItem pasteItem = editMenu.add("Paste");
pasteItem.addActionListener(listener);
```

It often happens that menu items trigger commands that can also be activated through other user interface elements such as toolbar buttons. In Chapter 8, you saw how to specify commands through Action objects. You define a class that implements the `Action` interface, usually by extending the `AbstractAction` convenience class. You specify the menu item label in the constructor of the `AbstractAction` object, and you override the `actionPerformed` method to hold the menu action handler. For example:

```
Action exitAction = new AbstractAction("Exit") // menu item text goes here
{
    public void actionPerformed(ActionEvent event)
    {
        // action code goes here
        System.exit(0);
    }
};
```

You can then add the action to the menu:

```
JMenuItem exitItem = fileMenu.add(exitAction);
```

This command adds a menu item to the menu, using the action name. The action object becomes its listener. This is just a convenient shortcut for

```
JMenuItem exitItem = new JMenuItem(exitAction);
fileMenu.add(exitItem);
```



**NOTE:** In Windows and Macintosh programs, menus are generally defined in an external resource file and tied to the application with resource identifiers. In Java, menus are still usually built inside the program because the mechanism for dealing with external resources is far more limited than it is in Windows or Mac OS.

**API** javax.swing.JMenu 1.2

- JMenu(String label)  
constructs a menu with the given label.
- JMenuItem add(JMenuItem item)  
adds a menu item (or a menu).
- JMenuItem add(String label)  
adds a menu item with the given label to this menu and returns the item.
- JMenuItem add(Action a)  
adds a menu item with the given action to this menu and returns the item.
- void addSeparator()  
adds a separator line to the menu.
- JMenuItem insert(JMenuItem menu, int index)  
adds a new menu item (or submenu) to the menu at a specific index.
- JMenuItem insert(Action a, int index)  
adds a new menu item with the given action at a specific index.
- void insertSeparator(int index)  
adds a separator to the menu.

*Parameters:*    index            Where to add the separator

- void remove(int index)
- void remove(JMenuItem item)  
removes a specific item from the menu.

**API** javax.swing.JMenuItem 1.2

- JMenuItem(String label)  
constructs a menu item with a given label.
- JMenuItem(Action a) 1.3  
constructs a menu item for the given action.

**API** javax.swing.AbstractButton 1.2

- void setAction(Action a) 1.3  
sets the action for this button or menu item.

**API** javax.swing.JFrame 1.2

- void setJMenuBar(JMenuBar menubar)  
sets the menu bar for this frame.

**Icons in Menu Items**

Menu items are very similar to buttons. In fact, the JMenuItem class extends the AbstractButton class. Just like buttons, menus can have just a text label, just an icon, or both. You can specify the icon with the JMenuItem(String, Icon) or JMenuItem(Icon) constructor, or you can



set it with the `setIcon` method that the `JMenuItem` class inherits from the `AbstractButton` class. Here is an example:

```
JMenuItem cutItem = new JMenuItem("Cut", new ImageIcon("cut.gif"));
```

In Figure 9–19 on page 406, you can see icons next to several menu items. By default, the menu item text is placed to the right of the icon. If you prefer the text to be placed on the left, call the `setHorizontalTextPosition` method that the `JMenuItem` class inherits from the `AbstractButton` class. For example, the call

```
cutItem.setHorizontalTextPosition(SwingConstants.LEFT);
```

moves the menu item text to the left of the icon.

You can also add an icon to an action:

```
cutAction.putValue(Action.SMALL_ICON, new ImageIcon("cut.gif"));
```

Whenever you construct a menu item out of an action, the `Action.NAME` value becomes the text of the menu item and the `Action.SMALL_ICON` value becomes the icon.

Alternatively, you can set the icon in the `AbstractAction` constructor:

```
cutAction = new
    AbstractAction("Cut", new ImageIcon("cut.gif"))
    {
        public void actionPerformed(ActionEvent event)
        {
            // action code goes here
        }
    };
```

#### API javax.swing.JMenuItem 1.2

- `JMenuItem(String label, Icon icon)` constructs a menu item with the given label and icon.

#### API javax.swing.AbstractButton 1.2

- `void setHorizontalTextPosition(int pos)` sets the horizontal position of the text relative to the icon.  
*Parameters:*    `pos`                    `SwingConstants.RIGHT` (text is to the right of icon) or `SwingConstants.LEFT`

#### API javax.swing.AbstractAction 1.2

- `AbstractAction(String name, Icon smallIcon)` constructs an abstract action with the given name and icon.

### Checkbox and Radio Button Menu Items

*Checkbox* and *radio button* menu items display a checkbox or radio button next to the name (see Figure 9–19 on page 406). When the user selects the menu item, the item automatically toggles between checked and unchecked.

Apart from the button decoration, you treat these menu items just as you would any others. For example, here is how you create a checkbox menu item:

```
JCheckBoxMenuItem readonlyItem = new JCheckBoxMenuItem("Read-only");
optionsMenu.add(readonlyItem);
```

The radio button menu items work just like regular radio buttons. You must add them to a button group. When one of the buttons in a group is selected, all others are automatically deselected.

```
ButtonGroup group = new ButtonGroup();
JRadioButtonMenuItem insertItem = new JRadioButtonMenuItem("Insert");
insertItem.setSelected(true);
JRadioButtonMenuItem overtypItem = new JRadioButtonMenuItem("Overtyp");
group.add(insertItem);
group.add(overtypItem);
optionsMenu.add(insertItem);
optionsMenu.add(overtypItem);
```

With these menu items, you don't necessarily want to be notified at the exact moment the user selects the item. Instead, you can simply use the `isSelected` method to test the current state of the menu item. (Of course, that means that you should keep a reference to the menu item stored in an instance field.) Use the `setSelected` method to set the state.

#### API javax.swing.JCheckBoxMenuItem 1.2

- `JCheckBoxMenuItem(String label)`  
constructs the checkbox menu item with the given label.
- `JCheckBoxMenuItem(String label, boolean state)`  
constructs the checkbox menu item with the given label and the given initial state (true is checked).

#### API javax.swing.JRadioButtonMenuItem 1.2

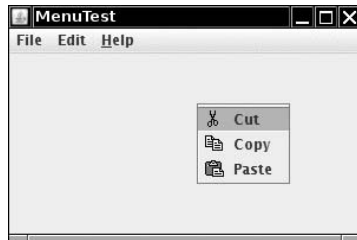
- `JRadioButtonMenuItem(String label)`  
constructs the radio button menu item with the given label.
- `JRadioButtonMenuItem(String label, boolean state)`  
constructs the radio button menu item with the given label and the given initial state (true is checked).

#### API javax.swing.AbstractButton 1.2

- `boolean isSelected()`
- `void setSelected(boolean state)`  
gets or sets the selection state of this item (true is checked).

### Pop-Up Menus

A *pop-up menu* is a menu that is not attached to a menu bar but that floats somewhere (see Figure 9–20).



**Figure 9-20** A pop-up menu

You create a pop-up menu similarly to the way you create a regular menu, but a pop-up menu has no title.

```
JPopupMenu popup = new JPopupMenu();
```

You then add menu items in the usual way:

```
JMenuItem item = new JMenuItem("Cut");
item.addActionListener(listener);
popup.add(item);
```

Unlike the regular menu bar that is always shown at the top of the frame, you must explicitly display a pop-up menu by using the `show` method. You specify the parent component and the location of the pop-up, using the coordinate system of the parent. For example:

```
popup.show(panel, x, y);
```

Usually you write code to pop up a menu when the user clicks a particular mouse button, the so-called *pop-up trigger*. In Windows and Linux, the pop-up trigger is the non-primary (usually, the right) mouse button. To pop up a menu when the user clicks on a component, using the pop-up trigger, simply call the method

```
component.setComponentPopupMenu(popup);
```

Very occasionally, you may place a component inside another component that has a pop-up menu. The child component can inherit the parent component's pop-up menu by calling

```
child.setInheritsPopupMenu(true);
```

These methods were added in Java SE 5.0 to insulate programmers from system dependencies with pop-up menus. Before Java SE 5.0, you had to install a mouse listener and add the following code to *both* the `mousePressed` and the `mouseReleased` listener methods:

```
if (popup.isPopupTrigger(event))
    popup.show(event.getComponent(), event.getX(), event.getY());
```

Some systems trigger pop-ups when the mouse button goes down, others when the mouse button goes up.

**API** javax.swing.JPopupMenu 1.2

- void show(Component c, int x, int y)  
shows the pop-up menu.

*Parameters:*

c	The component over which the pop-up menu is to appear
x, y	The coordinates (in the coordinate space of c) of the top-left corner of the pop-up menu

- boolean isPopupTrigger(MouseEvent event) 1.3  
returns true if the mouse event is the pop-up menu trigger.

**API** java.awt.event.MouseEvent 1.1

- boolean isPopupTrigger()  
returns true if this mouse event is the pop-up menu trigger.

**API** javax.swing.JComponent 1.2

- JPopupMenu getComponentPopupMenu() 5.0
- void setComponentPopupMenu(JPopupMenu popup) 5.0  
gets or sets the pop-up menu for this component.
- boolean getInheritsPopupMenu() 5.0
- void setInheritsPopupMenu(boolean b) 5.0  
gets or sets the inheritsPopupMenu property. If the property is set and this component's pop-up menu is null, it uses its parent's pop-up menu.

**Keyboard Mnemonics and Accelerators**

It is a real convenience for the experienced user to select menu items by *keyboard mnemonics*. You can specify keyboard mnemonics for menu items by specifying a mnemonic letter in the menu item constructor:

```
JMenuItem aboutItem = new JMenuItem("About", 'A');
```

The keyboard mnemonic is displayed automatically in the menu, with the mnemonic letter underlined (see Figure 9–21). For example, in the item defined in the last example, the label will be displayed as “About” with an underlined letter “A”. When the menu is displayed, the user just needs to press the A key, and the menu item is selected. (If the mnemonic letter is not part of the menu string, then typing it still selects the item, but the mnemonic is not displayed in the menu. Naturally, such invisible mnemonics are of dubious utility.)

Sometimes, you don't want to underline the first letter of the menu item that matches the mnemonic. For example, if you have a mnemonic “A” for the menu item “Save As,” then it makes more sense to underline the second “A” (Save As). As of Java SE 1.4, you can specify which character you want to have underlined; call the `setDisplayMnemonicIndex` method.

If you have an `Action` object, you can add the mnemonic as the value of the `Action.MNEMONIC_KEY` key, as follows:

```
cutAction.putValue(Action.MNEMONIC_KEY, new Integer('A'));
```



**Figure 9-21 Keyboard mnemonics**

You can supply a mnemonic letter only in the constructor of a menu item, not in the constructor for a menu. Instead, to attach a mnemonic to a menu, you call the `setMnemonic` method:

```
JMenu helpMenu = new JMenu("Help");  
helpMenu.setMnemonic('H');
```

To select a top-level menu from the menu bar, you press the ALT key together with the mnemonic letter. For example, you press ALT+H to select the Help menu from the menu bar.

Keyboard mnemonics let you select a submenu or menu item from the currently open menu. In contrast, *accelerators* are keyboard shortcuts that let you select menu items without ever opening a menu. For example, many programs attach the accelerators CTRL+O and CTRL+S to the Open and Save items in the File menu. You use the `setAccelerator` method to attach an accelerator key to a menu item. The `setAccelerator` method takes an object of type `KeyStroke`. For example, the following call attaches the accelerator CTRL+O to the `openItem` menu item:

```
openItem.setAccelerator(KeyStroke.getKeyStroke("ctrl O"));
```

When the user presses the accelerator key combination, this automatically selects the menu option and fires an action event, as if the user had selected the menu option manually.

You can attach accelerators only to menu items, not to menus. Accelerator keys don't actually open the menu. Instead, they directly fire the action event that is associated with a menu.

Conceptually, adding an accelerator to a menu item is similar to the technique of adding an accelerator to a Swing component. (We discussed that technique in Chapter 8.) However, when the accelerator is added to a menu item, the key combination is automatically displayed in the menu (see Figure 9-22).



**NOTE:** Under Windows, ALT+F4 closes a window. But this is not an accelerator that was programmed in Java. It is a shortcut defined by the operating system. This key combination will always trigger the `WindowClosing` event for the active window regardless of whether there is a Close item on the menu.

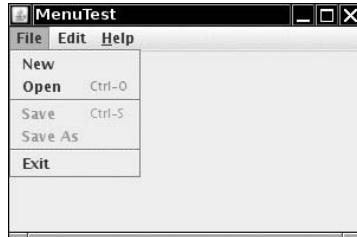


Figure 9-22 Accelerators

**API** javax.swing.JMenuItem 1.2

- JMenuItem(String label, int mnemonic)  
constructs a menu item with a given label and mnemonic.  
*Parameters:* label           The label for this menu item  
                  mnemonic       The mnemonic character for the item; this character will be underlined in the label
- void setAccelerator(KeyStroke k)  
sets the keystroke k as accelerator for this menu item. The accelerator key is displayed next to the label.

**API** javax.swing.AbstractButton 1.2

- void setMnemonic(int mnemonic)  
sets the mnemonic character for the button. This character will be underlined in the label.
- void setDisplayedMnemonicIndex(int index) 1.4  
sets the index of the character to be underlined in the button text. Use this method if you don't want the first occurrence of the mnemonic character to be underlined.

**Enabling and Disabling Menu Items**

Occasionally, a particular menu item should be selected only in certain contexts. For example, when a document is opened for reading only, then the Save menu item is not meaningful. Of course, we could remove the item from the menu with the `JMenu.remove` method, but users would react with some surprise to menus whose content keeps changing. Instead, it is better to deactivate the menu items that lead to temporarily inappropriate commands. A deactivated menu item is shown in gray, and it cannot be selected (see Figure 9-23).

To enable or disable a menu item, use the `setEnabled` method:

```
saveItem.setEnabled(false);
```

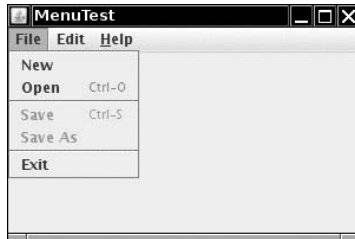
There are two strategies for enabling and disabling menu items. Each time circumstances change, you can call `setEnabled` on the relevant menu items or actions. For example, as soon as a document has been set to read-only mode, you can locate the Save and Save As menu items and disable them. Alternatively, you can disable items just before

displaying the menu. To do this, you must register a listener for the “menu selected” event. The `javax.swing.event` package defines a `MenuListener` interface with three methods:

```
void menuSelected(MenuEvent event)
void menuDeselected(MenuEvent event)
void menuCanceled(MenuEvent event)
```

The `menuSelected` method is called *before* the menu is displayed. It can therefore be used to disable or enable menu items. The following code shows how to disable the Save and Save As actions whenever the Read Only checkbox menu item is selected:

```
public void menuSelected(MenuEvent event)
{
    saveAction.setEnabled(!readOnlyItem.isSelected());
    saveAsAction.setEnabled(!readOnlyItem.isSelected());
}
```



**Figure 9-23** Disabled menu items



**CAUTION:** Disabling menu items just before displaying the menu is a clever idea, but it does not work for menu items that also have accelerator keys. Because the menu is never opened when the accelerator key is pressed, the action is never disabled, and it is still triggered by the accelerator key.

**API** `javax.swing.JMenuItem` 1.2

- `void setEnabled(boolean b)`  
enables or disables the menu item.

**API** `javax.swing.event.MenuListener` 1.2

- `void menuSelected(MenuEvent e)`  
is called when the menu has been selected, before it is opened.
- `void menuDeselected(MenuEvent e)`  
is called when the menu has been deselected, after it has been closed.
- `void menuCanceled(MenuEvent e)`  
is called when the menu has been canceled, for example, by a user clicking outside the menu.

Listing 9–8 is a sample program that generates a set of menus. It shows all the features that you saw in this section: nested menus, disabled menu items, checkbox and radio button menu items, a pop-up menu, and keyboard mnemonics and accelerators.

**Listing 9–8** MenuTest.java

```
1. import java.awt.EventQueue;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.23 2007-05-30
7.  * @author Cay Horstmann
8.  */
9. public class MenuTest
10. {
11.     public static void main(String[] args)
12.     {
13.         EventQueue.invokeLater(new Runnable()
14.         {
15.             public void run()
16.             {
17.                 MenuFrame frame = new MenuFrame();
18.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.                 frame.setVisible(true);
20.             }
21.         });
22.     }
23. }
24.
25. /**
26.  * A frame with a sample menu bar.
27.  */
28. class MenuFrame extends JFrame
29. {
30.     public MenuFrame()
31.     {
32.         setTitle("MenuTest");
33.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34.
35.         JMenu fileMenu = new JMenu("File");
36.         fileMenu.add(new TestAction("New"));
37.
38.         // demonstrate accelerators
39.
40.         JMenuItem openItem = fileMenu.add(new TestAction("Open"));
41.         openItem.setAccelerator(KeyStroke.getKeyStroke("ctrl O"));
42.
43.         fileMenu.addSeparator();
44.     }
45. }
```



**Listing 9-8** MenuTest.java (continued)

```
45. saveAction = new TestAction("Save");
46. JMenuItem saveItem = fileMenu.add(saveAction);
47. saveItem.setAccelerator(KeyStroke.getKeyStroke("ctrl S"));
48.
49. saveAsAction = new TestAction("Save As");
50. fileMenu.add(saveAsAction);
51. fileMenu.addSeparator();
52.
53. fileMenu.add(new AbstractAction("Exit")
54.     {
55.         public void actionPerformed(ActionEvent event)
56.         {
57.             System.exit(0);
58.         }
59.     });
60.
61. // demonstrate check box and radio button menus
62.
63. readonlyItem = new JCheckBoxMenuItem("Read-only");
64. readonlyItem.addActionListener(new ActionListener()
65.     {
66.         public void actionPerformed(ActionEvent event)
67.         {
68.             boolean saveOk = !readonlyItem.isSelected();
69.             saveAction.setEnabled(saveOk);
70.             saveAsAction.setEnabled(saveOk);
71.         }
72.     });
73.
74. ButtonGroup group = new ButtonGroup();
75.
76. JRadioButtonMenuItem insertItem = new JRadioButtonMenuItem("Insert");
77. insertItem.setSelected(true);
78. JRadioButtonMenuItem overtypeItem = new JRadioButtonMenuItem("Overtype");
79.
80. group.add(insertItem);
81. group.add(overtypeItem);
82.
83. // demonstrate icons
84.
85. Action cutAction = new TestAction("Cut");
86. cutAction.putValue(Action.SMALL_ICON, new ImageIcon("cut.gif"));
87. Action copyAction = new TestAction("Copy");
88. copyAction.putValue(Action.SMALL_ICON, new ImageIcon("copy.gif"));
89. Action pasteAction = new TestAction("Paste");
90. pasteAction.putValue(Action.SMALL_ICON, new ImageIcon("paste.gif"));
91.
92. JMenu editMenu = new JMenu("Edit");
93. editMenu.add(cutAction);
```

**Listing 9-8** MenuTest.java (continued)

```
94.     editMenu.add(copyAction);
95.     editMenu.add(pasteAction);
96.
97.     // demonstrate nested menus
98.
99.     JMenu optionMenu = new JMenu("Options");
100.
101.     optionMenu.add(readonlyItem);
102.     optionMenu.addSeparator();
103.     optionMenu.add(insertItem);
104.     optionMenu.add(overtypItem);
105.
106.     editMenu.addSeparator();
107.     editMenu.add(optionMenu);
108.
109.     // demonstrate mnemonics
110.
111.     JMenu helpMenu = new JMenu("Help");
112.     helpMenu.setMnemonic('H');
113.
114.     JMenuItem indexItem = new JMenuItem("Index");
115.     indexItem.setMnemonic('I');
116.     helpMenu.add(indexItem);
117.
118.     // you can also add the mnemonic key to an action
119.     Action aboutAction = new TestAction("About");
120.     aboutAction.putValue(Action.MNEMONIC_KEY, new Integer('A'));
121.     helpMenu.add(aboutAction);
122.
123.     // add all top-level menus to menu bar
124.
125.     JMenuBar menuBar = new JMenuBar();
126.     setJMenuBar(menuBar);
127.
128.     menuBar.add(fileMenu);
129.     menuBar.add(editMenu);
130.     menuBar.add(helpMenu);
131.
132.     // demonstrate pop-ups
133.
134.     JPopupMenu popup = new JPopupMenu();
135.     popup.add(cutAction);
136.     popup.add(copyAction);
137.     popup.add(pasteAction);
138.
139.     JPanel panel = new JPanel();
140.     panel.setComponentPopupMenu(popup);
141.     add(panel);
142.
```

**Listing 9-8** MenuTest.java (continued)

```
143. // the following line is a workaround for bug 4966109
144. panel.addMouseListener(new MouseAdapter()
145.     {
146.         });
147. }
148.
149. public static final int DEFAULT_WIDTH = 300;
150. public static final int DEFAULT_HEIGHT = 200;
151.
152. private Action saveAction;
153. private Action saveAsAction;
154. private JCheckBoxMenuItem readonlyItem;
155. private JPopupMenu popup;
156. }
157.
158. /**
159.  * A sample action that prints the action name to System.out
160.  */
161. class TestAction extends AbstractAction
162. {
163.     public TestAction(String name)
164.     {
165.         super(name);
166.     }
167.
168.     public void actionPerformed(ActionEvent event)
169.     {
170.         System.out.println(getValue(Action.NAME) + " selected.");
171.     }
172. }
```

**Toolbars**

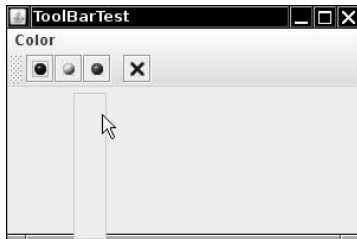
A toolbar is a button bar that gives quick access to the most commonly used commands in a program (see Figure 9-24).



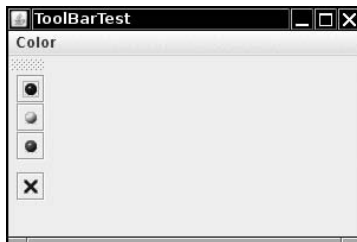
**Figure 9-24** A toolbar

What makes toolbars special is that you can move them elsewhere. You can drag the toolbar to one of the four borders of the frame (see Figure 9–25). When you release the mouse button, the toolbar is dropped into the new location (see Figure 9–26).

NOTE: Toolbar dragging works if the toolbar is inside a container with a border layout, or any other layout manager that supports the North, East, South, and West constraints.

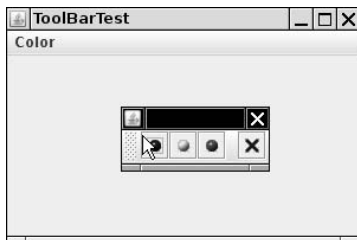


**Figure 9–25** Dragging the toolbar



**Figure 9–26** The toolbar has been dragged to another border

The toolbar can even be completely detached from the frame. A detached toolbar is contained in its own frame (see Figure 9–27). When you close the frame containing a detached toolbar, the toolbar jumps back into the original frame.



**Figure 9–27** Detaching the toolbar

Toolbars are straightforward to program. You add components into the toolbar:

```
JToolBar bar = new JToolBar();  
bar.add(blueButton);
```

The `JToolBar` class also has a method to add an `Action` object. Simply populate the toolbar with `Action` objects, like this:

```
bar.add(blueAction);
```

The small icon of the action is displayed in the toolbar.

You can separate groups of buttons with a separator:

```
bar.addSeparator();
```

For example, the toolbar in Figure 9–24 has a separator between the third and fourth button.

Then, you add the toolbar to the frame.

```
add(bar, BorderLayout.NORTH);
```

You can also specify a title for the toolbar that appears when the toolbar is undocked:

```
bar = new JToolBar(titleString);
```

By default, toolbars are initially horizontal. To have a toolbar start out as vertical, use

```
bar = new JToolBar(SwingConstants.VERTICAL)
```

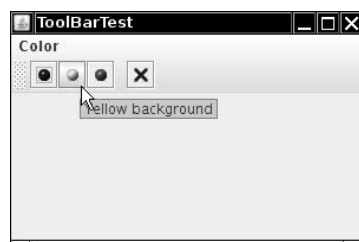
or

```
bar = new JToolBar(titleString, SwingConstants.VERTICAL)
```

Buttons are the most common components inside toolbars. But there is no restriction on the components that you can add to a toolbar. For example, you can add a combo box to a toolbar.

### Tooltips

A disadvantage of toolbars is that users are often mystified by the meanings of the tiny icons in toolbars. To solve this problem, user interface designers invented *tooltips*. A tooltip is activated when the cursor rests for a moment over a button. The tooltip text is displayed inside a colored rectangle. When the user moves the mouse away, the tooltip is removed. (See Figure 9–28.)



**Figure 9–28** A tooltip

In Swing, you can add tooltips to any `JComponent` simply by calling the `setToolTipText` method:

```
exitButton.setToolTipText("Exit");
```

Alternatively, if you use Action objects, you associate the tooltip with the `SHORT_DESCRIPTION` key:

```
exitAction.putValue(Action.SHORT_DESCRIPTION, "Exit");
```

Listing 9-9 shows how the same Action objects can be added to a menu and a toolbar. Note that the action names show up as the menu item names in the menu, and the short descriptions as the tooltips in the toolbar.

**Listing 9-9** ToolBarTest.java

```

1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.13 2007-06-12
7.  * @author Cay Horstmann
8.  */
9. public class ToolBarTest
10. {
11.     public static void main(String[] args)
12.     {
13.         EventQueue.invokeLater(new Runnable()
14.         {
15.             public void run()
16.             {
17.                 ToolBarFrame frame = new ToolBarFrame();
18.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.                 frame.setVisible(true);
20.             }
21.         });
22.     }
23. }
24.
25. /**
26.  * A frame with a toolbar and menu for color changes.
27.  */
28. class ToolBarFrame extends JFrame
29. {
30.     public ToolBarFrame()
31.     {
32.         setTitle("ToolBarTest");
33.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34.
35.         // add a panel for color change
36.
37.         panel = new JPanel();
38.         add(panel, BorderLayout.CENTER);
39.
40.         // set up actions
41.

```

**Listing 9-9** ToolBarTest.java (continued)

```
42. Action blueAction = new ColorAction("Blue", new ImageIcon("blue-ball.gif"), Color.BLUE);
43. Action yellowAction = new ColorAction("Yellow", new ImageIcon("yellow-ball.gif"),
44.     Color.YELLOW);
45. Action redAction = new ColorAction("Red", new ImageIcon("red-ball.gif"), Color.RED);
46.
47. Action exitAction = new AbstractAction("Exit", new ImageIcon("exit.gif"))
48.     {
49.     public void actionPerformed(ActionEvent event)
50.     {
51.         System.exit(0);
52.     }
53. };
54. exitAction.putValue(Action.SHORT_DESCRIPTION, "Exit");
55.
56. // populate tool bar
57.
58. JToolBar bar = new JToolBar();
59. bar.add(blueAction);
60. bar.add(yellowAction);
61. bar.add(redAction);
62. bar.addSeparator();
63. bar.add(exitAction);
64. add(bar, BorderLayout.NORTH);
65.
66. // populate menu
67.
68. JMenu menu = new JMenu("Color");
69. menu.add(yellowAction);
70. menu.add(blueAction);
71. menu.add(redAction);
72. menu.add(exitAction);
73. JMenuBar menuBar = new JMenuBar();
74. menuBar.add(menu);
75. setJMenuBar(menuBar);
76. }
77.
78. public static final int DEFAULT_WIDTH = 300;
79. public static final int DEFAULT_HEIGHT = 200;
80.
81. private JPanel panel;
82.
83. /**
84.  * The color action sets the background of the frame to a given color.
85.  */
86. class ColorAction extends AbstractAction
87. {
88.     public ColorAction(String name, Icon icon, Color c)
89.     {
```

**Listing 9-9** `ToolBarTest.java` (continued)

```

90.     putValue(Action.NAME, name);
91.     putValue(Action.SMALL_ICON, icon);
92.     putValue(Action.SHORT_DESCRIPTION, name + " background");
93.     putValue("Color", c);
94. }
95.
96. public void actionPerformed(ActionEvent event)
97. {
98.     Color c = (Color) getValue("Color");
99.     panel.setBackground(c);
100. }
101. }
102. }

```

**API** `javax.swing.JToolBar` 1.2

- `JToolBar()`
- `JToolBar(String titleString)`
- `JToolBar(int orientation)`
- `JToolBar(String titleString, int orientation)`  
constructs a toolbar with the given title string and orientation. `orientation` is one of `SwingConstants.HORIZONTAL` (the default) and `SwingConstants.VERTICAL`.
- `JButton add(Action a)`  
constructs a new button inside the toolbar with name, icon, short description, and action callback from the given action, and adds the button to the end of the toolbar.
- `void addSeparator()`  
adds a separator to the end of the toolbar.

**API** `javax.swing.JComponent` 1.2

- `void setToolTipText(String text)`  
sets the text that should be displayed as a tooltip when the mouse hovers over the component.

**Sophisticated Layout Management**

We have managed to lay out the user interface components of our sample applications so far by using only the border layout, flow layout, and grid layout. For more complex tasks, this is not going to be enough. In this section, we discuss advanced layout management in detail.

Windows programmers may well wonder why Java makes so much fuss about layout managers. After all, in Windows, layout management is not a big deal: First, you use a dialog editor to drag and drop your components onto the surface of a dialog, and then you use editor tools to line up components, to space them equally, to center them, and so on. If you are working on a big project, you probably don't have to worry about component layout at all—a skilled user interface designer does all this for you.



The problem with this approach is that the resulting layout must be manually updated if the size of the components changes. Why would the component size change? There are two common cases. First, a user may choose a larger font for button labels and other dialog text. If you try this out for yourself in Windows, you will find that many applications deal with this exceedingly poorly. The buttons do not grow, and the larger font is simply crammed into the same space as before. The same problem can occur when the strings in an application are translated to a foreign language. For example, the German word for “Cancel” is “Abbrechen.” If a button has been designed with just enough room for the string “Cancel”, then the German version will look broken, with a clipped command string.

Why don’t Windows buttons simply grow to accommodate the labels? Because the designer of the user interface gave no instructions in which direction they should grow. After the dragging and dropping and arranging, the dialog editor merely remembers the pixel position and size of each component. It does not remember *why* the components were arranged in this fashion.

The Java layout managers are a much better approach to component layout. With a layout manager, the layout comes with instructions about the relationships among the components. This was particularly important in the original AWT, which used native user interface elements. The size of a button or list box in Motif, Windows, and the Macintosh could vary widely, and an application or applet would not know *a priori* on which platform it would display its user interface. To some extent, that degree of variability has gone away with Swing. If your application forces a particular look and feel, such as the Metal look and feel, then it looks identical on all platforms. However, if you let users of your application choose their favorite look and feel, then you again need to rely on the flexibility of layout managers to arrange the components.

Since Java 1.0, the AWT includes the *grid bag layout* that lays out components in rows and columns. The row and column sizes are flexible and components can span multiple rows and columns. This layout manager is very flexible, but it is also very complex. The mere mention of the words “grid bag layout” has been known to strike fear in the hearts of Java programmers.

In an unsuccessful attempt to design a layout manager that would free programmers from the tyranny of the grid bag layout, the Swing designers came up with the *box layout*. According to the JDK documentation of the `BoxLayout` class: “Nesting multiple panels with different combinations of horizontal and vertical [*sic*] gives an effect similar to Grid-Bag layout, without the complexity.” However, because each box is laid out independently, you cannot use box layouts to arrange neighboring components both horizontally and vertically.

Java SE 1.4 saw yet another attempt to design a replacement for the grid bag layout—the *spring layout*. You use imaginary springs to connect the components in a container. As the container is resized, the springs stretch or shrink, thereby adjusting the positions of the components. This sounds tedious and confusing, and it is. The spring layout quickly sank into obscurity.

In 2005, the NetBeans team invented the Matisse technology, which combines a layout tool and a layout manager. A user interface designer uses the tool to drop components into a container and to indicate which components should line up. The tool translates

the designer's intentions into instructions for the *group layout manager*. This is much more convenient than writing layout management code by hand. The group layout manager is now a part of Java SE 6. Even if you don't use NetBeans as your IDE, we think you should consider using its GUI builder tool. You can design your GUI in NetBeans and paste the resulting code into your IDE of choice.

In the coming sections, we cover the grid bag layout because it is commonly used and is still the easiest mechanism for producing layout code for older Java versions. We will tell you a strategy that makes grid bag layouts relatively painless in common situations.

Next, we cover the Matisse tool and the group layout manager. You will want to know how the group layout manager works so that you can check whether Matisse recorded the correct instructions when you visually positioned your components.

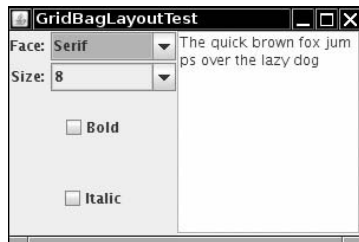
We end the discussion of layout managers by showing you how you can bypass layout management altogether and place components manually, and how you can write your own layout manager.

### The Grid Bag Layout

The grid bag layout is the mother of all layout managers. You can think of a grid bag layout as a grid layout without the limitations. In a grid bag layout, the rows and columns can have variable sizes. You can join adjacent cells to make room for larger components. (Many word processors, as well as HTML, have the same capability when tables are edited: you start out with a grid and then merge adjacent cells if need be.) The components need not fill the entire cell area, and you can specify their alignment within cells.

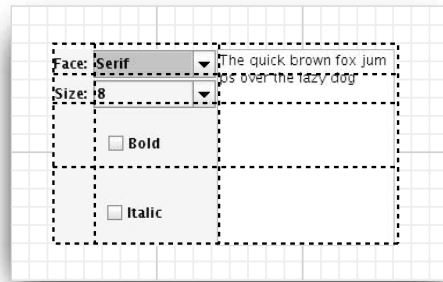
Consider the font selector of Figure 9–29. It consists of the following components:

- Two combo boxes to specify the font face and size
- Labels for these two combo boxes
- Two checkboxes to select bold and italic
- A text area for the sample string



**Figure 9–29** A font selector

Now, chop up the container into a grid of cells, as shown in Figure 9–30. (The rows and columns need not have equal size.) Each checkbox spans two columns, and the text area spans four rows.



**Figure 9-30** Dialog box grid used in design

To describe the layout to the grid bag manager, use the following procedure:

1. Create an object of type `GridBagLayout`. You don't tell it how many rows and columns the underlying grid has. Instead, the layout manager will try to guess it from the information you give it later.
2. Set this `GridBagLayout` object to be the layout manager for the component.
3. For each component, create an object of type `GridBagConstraints`. Set field values of the `GridBagConstraints` object to specify how the components are laid out within the grid bag.
4. Finally, add each component with its constraints by using the call  
`add(component, constraints);`

Here's an example of the code needed. (We go over the various constraints in more detail in the sections that follow—so don't worry if you don't know what some of the constraints do.)

```
GridBagLayout layout = new GridBagLayout();
panel.setLayout(layout);
GridBagConstraints constraints = new GridBagConstraints();
constraints.weightx = 100;
constraints.weighty = 100;
constraints.gridx = 0;
constraints.gridy = 2;
constraints.gridwidth = 2;
constraints.gridheight = 1;
panel.add(component, constraints);
```

The trick is knowing how to set the state of the `GridBagConstraints` object. We go over the most important constraints for using this object in the sections that follow.

#### **The `gridx`, `gridy`, `gridwidth`, and `gridheight` Parameters**

The `gridx`, `gridy`, `gridwidth`, and `gridheight` constraints define where the component is located in the grid. The `gridx` and `gridy` values specify the column and row positions of the upper-left corner of the component to be added. The `gridwidth` and `gridheight` values determine how many columns and rows the component occupies.

The grid coordinates start with 0. In particular, `gridx = 0` and `gridy = 0` denotes the top-left corner. For example, the text area in our example has `gridx = 2`, `gridy = 0` because it starts

in column 2 (that is, the third column) of row 0. It has `gridwidth = 1` and `gridheight = 4` because it spans one column and four rows.

### Weight Fields

You always need to set the *weight* fields (`weightx` and `weighty`) for each area in a grid bag layout. If you set the weight to 0, then the area never grows or shrinks beyond its initial size in that direction. In the grid bag layout for Figure 9–29 on page 426, we set the `weightx` field of the labels to be 0. This allows the labels to remain a constant width when you resize the window. On the other hand, if you set the weights for all areas to 0, the container will huddle in the center of its allotted area rather than stretching to fill it.

Conceptually, the problem with the weight parameters is that weights are properties of rows and columns, not individual cells. But you need to specify them in terms of cells because the grid bag layout does not expose the rows and columns. The row and column weights are computed as the maxima of the cell weights in each row or column. Thus, if you want a row or column to stay at a fixed size, you need to set the weights of all components in it to zero.

Note that the weights don't actually give the relative sizes of the columns. They tell what proportion of the "slack" space should be allocated to each area if the container exceeds its preferred size. This isn't particularly intuitive. We recommend that you set all weights at 100. Then, run the program and see how the layout looks. Resize the dialog to see how the rows and columns adjust. If you find that a particular row or column should not grow, set the weights of all components in it to zero. You can tinker with other weight values, but it is usually not worth the effort.

### The fill and anchor Parameters

If you don't want a component to stretch out and fill the entire area, you set the `fill` constraint. You have four possibilities for this parameter: the valid values are used in the forms `GridBagConstraints.NONE`, `GridBagConstraints.HORIZONTAL`, `GridBagConstraints.VERTICAL`, and `GridBagConstraints.BOTH`.

If the component does not fill the entire area, you can specify where in the area you want it by setting the `anchor` field. The valid values are `GridBagConstraints.CENTER` (the default), `GridBagConstraints.NORTH`, `GridBagConstraints.NORTHEAST`, `GridBagConstraints.EAST`, and so on.

### Padding

You can surround a component with additional blank space by setting the `insets` field of `GridBagConstraints`. Set the `left`, `top`, `right` and `bottom` values of the `Insets` object to the amount of space that you want to have around the component. This is called the *external padding*.

The `ipadx` and `ipady` values set the *internal padding*. These values are added to the minimum width and height of the component. This ensures that the component does not shrink down to its minimum size.

### Alternative Method to Specify the gridx, gridy, gridwidth, and gridheight Parameters

The AWT documentation recommends that instead of setting the `gridx` and `gridy` values to absolute positions, you set them to the constant `GridBagConstraints.RELATIVE`. Then, add the components to the grid bag layout in a standardized order, going from left to right in the first row, then moving along the next row, and so on.

You still specify the number of rows and columns spanned, by giving the appropriate `gridheight` and `gridwidth` fields. Except, if the component extends to the *last* row or column, you aren't supposed to specify the actual number, but the constant `GridBagConstraints.REMAINDER`. This tells the layout manager that the component is the last one in its row.

This scheme does seem to work. But it sounds really goofy to hide the actual placement information from the layout manager and hope that it will rediscover it.

All this sounds like a lot of trouble and complexity. But in practice, the strategy in the following recipe makes grid bag layouts relatively trouble-free:

1. Sketch out the component layout on a piece of paper.
2. Find a grid such that the small components are each contained in a cell and the larger components span multiple cells.
3. Label the rows and columns of your grid with 0, 1, 2, 3, . . . You can now read off the `gridx`, `gridy`, `gridwidth`, and `gridheight` values.
4. For each component, ask yourself whether it needs to fill its cell horizontally or vertically. If not, how do you want it aligned? This tells you the `fill` and `anchor` parameters.
5. Set all weights to 100. However, if you want a particular row or column to always stay at its default size, set the `weightx` or `weighty` to 0 in all components that belong to that row or column.
6. Write the code. Carefully double-check your settings for the `GridBagConstraints`. One wrong constraint can ruin your whole layout.
7. Compile, run, and enjoy.

Some GUI builders even have tools for specifying the constraints visually—see Figure 9–31 for the configuration dialog in NetBeans.

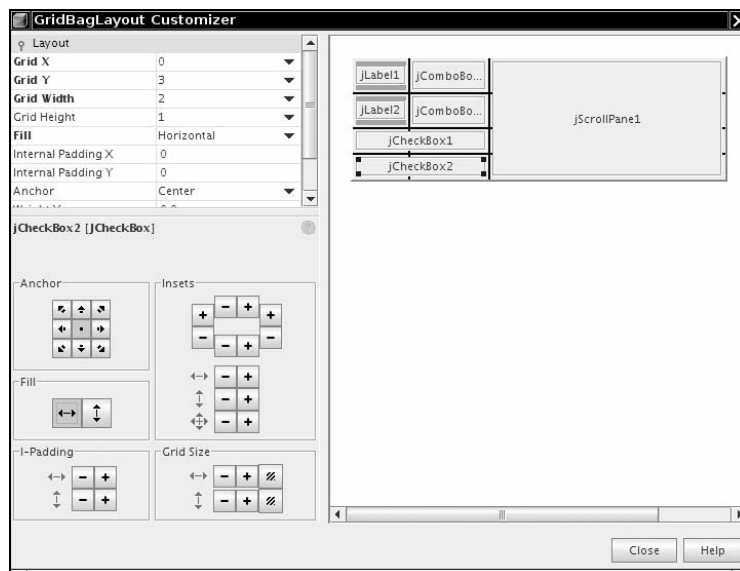


Figure 9–31 Specifying grid bag constraints in NetBeans

### A Helper Class to Tame the Grid Bag Constraints

The most tedious aspect of the grid bag layout is writing the code that sets the constraints. Most programmers write helper functions or a small helper class for this purpose. We present such a class after the complete code for the font dialog example. This class has the following features:

- Its name is short: `GBC` instead of `GridBagConstraints`.
- It extends `GridBagConstraints`, so you can use shorter names such as `GBC.EAST` for the constants.
- Use a `GBC` object when adding a component, such as  
`add(component, new GBC(1, 2));`
- There are two constructors to set the most common parameters: `gridx` and `gridy`, or `gridx`, `gridy`, `gridwidth`, and `gridheight`.  
`add(component, new GBC(1, 2, 1, 4));`
- There are convenient setters for the fields that come in `x/y` pairs:  
`add(component, new GBC(1, 2).setWeight(100, 100));`
- The setter methods return `this`, so you can chain them:  
`add(component, new GBC(1, 2).setAnchor(GBC.EAST).setWeight(100, 100));`
- The `setInsets` methods construct the `Insets` object for you. To get one-pixel insets, simply call  
`add(component, new GBC(1, 2).setAnchor(GBC.EAST).setInsets(1));`

Listing 9–10 shows the complete code for the font dialog example. Here is the code that adds the components to the grid bag:

```
add(faceLabel, new GBC(0, 0).setAnchor(GBC.EAST));
add(face, new GBC(1, 0).setFill(GBC.HORIZONTAL).setWeight(100, 0).setInsets(1));
add(sizeLabel, new GBC(0, 1).setAnchor(GBC.EAST));
add(size, new GBC(1, 1).setFill(GBC.HORIZONTAL).setWeight(100, 0).setInsets(1));
add(italic, new GBC(0, 2, 2, 1).setAnchor(GBC.CENTER).setWeight(100, 100));
add(bold, new GBC(0, 3, 2, 1).setAnchor(GBC.CENTER).setWeight(100, 100));
add(sample, new GBC(2, 0, 1, 4).setFill(GBC.BOTH).setWeight(100, 100));
```

Once you understand the grid bag constraints, this kind of code is fairly easy to read and debug.



**NOTE:** The Sun tutorial at <http://java.sun.com/docs/books/tutorial/uiwing/layout/grid-bag.html> suggests that you reuse the same `GridBagConstraints` object for all components. We find the resulting code hard to read and error prone. For example, look at the demo at <http://java.sun.com/docs/books/tutorial/uiwing/events/containerlistener.html>. Was it really intended that the buttons are stretched horizontally, or did the programmer just forget to turn off the fill constraint?

**Listing 9-10** GridBagLayoutTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.33 2007-06-12
7.  * @author Cay Horstmann
8.  */
9. public class GridBagLayoutTest
10. {
11.     public static void main(String[] args)
12.     {
13.         EventQueue.invokeLater(new Runnable()
14.         {
15.             public void run()
16.             {
17.                 FontFrame frame = new FontFrame();
18.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.                 frame.setVisible(true);
20.             }
21.         });
22.     }
23. }
24.
25. /**
26.  * A frame that uses a grid bag layout to arrange font selection components.
27.  */
28. class FontFrame extends JFrame
29. {
30.     public FontFrame()
31.     {
32.         setTitle("GridBagLayoutTest");
33.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34.
35.         GridBagLayout layout = new GridBagLayout();
36.         setLayout(layout);
37.
38.         ActionListener listener = new FontAction();
39.
40.         // construct components
41.
42.         JLabel faceLabel = new JLabel("Face: ");
43.
44.         face = new JComboBox(new String[] { "Serif", "SansSerif", "Monospaced", "Dialog",
45.             "DialogInput" });
46.
47.         face.addActionListener(listener);
48.
49.         JLabel sizeLabel = new JLabel("Size: ");
50.
51.         size = new JComboBox(new String[] { "8", "10", "12", "15", "18", "24", "36", "48" });
```

**Listing 9-10** GridBagLayoutTest.java (continued)

```
52.
53.     size.addActionListener(listener);
54.
55.     bold = new JCheckBox("Bold");
56.     bold.addActionListener(listener);
57.
58.     italic = new JCheckBox("Italic");
59.     italic.addActionListener(listener);
60.
61.     sample = new JTextArea();
62.     sample.setText("The quick brown fox jumps over the lazy dog");
63.     sample.setEditable(false);
64.     sample.setLineWrap(true);
65.     sample.setBorder(BorderFactory.createEtchedBorder());
66.
67.     // add components to grid, using GBC convenience class
68.
69.     add(faceLabel, new GBC(0, 0).setAnchor(GBC.EAST));
70.     add(face, new GBC(1, 0).setFill(GBC.HORIZONTAL).setWeight(100, 0).setInsets(1));
71.     add(sizeLabel, new GBC(0, 1).setAnchor(GBC.EAST));
72.     add(size, new GBC(1, 1).setFill(GBC.HORIZONTAL).setWeight(100, 0).setInsets(1));
73.     add(bold, new GBC(0, 2, 2, 1).setAnchor(GBC.CENTER).setWeight(100, 100));
74.     add(italic, new GBC(0, 3, 2, 1).setAnchor(GBC.CENTER).setWeight(100, 100));
75.     add(sample, new GBC(2, 0, 1, 4).setFill(GBC.BOTH).setWeight(100, 100));
76. }
77.
78. public static final int DEFAULT_WIDTH = 300;
79. public static final int DEFAULT_HEIGHT = 200;
80.
81. private JComboBox face;
82. private JComboBox size;
83. private JCheckBox bold;
84. private JCheckBox italic;
85. private JTextArea sample;
86.
87. /**
88.  * An action listener that changes the font of the sample text.
89.  */
90. private class FontAction implements ActionListener
91. {
92.     public void actionPerformed(ActionEvent event)
93.     {
94.         String fontFace = (String) face.getSelectedItem();
95.         int fontStyle = (bold.isSelected() ? Font.BOLD : 0)
96.             + (italic.isSelected() ? Font.ITALIC : 0);
97.         int fontSize = Integer.parseInt((String) size.getSelectedItem());
98.         Font font = new Font(fontFace, fontStyle, fontSize);
99.         sample.setFont(font);
100.        sample.repaint();
101.    }
102. }
103. }
```



Listing 9–11 shows the code of the GBC helper class.

**Listing 9–11** GBC.java

```
1. import java.awt.*;
2.
3. /**
4.  * This class simplifies the use of the GridBagConstraints class.
5.  * @version 1.01 2004-05-06
6.  * @author Cay Horstmann
7.  */
8. public class GBC extends GridBagConstraints
9. {
10.     /**
11.      * Constructs a GBC with a given gridx and gridy position and all other grid
12.      * bag constraint values set to the default.
13.      * @param gridx the gridx position
14.      * @param gridy the gridy position
15.      */
16.     public GBC(int gridx, int gridy)
17.     {
18.         this.gridx = gridx;
19.         this.gridy = gridy;
20.     }
21.
22.     /**
23.      * Constructs a GBC with given gridx, gridy, gridwidth, gridheight and all
24.      * other grid bag constraint values set to the default.
25.      * @param gridx the gridx position
26.      * @param gridy the gridy position
27.      * @param gridwidth the cell span in x-direction
28.      * @param gridheight the cell span in y-direction
29.      */
30.     public GBC(int gridx, int gridy, int gridwidth, int gridheight)
31.     {
32.         this.gridx = gridx;
33.         this.gridy = gridy;
34.         this.gridwidth = gridwidth;
35.         this.gridheight = gridheight;
36.     }
37.
38.     /**
39.      * Sets the anchor.
40.      * @param anchor the anchor value
41.      * @return this object for further modification
42.      */
43.     public GBC setAnchor(int anchor)
44.     {
45.         this.anchor = anchor;
46.         return this;
47.     }
}
```

**Listing 9-11** GBC.java (continued)

```
48.
49.  /**
50.   * Sets the fill direction.
51.   * @param fill the fill direction
52.   * @return this object for further modification
53.   */
54. public GBC setFill(int fill)
55. {
56.     this.fill = fill;
57.     return this;
58. }
59.
60. /**
61.  * Sets the cell weights.
62.  * @param weightx the cell weight in x-direction
63.  * @param weighty the cell weight in y-direction
64.  * @return this object for further modification
65.  */
66. public GBC setWeight(double weightx, double weighty)
67. {
68.     this.weightx = weightx;
69.     this.weighty = weighty;
70.     return this;
71. }
72.
73. /**
74.  * Sets the insets of this cell.
75.  * @param distance the spacing to use in all directions
76.  * @return this object for further modification
77.  */
78. public GBC setInsets(int distance)
79. {
80.     this.insets = new Insets(distance, distance, distance, distance);
81.     return this;
82. }
83.
84. /**
85.  * Sets the insets of this cell.
86.  * @param top the spacing to use on top
87.  * @param left the spacing to use to the left
88.  * @param bottom the spacing to use on the bottom
89.  * @param right the spacing to use to the right
90.  * @return this object for further modification
91.  */
92. public GBC setInsets(int top, int left, int bottom, int right)
93. {
94.     this.insets = new Insets(top, left, bottom, right);
95.     return this;
96. }
97.
```

**Listing 9-11** GBC.java (continued)

```

98.  /**
99.   * Sets the internal padding
100.  * @param ipadx the internal padding in x-direction
101.  * @param ipady the internal padding in y-direction
102.  * @return this object for further modification
103.  */
104.  public GBC setIpad(int ipadx, int ipady)
105.  {
106.      this.ipadx = ipadx;
107.      this.ipady = ipady;
108.      return this;
109.  }
110. }
```

**API** java.awt.GridBagConstraints 1.0

- `int gridx, gridy`  
specifies the starting column and row of the cell. The default is 0.
- `int gridwidth, gridheight`  
specifies the column and row extent of the cell. The default is 1.
- `double weightx, weighty`  
specifies the capacity of the cell to grow. The default is 0.
- `int anchor`  
indicates the alignment of the component inside the cell. You can choose between absolute positions:

NORTHWEST	NORTH	NORTHEAST
WEST	CENTER	EAST
SOUTHWEST	SOUTH	SOUTHEAST

or their orientation-independent counterparts:

FIRST_LINE_START	LINE_START	FIRST_LINE_END
PAGE_START	CENTER	PAGE_END
LAST_LINE_START	LINE_END	LAST_LINE_END

Use the latter if your application may be localized for right-to-left or top-to-bottom text. The default is CENTER.

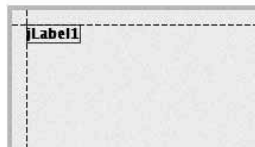
- `int fill`  
specifies the fill behavior of the component inside the cell, one of NONE, BOTH, HORIZONTAL, or VERTICAL. The default is NONE.
- `int ipadx, ipady`  
specifies the “internal” padding around the component. The default is 0.

- Insets insets specifies the “external” padding along the cell boundaries. The default is no padding.
- GridBagConstraints(int gridx, int gridy, int gridwidth, int gridheight, double weightx, double weighty, int anchor, int fill, Insets insets, int ipadx, int ipady) **1.2** constructs a GridBagConstraints with all its fields specified in the arguments. Sun recommends that this constructor be used only by automatic code generators because it makes your source code very hard to read.

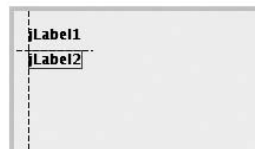
### Group Layout

Before discussing the API of the GroupLayout class, let us have a quick look at the Matisse GUI builder in NetBeans. We won’t give you a full Matisse tutorial—see <http://www.netbeans.org/kb/articles/matisse.html> for more information.

Here is the workflow for laying out the top of the dialog in Figure 9–13. Start a new project and add a new JFrame form. Drag a label until two guidelines appear that separate it from the container borders:



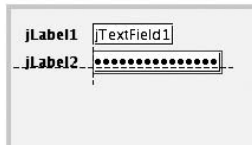
Place another label below the first row:



Drag a text field so that its baseline lines up with the baseline of the first label. Again, note the guidelines:



Finally, line up a password field with the label to the left and the text field above.



Matisse translates these actions into the following Java code:

```

layout.setHorizontalGroup(
    layout.createParallelGroup(GroupLayout.Alignment.LEADING)
    .addGroup(layout.createSequentialGroup()
        .addContainerGap()
        .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
            .addGroup(layout.createSequentialGroup()
                .addComponent(jLabel1)
                .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
                .addComponent(jTextField1))
            .addGroup(layout.createSequentialGroup()
                .addComponent(jLabel2)
                .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
                .addComponent(jPasswordField1)))
        .addContainerGap(222, Short.MAX_VALUE));
layout.setVerticalGroup(
    layout.createParallelGroup(GroupLayout.Alignment.LEADING)
    .addGroup(layout.createSequentialGroup()
        .addContainerGap()
        .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
            .addComponent(jLabel1)
            .addComponent(jTextField1))
        .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
        .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
            .addComponent(jLabel2)
            .addComponent(jPasswordField1))
        .addContainerGap(244, Short.MAX_VALUE));

```

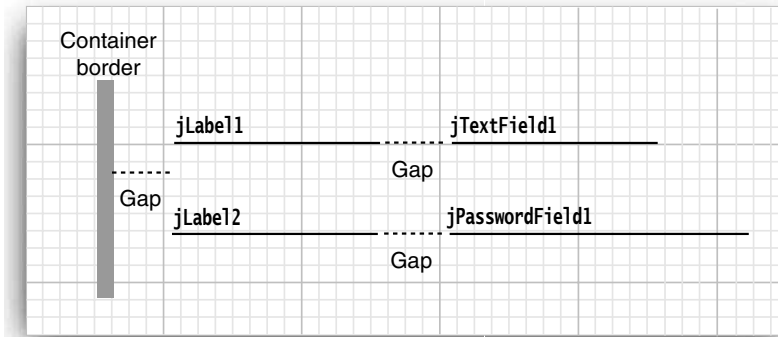
That looks a bit scary, but fortunately you don't have to write the code. However, it is helpful to have a basic understanding of the layout actions so that you can spot errors. We will analyze the basic structure of the code. The API notes at the end of this section explain each of the classes and methods in detail.

Components are organized by placing them into objects of type `GroupLayout.SequentialGroup` or `GroupLayout.ParallelGroup`. These classes are subclasses of `GroupLayout.Group`. Groups can contain components, gaps, and nested groups. The various `add` methods of the group classes return the group object so that method calls can be chained, like this:

```
group.addComponent(...).addPreferredGap(...).addComponent(...);
```

As you can see from the sample code, the group layout separates the horizontal and vertical layout computations.

To visualize the horizontal computations, imagine that the components are flattened so they have zero height, like this:



There are two parallel sequences of components, corresponding to the (slightly simplified) code:

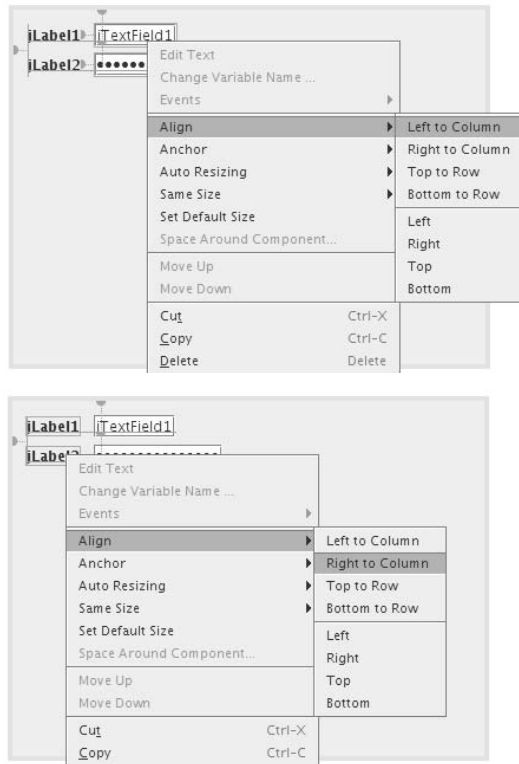
```
.addContainerGap()
.addGroup(layout.createParallelGroup()
    .addGroup(layout.createSequentialGroup()
        .addComponent(jLabel1)
        .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
        .addComponent(jTextField1))
    .addGroup(layout.createSequentialGroup()
        .addComponent(jLabel2)
        .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
        .addComponent(jPasswordField1)))
```

But wait, that can't be right. If the labels have different lengths, the text field and the password field won't line up.

We have to tell Matisse that we want the fields to line up. Select both fields, right-click, and select Align -> Left to Column from the menu. Also line up the labels (see Figure 9-32).

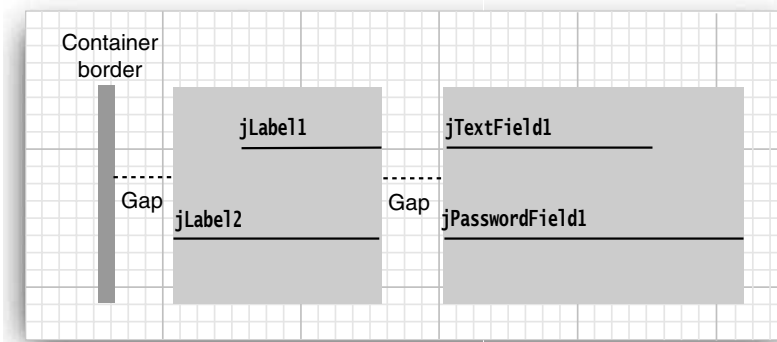
This dramatically changes the layout code:

```
.addGroup(layout.createSequentialGroup()
    .addContainerGap()
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
        .addComponent(jLabel1, GroupLayout.Alignment.TRAILING)
        .addComponent(jLabel2, GroupLayout.Alignment.TRAILING))
    .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
        .addComponent(jTextField1)
        .addComponent(jPasswordField1)))
```



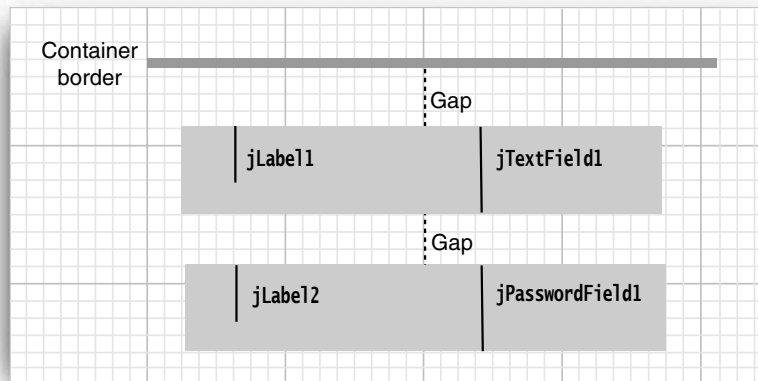
**Figure 9-32** Aligning the labels and text fields in Matisse

Now the labels and fields are each placed in a parallel group. The first group has an alignment of TRAILING (which means alignment to the right when the text direction is left-to-right):



It seems like magic that Matisse can translate the designer's instructions into nested groups, but as Arthur C. Clarke said, any sufficiently advanced technology is indistinguishable from magic.

For completeness, let's look at the vertical computation. Now you should think of the components as having no width. We have a sequential group that contains two parallel groups, separated by gaps:



The corresponding code is

```
layout.createSequentialGroup()
    .addContainerGap()
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
        .addComponent(jLabel1)
        .addComponent(jTextField1))
    .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
    .addGroup(layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
        .addComponent(jLabel12)
        .addComponent(jPasswordField1))
```

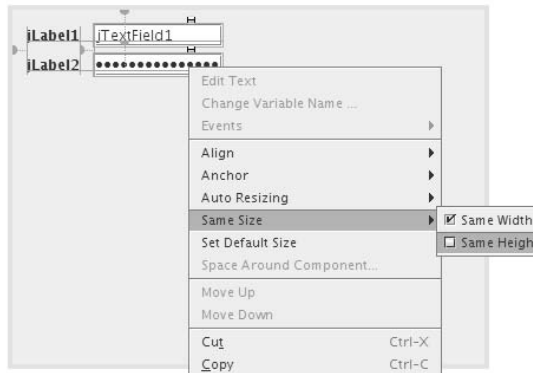
As you can see from the code, the components are aligned by their baselines. (The baseline is the line on which the component text is aligned.)



**NOTE:** Accurate baseline alignment was not possible in earlier versions of Java. Finally, Java SE 6 added a `getBaseline` method to the `Component` class for determining the exact baseline of a component containing text.

You can force a set of components to have equal size. For example, we may want to make sure that the text field and password field width match exactly. In Matisse, select both, right-click, and select Same Size -> Same Width from the menu (see Figure 9-33).





**Figure 9-33** Forcing two components to have the same width

Matisse adds the following statement to the layout code:

```
layout.linkSize(SwingConstants.HORIZONTAL, new Component[] {jPasswordField1, jTextField1});
```

The code in Listing 9-12 shows how to lay out the font selector of the preceding section, using the `GroupLayout` instead of the `GridBagLayout`. The code may not look any simpler than that of Listing 9-10 on page 431, but we didn't have to write it. We used Matisse to do the layout and then cleaned up the code a bit.

**Listing 9-12** `GroupLayoutTest.java`

```

1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.0 2007-04-27
7.  * @author Cay Horstmann
8.  */
9. public class GroupLayoutTest
10. {
11.     public static void main(String[] args)
12.     {
13.        .EventQueue.invokeLater(new Runnable()
14.         {
15.             public void run()
16.             {
17.                 FontFrame frame = new FontFrame();
18.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.                 frame.setVisible(true);
20.             }
21.         });
22.     }
23. }

```

**Listing 9–12** GroupLayoutTest.java (continued)

```
24.
25. /**
26.  * A frame that uses a group layout to arrange font selection components.
27.  */
28. class FontFrame extends JFrame
29. {
30.     public FontFrame()
31.     {
32.         setTitle("GroupLayoutTest");
33.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34.
35.         ActionListener listener = new FontAction();
36.
37.         // construct components
38.
39.         JLabel faceLabel = new JLabel("Face: ");
40.
41.         face = new JComboBox(new String[] { "Serif", "SansSerif", "Monospaced", "Dialog",
42.             "DialogInput" });
43.
44.         face.addActionListener(listener);
45.
46.         JLabel sizeLabel = new JLabel("Size: ");
47.
48.         size = new JComboBox(new String[] { "8", "10", "12", "15", "18", "24", "36", "48" });
49.
50.         size.addActionListener(listener);
51.
52.         bold = new JCheckBox("Bold");
53.         bold.addActionListener(listener);
54.
55.         italic = new JCheckBox("Italic");
56.         italic.addActionListener(listener);
57.
58.         sample = new JTextArea();
59.         sample.setText("The quick brown fox jumps over the lazy dog");
60.         sample.setEditable(false);
61.         sample.setLineWrap(true);
62.         sample.setBorder(BorderFactory.createEtchedBorder());
63.
64.         pane = new JScrollPane(sample);
65.
66.         GroupLayout layout = new GroupLayout(getContentPane());
67.         setLayout(layout);
68.         layout.setHorizontalGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
69.             .addGroup(
70.                 layout.createSequentialGroup().addContainerGap().addGroup(
71.                     layout.createParallelGroup(GroupLayout.Alignment.LEADING).addGroup(
```

**Listing 9-12** GroupLayoutTest.java (continued)

```

72.             GroupLayout.Alignment.TRAILING,
73.             layout.createSequentialGroup().addGroup(
74.                 layout.createParallelGroup(GroupLayout.Alignment.TRAILING)
75.                     .addComponent(faceLabel).addComponent(sizeLabel))
76.                 .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
77.                 .addGroup(
78.                     layout.createParallelGroup(
79.                         GroupLayout.Alignment.LEADING, false)
80.                         .addComponent(size).addComponent(face)))
81.                 .addComponent(italic).addComponent(bold).addPreferredGap(
82.                     LayoutStyle.ComponentPlacement.RELATED).addComponent(pane)
83.                 .addContainerGap());
84.
85.         layout.linkSize(SwingConstants.HORIZONTAL, new java.awt.Component[] { face, size });
86.
87.         layout.setVerticalGroup(layout.createParallelGroup(GroupLayout.Alignment.LEADING)
88.             .addGroup(
89.                 layout.createSequentialGroup().addContainerGap().addGroup(
90.                     layout.createParallelGroup(GroupLayout.Alignment.LEADING).addComponent(
91.                         pane, GroupLayout.Alignment.TRAILING).addGroup(
92.                             layout.createSequentialGroup().addGroup(
93.                                 layout.createParallelGroup(GroupLayout.Alignment.BASELINE)
94.                                     .addComponent(face).addComponent(faceLabel))
95.                                 .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
96.                                 .addGroup(
97.                                     layout.createParallelGroup(
98.                                         GroupLayout.Alignment.BASELINE).addComponent(size)
99.                                         .addComponent(sizeLabel)).addPreferredGap(
100.                                             LayoutStyle.ComponentPlacement.RELATED).addComponent(
101.                                                 italic, GroupLayout.DEFAULT_SIZE,
102.                                                 GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
103.                                             .addPreferredGap(LayoutStyle.ComponentPlacement.RELATED)
104.                                             .addComponent(bold, GroupLayout.DEFAULT_SIZE,
105.                                                 GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)))
106.                             .addContainerGap()));
107.         }
108.
109.         public static final int DEFAULT_WIDTH = 300;
110.         public static final int DEFAULT_HEIGHT = 200;
111.
112.         private JComboBox face;
113.         private JComboBox size;
114.         private JCheckBox bold;
115.         private JCheckBox italic;
116.         private JScrollPane pane;
117.         private JTextArea sample;
118.
119.         /**
120.          * An action listener that changes the font of the sample text.
121.          */

```

**Listing 9–12** GroupLayoutTest.java (continued)

```

122. private class FontAction implements ActionListener
123. {
124.     public void actionPerformed(ActionEvent event)
125.     {
126.         String fontFace = (String) face.getSelectedItem();
127.         int fontStyle = (bold.isSelected() ? Font.BOLD : 0)
128.             + (italic.isSelected() ? Font.ITALIC : 0);
129.         int fontSize = Integer.parseInt((String) size.getSelectedItem());
130.         Font font = new Font(fontFace, fontStyle, fontSize);
131.         sample.setFont(font);
132.         sample.repaint();
133.     }
134. }
135. }

```

**API** javax.swing.GroupLayout 6

- GroupLayout(Container host)  
constructs a GroupLayout for laying out the components in the host container. (Note that you still need to call `setLayout` on the host object.)
  - void setHorizontalGroup(GroupLayout.Group g)
  - void setVerticalGroup(GroupLayout.Group g)  
sets the group that controls horizontal or vertical layout.
  - void linkSize(Component... components)
  - void linkSize(int axis, Component... component)  
forces the given components to have the same size, or the same size along the given axis (one of `SwingConstants.HORIZONTAL` or `SwingConstants.VERTICAL`).
  - GroupLayout.SequentialGroup createSequentialGroup()  
creates a group that lays out its children sequentially.
  - GroupLayout.ParallelGroup createParallelGroup()
  - GroupLayout.ParallelGroup createParallelGroup(GroupLayout.Alignment align)
  - GroupLayout.ParallelGroup createParallelGroup(GroupLayout.Alignment align, boolean resizable)  
creates a group that lays out its children in parallel.
- Parameters:*
- |           |  |
|-----------|--|
| align     | One of <code>BASELINE</code> , <code>LEADING</code> (default), <code>TRAILING</code> , or <code>CENTER</code>  |
| resizable | true (default) when the group can be resized; false if the preferred size is also the minimum and maximum size |
- boolean getHonorsVisibility()
  - void setHonorsVisibility(boolean b)  
gets or sets the `honorsVisibility` property. When true (the default), non-visible components are not laid out. When false, they are laid out as if they were visible. This is useful when you temporarily hide some components and don't want the layout to change.

- `boolean getAutoCreateGaps()`
- `void setAutoCreateGaps(boolean b)`
- `boolean getAutoCreateContainerGaps()`
- `void setAutoCreateContainerGaps(boolean b)`  
gets and sets the `autoCreateGaps` and `autoCreateContainerGaps` properties. When true, gaps are automatically added between components or the at the container boundaries. The default is false. A true value is useful when you manually produce a `GroupLayout`.

**API** `javax.swing.GroupLayout.Group`

- `GroupLayout.Group addComponent(Component c)`
- `GroupLayout.Group addComponent(Component c, int minimumSize, int preferredSize, int maximumSize)`  
adds a component to this group. The size parameters can be actual (nonnegative) values, or the special constants `GroupLayout.DEFAULT_SIZE` or `GroupLayout.PREFERRED_SIZE`. When `DEFAULT_SIZE` is used, the component's `getMinimumSize`, `getPreferredSize`, or `getMaximumSize` is called. When `PREFERRED_SIZE` is used, the component's `getPreferredSize` method is called.
- `GroupLayout.Group addGap(int size)`
- `GroupLayout.Group addGap(int minimumSize, int preferredSize, int maximumSize)`  
adds a gap of the given rigid or flexible size.
- `GroupLayout.Group addGroup(GroupLayout.Group g)`  
adds the given group to this group.

**API** `javax.swing.GroupLayout.ParallelGroup`

- `GroupLayout.ParallelGroup addComponent(Component c, GroupLayout.Alignment align)`
- `GroupLayout.ParallelGroup addComponent(Component c, GroupLayout.Alignment align, int minimumSize, int preferredSize, int maximumSize)`
- `GroupLayout.ParallelGroup addGroup(GroupLayout.Group g, GroupLayout.Alignment align)`  
adds a component or group to this group, using the given alignment (one of `BASELINE`, `LEADING`, `TRAILING`, or `CENTER`).

**API** `javax.swing.GroupLayout.SequentialGroup`

- `GroupLayout.SequentialGroup addContainerGap()`
- `GroupLayout.SequentialGroup addContainerGap(int preferredSize, int maximumSize)`  
adds a gap for separating a component and the edge of the container.
- `GroupLayout.SequentialGroup addPreferredGap(LayoutStyle.ComponentPlacement type)`  
adds a gap for separating components. The type is `LayoutStyle.ComponentPlacement.RELATED` or `LayoutStyle.ComponentPlacement.UNRELATED`.

**Using No Layout Manager**

There will be times when you don't want to bother with layout managers but just want to drop a component at a fixed location (sometimes called *absolute positioning*). This is not a great idea for platform-independent applications, but there is nothing wrong with using it for a quick prototype.

Here is what you do to place a component at a fixed location:

1. Set the layout manager to `null`.
2. Add the component you want to the container.
3. Then specify the position and size that you want:

```
frame.setLayout(null);
JButton ok = new JButton("Ok");
frame.add(ok);
ok.setBounds(10, 10, 30, 15);
```

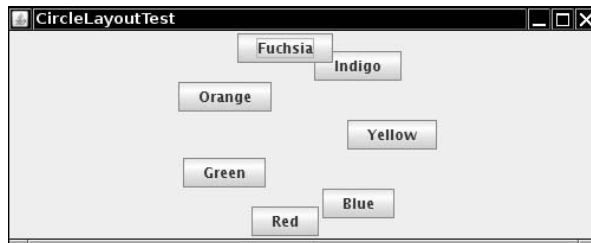
**API** `java.awt.Component 1.0`

- `void setBounds(int x, int y, int width, int height)`  
moves and resizes a component.

*Parameters:*    `x, y`                      The new top-left corner of the component  
                         `width, height`        The new size of the component

### Custom Layout Managers

You can design your own `LayoutManager` class that manages components in a special way. As a fun example, we show you how to arrange all components in a container to form a circle (see Figure 9–34).



**Figure 9–34** Circle layout

Your own layout manager must implement the `LayoutManager` interface. You need to override the following five methods:

```
void addLayoutComponent(String s, Component c);
void removeLayoutComponent(Component c);
Dimension preferredLayoutSize(Container parent);
Dimension minimumLayoutSize(Container parent);
void layoutContainer(Container parent);
```

The first two methods are called when a component is added or removed. If you don't keep any additional information about the components, you can make them do nothing. The next two methods compute the space required for the minimum and the preferred layout of the components. These are usually the same quantity. The fifth method does the actual work and invokes `setBounds` on all components.



NOTE: The AWT has a second interface, called `LayoutManager2`, with 10 methods to implement rather than 5. The main point of the `LayoutManager2` interface is to allow the user to use the `add` method with constraints. For example, the `BorderLayout` and `GridBagLayout` implement the `LayoutManager2` interface.

Listing 9–13 shows the code for the `CircleLayout` manager, which, amazingly and uselessly enough, lays out the components along a circle inside the parent.

**Listing 9–13** `CircleLayoutTest.java`

```
1. import java.awt.*;
2.
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.32 2007-06-12
7.  * @author Cay Horstmann
8.  */
9. public class CircleLayoutTest
10. {
11.     public static void main(String[] args)
12.     {
13.         EventQueue.invokeLater(new Runnable()
14.         {
15.             public void run()
16.             {
17.                 CircleLayoutFrame frame = new CircleLayoutFrame();
18.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.                 frame.setVisible(true);
20.             }
21.         });
22.     }
23. }
24.
25. /**
26.  * A frame that shows buttons arranged along a circle.
27.  */
28. class CircleLayoutFrame extends JFrame
29. {
30.     public CircleLayoutFrame()
31.     {
32.         setTitle("CircleLayoutTest");
33.
34.         setLayout(new CircleLayout());
35.         add(new JButton("Yellow"));
36.         add(new JButton("Blue"));
37.         add(new JButton("Red"));
38.         add(new JButton("Green"));
39.         add(new JButton("Orange"));
```

**Listing 9-13** CircleLayoutTest.java (continued)

```
40.     add(new JButton("Fuchsia"));
41.     add(new JButton("Indigo"));
42.     pack();
43. }
44. }
45.
46. /**
47.  * A layout manager that lays out components along a circle.
48.  */
49. class CircleLayout implements LayoutManager
50. {
51.     public void addLayoutComponent(String name, Component comp)
52.     {
53.     }
54.
55.     public void removeLayoutComponent(Component comp)
56.     {
57.     }
58.
59.     public void setSizes(Container parent)
60.     {
61.         if (sizesSet) return;
62.         int n = parent.getComponentCount();
63.
64.         preferredWidth = 0;
65.         preferredHeight = 0;
66.         minWidth = 0;
67.         minHeight = 0;
68.         maxComponentWidth = 0;
69.         maxComponentHeight = 0;
70.
71.         // compute the maximum component widths and heights
72.         // and set the preferred size to the sum of the component sizes.
73.         for (int i = 0; i < n; i++)
74.         {
75.             Component c = parent.getComponent(i);
76.             if (c.isVisible())
77.             {
78.                 Dimension d = c.getPreferredSize();
79.                 maxComponentWidth = Math.max(maxComponentWidth, d.width);
80.                 maxComponentHeight = Math.max(maxComponentHeight, d.height);
81.                 preferredWidth += d.width;
82.                 preferredHeight += d.height;
83.             }
84.         }
85.         minWidth = preferredWidth / 2;
86.         minHeight = preferredHeight / 2;
87.         sizesSet = true;
88.     }
89.
```



**Listing 9-13** CircleLayoutTest.java (continued)

```
90. public Dimension preferredLayoutSize(Container parent)
91. {
92.     setSizes(parent);
93.     Insets insets = parent.getInsets();
94.     int width = preferredWidth + insets.left + insets.right;
95.     int height = preferredHeight + insets.top + insets.bottom;
96.     return new Dimension(width, height);
97. }
98.
99. public Dimension minimumLayoutSize(Container parent)
100. {
101.     setSizes(parent);
102.     Insets insets = parent.getInsets();
103.     int width = minWidth + insets.left + insets.right;
104.     int height = minHeight + insets.top + insets.bottom;
105.     return new Dimension(width, height);
106. }
107.
108. public void layoutContainer(Container parent)
109. {
110.     setSizes(parent);
111.
112.     // compute center of the circle
113.
114.     Insets insets = parent.getInsets();
115.     int containerWidth = parent.getSize().width - insets.left - insets.right;
116.     int containerHeight = parent.getSize().height - insets.top - insets.bottom;
117.
118.     int xcenter = insets.left + containerWidth / 2;
119.     int ycenter = insets.top + containerHeight / 2;
120.
121.     // compute radius of the circle
122.
123.     int xradius = (containerWidth - maxComponentWidth) / 2;
124.     int yradius = (containerHeight - maxComponentHeight) / 2;
125.     int radius = Math.min(xradius, yradius);
126.
127.     // lay out components along the circle
128.
129.     int n = parent.getComponentCount();
130.     for (int i = 0; i < n; i++)
131.     {
132.         Component c = parent.getComponent(i);
133.         if (c.isVisible())
134.         {
135.             double angle = 2 * Math.PI * i / n;
136.
137.             // center point of component
138.             int x = xcenter + (int) (Math.cos(angle) * radius);
139.             int y = ycenter + (int) (Math.sin(angle) * radius);
```

**Listing 9-13** CircleLayoutTest.java (continued)

```

140.
141.     // move component so that its center is (x, y)
142.     // and its size is its preferred size
143.     Dimension d = c.getPreferredSize();
144.     c.setBounds(x - d.width / 2, y - d.height / 2, d.width, d.height);
145.     }
146. }
147. }
148.
149. private int minWidth = 0;
150. private int minHeight = 0;
151. private int preferredWidth = 0;
152. private int preferredHeight = 0;
153. private boolean sizesSet = false;
154. private int maxComponentWidth = 0;
155. private int maxComponentHeight = 0;
156. }

```

**API** java.awt.LayoutManager 1.0

- void addLayoutComponent(String name, Component comp)  
adds a component to the layout.  
*Parameters:* name            An identifier for the component placement  
                 comp            The component to be added
- void removeLayoutComponent(Component comp)  
removes a component from the layout.
- Dimension preferredLayoutSize(Container cont)  
returns the preferred size dimensions for the container under this layout.
- Dimension minimumLayoutSize(Container cont)  
returns the minimum size dimensions for the container under this layout.
- void layoutContainer(Container cont)  
lays out the components in a container.

**Traversal Order**

When you add many components into a window, you need to give some thought to the *traversal order*. When a window is first displayed, the first component in the traversal order has the keyboard focus. Each time the user presses the TAB key, the next component gains focus. (Recall that a component that has the keyboard focus can be manipulated with the keyboard. For example, a button can be “clicked” with the space bar when it has focus.) You may not personally care about using the TAB key to navigate through a set of controls, but plenty of users do. Among them are the mouse haters and those who cannot use a mouse, perhaps because of a handicap or because they are navigating the user interface by voice. For that reason, you need to know how Swing handles traversal order.

The traversal order is straightforward, first left to right and then top to bottom. For example, in the font dialog example, the components are traversed in the following order (see Figure 9–35):

- ❶ Face combo box
- ❷ Sample text area (press CTRL+TAB to move to the next field; the TAB character is considered text input)
- ❸ Size combo box
- ❹ Bold checkbox
- ❺ Italic checkbox



NOTE: In the old AWT, the traversal order was determined by the order in which you inserted components into a container. In Swing, the insertion order does not matter—only the layout of the components is considered.

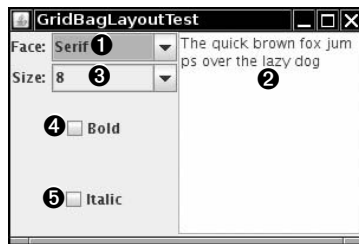


Figure 9–35 Geometric traversal order

The situation is more complex if your container contains other containers. When the focus is given to another container, it automatically ends up within the top-left component in that container and then it traverses all other components in that container. Finally, the focus is given to the component following the container.

You can use this to your advantage by grouping related elements in another container such as a panel.



NOTE: As of Java SE 1.4, you call `component.setFocusable(false);` to remove a component from the focus traversal. Previously, you had to override the `isFocusTraversable` method, but that method is now deprecated.

In summary, there are two standard traversal policies in Java SE 1.4:

- Pure AWT applications use the `DefaultFocusTraversalPolicy`. Components are included in the focus traversal if they are visible, displayable, enabled, and focusable, and if their native peers are focusable. The components are traversed in the order in which they were inserted in the container.

- Swing applications use the `LayoutFocusTraversalPolicy`. Components are included in the focus traversal if they are visible, displayable, enabled, and focusable. The components are traversed in geometric order: left to right, then top to bottom. However, a container introduces a new “cycle”—its components are traversed first before the successor of the container gains focus.



NOTE: The “cycle” notion is a bit confusing. After reaching the last element in a child container, the focus does not go back to its first element, but instead to the container’s successor. The API supports true cycles, including keystrokes that move up and down in a cycle hierarchy. However, the standard traversal policy does not use hierarchical cycles. It flattens the cycle hierarchy into a linear (depth-first) traversal.



NOTE: In Java SE 1.3, you could change the default traversal order by calling the `setNextFocusableComponent` method of the `JComponent` class. That method is now deprecated. To change the traversal order, try grouping related components into panels so that they form cycles. If that doesn’t work, you have to either install a comparator that sorts the components differently or completely replace the traversal policy. Neither operation seems intended for the faint of heart—see the Sun API documentation for details.

## Dialog Boxes

So far, all our user interface components have appeared inside a frame window that was created in the application. This is the most common situation if you write *applets* that run inside a web browser. But if you write applications, you usually want separate dialog boxes to pop up to give information to or get information from the user.

Just as with most windowing systems, AWT distinguishes between *modal* and *modeless* dialog boxes. A modal dialog box won’t let users interact with the remaining windows of the application until he or she deals with it. You use a modal dialog box when you need information from the user before you can proceed with execution. For example, when the user wants to read a file, a modal file dialog box is the one to pop up. The user must specify a file name before the program can begin the read operation. Only when the user closes the (modal) dialog box can the application proceed.

A modeless dialog box lets the user enter information in both the dialog box and the remainder of the application. One example of a modeless dialog is a toolbar. The toolbar can stay in place as long as needed, and the user can interact with both the application window and the toolbar as needed.

We start this section with the simplest dialogs—modal dialogs with just a single message. Swing has a convenient `JOptionPane` class that lets you put up a simple dialog without writing any special dialog box code. Next, you see how to write more complex dialogs by implementing your own dialog windows. Finally, you see how to transfer data from your application into a dialog and back.

We conclude this section by looking at two standard dialogs: file dialogs and color dialogs. File dialogs are complex, and you definitely want to be familiar with the Swing `JFileChooser` for this purpose—it would be a real challenge to write your own. The `JColorChooser` dialog is useful when you want users to pick colors.

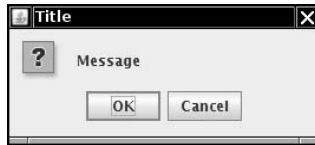
### Option Dialogs

Swing has a set of ready-made simple dialogs that suffice when you need to ask the user for a single piece of information. The `JOptionPane` has four static methods to show these simple dialogs:

<code>showMessageDialog</code>	Show a message and wait for the user to click OK
<code>showConfirmDialog</code>	Show a message and get a confirmation (like OK/Cancel)
<code>showOptionDialog</code>	Show a message and get a user option from a set of options
<code>showInputDialog</code>	Show a message and get one line of user input

Figure 9–36 shows a typical dialog. As you can see, the dialog has the following components:

- An icon
- A message
- One or more option buttons



**Figure 9–36** An option dialog

The input dialog has an additional component for user input. This can be a text field into which the user can type an arbitrary string, or a combo box from which the user can select one item.

The exact layout of these dialogs, and the choice of icons for standard message types, depend on the pluggable look and feel.

The icon on the left side depends on one of five *message types*:

```
ERROR_MESSAGE
INFORMATION_MESSAGE
WARNING_MESSAGE
QUESTION_MESSAGE
PLAIN_MESSAGE
```

The `PLAIN_MESSAGE` type has no icon. Each dialog type also has a method that lets you supply your own icon instead.

For each dialog type, you can specify a message. This message can be a string, an icon, a user interface component, or any other object. Here is how the message object is displayed:

String	Draw the string
Icon	Show the icon
Component	Show the component
Object[]	Show all objects in the array, stacked on top of each other
Any other object	Apply <code>toString</code> and show the resulting string

You can see these options by running the program in Listing 9–14 on page 455.

Of course, supplying a message string is by far the most common case. Supplying a `Component` gives you ultimate flexibility because you can make the `paintComponent` method draw anything you want.

The buttons on the bottom depend on the dialog type and the *option type*. When calling `showMessageDialog` and `showInputDialog`, you get only a standard set of buttons (OK and OK/Cancel, respectively). When calling `showConfirmDialog`, you can choose among four option types:

```
DEFAULT_OPTION
YES_NO_OPTION
YES_NO_CANCEL_OPTION
OK_CANCEL_OPTION
```

With the `showOptionDialog` you can specify an arbitrary set of options. You supply an array of objects for the options. Each array element is rendered as follows:

String	Make a button with the string as label
Icon	Make a button with the icon as label
Component	Show the component
Any other object	Apply <code>toString</code> and make a button with the resulting string as label

The return values of these functions are as follows:

<code>showMessageDialog</code>	None
<code>showConfirmDialog</code>	An integer representing the chosen option
<code>showOptionDialog</code>	An integer representing the chosen option
<code>showInputDialog</code>	The string that the user supplied or selected

The `showConfirmDialog` and `showOptionDialog` return integers to indicate which button the user chose. For the option dialog, this is simply the index of the chosen option or the value `CLOSED_OPTION` if the user closed the dialog instead of choosing an option. For the confirmation dialog, the return value can be one of the following:

```
OK_OPTION
CANCEL_OPTION
YES_OPTION
NO_OPTION
CLOSED_OPTION
```

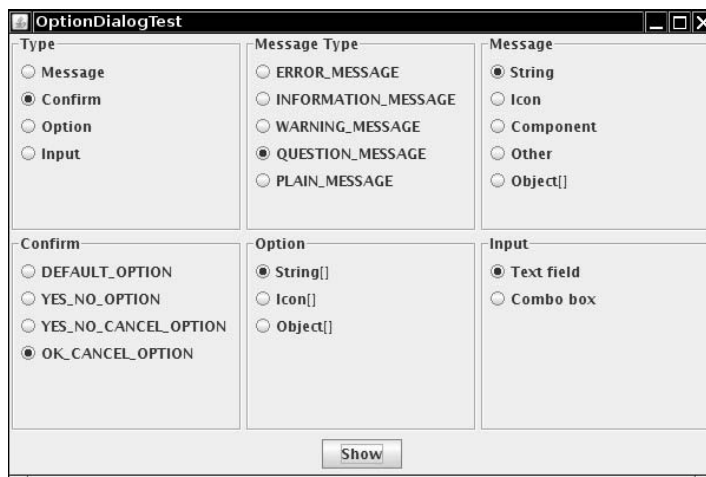
This all sounds like a bewildering set of choices, but in practice it is simple. Follow these steps:

1. Choose the dialog type (message, confirmation, option, or input).
2. Choose the icon (error, information, warning, question, none, or custom).
3. Choose the message (string, icon, custom component, or a stack of them).
4. For a confirmation dialog, choose the option type (default, Yes/No, Yes/No/Cancel, or OK/Cancel).
5. For an option dialog, choose the options (strings, icons, or custom components) and the default option.
6. For an input dialog, choose between a text field and a combo box.
7. Locate the appropriate method to call in the `JOptionPane` API.

For example, suppose you want to show the dialog in Figure 9–36. The dialog shows a message and asks the user to confirm or cancel. Thus, it is a confirmation dialog. The icon is a question icon. The message is a string. The option type is `OK_CANCEL_OPTION`. Here is the call you would make:

```
int selection = JOptionPane.showConfirmDialog(parent,
    "Message", "Title",
    JOptionPane.OK_CANCEL_OPTION,
    JOptionPane.QUESTION_MESSAGE);
if (selection == JOptionPane.OK_OPTION) . . .
```

**!** TIP: The message string can contain newline ('\n') characters. Such a string is displayed in multiple lines.



**Figure 9–37** The `OptionDialogTest` program

The program in Listing 9–14 lets you make the selections shown in Figure 9–37. It then shows you the resulting dialog.

**Listing 9–14** `OptionDialogTest.java`

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.awt.geom.*;
4. import java.util.*;
5. import javax.swing.*;
6.
```

**Listing 9-14** OptionDialogTest.java (continued)

```
7. /**
8.  * @version 1.33 2007-04-28
9.  * @author Cay Horstmann
10. */
11. public class OptionDialogTest
12. {
13.     public static void main(String[] args)
14.     {
15.         EventQueue.invokeLater(new Runnable()
16.         {
17.             public void run()
18.             {
19.                 OptionDialogFrame frame = new OptionDialogFrame();
20.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21.                 frame.setVisible(true);
22.             }
23.         });
24.     }
25. }
26.
27. /**
28.  * A panel with radio buttons inside a titled border.
29.  */
30. class ButtonPanel extends JPanel
31. {
32.     /**
33.      * Constructs a button panel.
34.      * @param title the title shown in the border
35.      * @param options an array of radio button labels
36.      */
37.     public ButtonPanel(String title, String... options)
38.     {
39.         setBorder(BorderFactory.createTitledBorder(BorderFactory.createEtchedBorder(), title));
40.         setLayout(new BorderLayout(this, BorderLayout.Y_AXIS));
41.         group = new ButtonGroup();
42.
43.         // make one radio button for each option
44.         for (String option : options)
45.         {
46.             JRadioButton b = new JRadioButton(option);
47.             b.setActionCommand(option);
48.             add(b);
49.             group.add(b);
50.             b.setSelected(option == options[0]);
51.         }
52.     }
53. }
```



**Listing 9-14** OptionDialogTest.java (continued)

```
54.  /**
55.   * Gets the currently selected option.
56.   * @return the label of the currently selected radio button.
57.   */
58.  public String getSelection()
59.  {
60.      return group.getSelection().getActionCommand();
61.  }
62.
63.  private ButtonGroup group;
64. }
65.
66. /**
67.  * A frame that contains settings for selecting various option dialogs.
68.  */
69. class OptionDialogFrame extends JFrame
70. {
71.     public OptionDialogFrame()
72.     {
73.         setTitle("OptionDialogTest");
74.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
75.
76.         JPanel gridPanel = new JPanel();
77.         gridPanel.setLayout(new GridLayout(2, 3));
78.
79.         typePanel = new ButtonPanel("Type", "Message", "Confirm", "Option", "Input");
80.         messageTypePanel = new ButtonPanel("Message Type", "ERROR_MESSAGE", "INFORMATION_MESSAGE",
81.             "WARNING_MESSAGE", "QUESTION_MESSAGE", "PLAIN_MESSAGE");
82.         messagePanel = new ButtonPanel("Message", "String", "Icon", "Component", "Other", "Object[]");
83.         optionTypePanel = new ButtonPanel("Confirm", "DEFAULT_OPTION", "YES_NO_OPTION",
84.             "YES_NO_CANCEL_OPTION", "OK_CANCEL_OPTION");
85.         optionsPanel = new ButtonPanel("Option", "String[]", "Icon[]", "Object[]");
86.         inputPanel = new ButtonPanel("Input", "Text field", "Combo box");
87.
88.         gridPanel.add(typePanel);
89.         gridPanel.add(messageTypePanel);
90.         gridPanel.add(messagePanel);
91.         gridPanel.add(optionTypePanel);
92.         gridPanel.add(optionsPanel);
93.         gridPanel.add(inputPanel);
94.
95.         // add a panel with a Show button
96.
97.         JPanel showPanel = new JPanel();
98.         JButton showButton = new JButton("Show");
99.         showButton.addActionListener(new ShowAction());
100.        showPanel.add(showButton);
101.    }
```

**Listing 9-14** OptionDialogTest.java (continued)

```

102.     add(gridPanel, BorderLayout.CENTER);
103.     add(showPanel, BorderLayout.SOUTH);
104. }
105.
106. /**
107.  * Gets the currently selected message.
108.  * @return a string, icon, component or object array, depending on the Message panel selection
109.  */
110. public Object getMessage()
111. {
112.     String s = messagePanel.getSelection();
113.     if (s.equals("String")) return messageString;
114.     else if (s.equals("Icon")) return messageIcon;
115.     else if (s.equals("Component")) return messageComponent;
116.     else if (s.equals("Object[]")) return new Object[] { messageString, messageIcon,
117.         messageComponent, messageObject };
118.     else if (s.equals("Other")) return messageObject;
119.     else return null;
120. }
121.
122. /**
123.  * Gets the currently selected options.
124.  * @return an array of strings, icons or objects, depending on the Option panel selection
125.  */
126. public Object[] getOptions()
127. {
128.     String s = optionsPanel.getSelection();
129.     if (s.equals("String[]")) return new String[] { "Yellow", "Blue", "Red" };
130.     else if (s.equals("Icon[]")) return new Icon[] { new ImageIcon("yellow-ball.gif"),
131.         new ImageIcon("blue-ball.gif"), new ImageIcon("red-ball.gif") };
132.     else if (s.equals("Object[]")) return new Object[] { messageString, messageIcon,
133.         messageComponent, messageObject };
134.     else return null;
135. }
136.
137. /**
138.  * Gets the selected message or option type
139.  * @param panel the Message Type or Confirm panel
140.  * @return the selected XXX_MESSAGE or XXX_OPTION constant from the JOptionPane class
141.  */
142. public int getType(ButtonPanel panel)
143. {
144.     String s = panel.getSelection();
145.     try
146.     {
147.         return JOptionPane.class.getField(s).getInt(null);
148.     }
149.     catch (Exception e)
150.     {

```

**Listing 9-14** OptionDialogTest.java (continued)

```

151.         return -1;
152.     }
153. }
154.
155. /**
156.  * The action listener for the Show button shows a Confirm, Input, Message or Option dialog
157.  * depending on the Type panel selection.
158.  */
159. private class ShowAction implements ActionListener
160. {
161.     public void actionPerformed(ActionEvent event)
162.     {
163.         if (typePanel.getSelection().equals("Confirm")) JOptionPane.showConfirmDialog(
164.             OptionDialogFrame.this, getMessage(), "Title", getType(optionTypePanel),
165.             getType(messageTypePanel));
166.         else if (typePanel.getSelection().equals("Input"))
167.         {
168.             if (inputPanel.getSelection().equals("Text field")) JOptionPane.showInputDialog(
169.                 OptionDialogFrame.this, getMessage(), "Title", getType(messageTypePanel));
170.             else JOptionPane.showInputDialog(OptionDialogFrame.this, getMessage(), "Title",
171.                 getType(messageTypePanel), null, new String[] { "Yellow", "Blue", "Red" },
172.                 "Blue");
173.         }
174.         else if (typePanel.getSelection().equals("Message")) JOptionPane.showMessageDialog(
175.             OptionDialogFrame.this, getMessage(), "Title", getType(messageTypePanel));
176.         else if (typePanel.getSelection().equals("Option")) JOptionPane.showOptionDialog(
177.             OptionDialogFrame.this, getMessage(), "Title", getType(optionTypePanel),
178.             getType(messageTypePanel), null, getOptions(), getOptions()[0]);
179.     }
180. }
181.
182. public static final int DEFAULT_WIDTH = 600;
183. public static final int DEFAULT_HEIGHT = 400;
184.
185. private ButtonPanel typePanel;
186. private ButtonPanel messagePanel;
187. private ButtonPanel messageTypePanel;
188. private ButtonPanel optionTypePanel;
189. private ButtonPanel optionsPanel;
190. private ButtonPanel inputPanel;
191.
192. private String messageString = "Message";
193. private Icon messageIcon = new ImageIcon("blue-ball.gif");
194. private Object messageObject = new Date();
195. private Component messageComponent = new SampleComponent();
196. }
197.
198. /**
199.  * A component with a painted surface
200.  */

```

**Listing 9–14** OptionDialogTest.java (continued)

```

201.
202. class SampleComponent extends JComponent
203. {
204.     public void paintComponent(Graphics g)
205.     {
206.         Graphics2D g2 = (Graphics2D) g;
207.         Rectangle2D rect = new Rectangle2D.Double(0, 0, getWidth() - 1, getHeight() - 1);
208.         g2.setPaint(Color.YELLOW);
209.         g2.fill(rect);
210.         g2.setPaint(Color.BLUE);
211.         g2.draw(rect);
212.     }
213.
214.     public Dimension getPreferredSize()
215.     {
216.         return new Dimension(10, 10);
217.     }
218. }

```

**API** javax.swing.JOptionPane 1.2

- static void showMessageDialog(Component parent, Object message, String title, int messageType, Icon icon)
- static void showMessageDialog(Component parent, Object message, String title, int messageType)
- static void showMessageDialog(Component parent, Object message)
- static void showInternalMessageDialog(Component parent, Object message, String title, int messageType, Icon icon)
- static void showInternalMessageDialog(Component parent, Object message, String title, int messageType)
- static void showInternalMessageDialog(Component parent, Object message)

shows a message dialog or an internal message dialog. (An internal dialog is rendered entirely within its owner frame.)

<i>Parameters:</i>	parent	The parent component (can be null)
	message	The message to show on the dialog (can be a string, icon, component, or an array of them)
	title	The string in the title bar of the dialog
	messageType	One of ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE, PLAIN_MESSAGE
	icon	An icon to show instead of one of the standard icons

- static int showConfirmDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon)
- static int showConfirmDialog(Component parent, Object message, String title, int optionType, int messageType)

- `static int showConfirmDialog(Component parent, Object message, String title, int optionType)`
- `static int showConfirmDialog(Component parent, Object message)`
- `static int showInternalConfirmDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon)`
- `static int showInternalConfirmDialog(Component parent, Object message, String title, int optionType, int messageType)`
- `static int showInternalConfirmDialog(Component parent, Object message, String title, int optionType)`
- `static int showInternalConfirmDialog(Component parent, Object message)`  
shows a confirmation dialog or an internal confirmation dialog. (An internal dialog is rendered entirely within its owner frame.) Returns the option selected by the user (one of `OK_OPTION`, `CANCEL_OPTION`, `YES_OPTION`, `NO_OPTION`), or `CLOSED_OPTION` if the user closed the dialog.

*Parameters:*

parent	The parent component (can be null)
message	The message to show on the dialog (can be a string, icon, component, or an array of them)
title	The string in the title bar of the dialog
messageType	One of <code>ERROR_MESSAGE</code> , <code>INFORMATION_MESSAGE</code> , <code>WARNING_MESSAGE</code> , <code>QUESTION_MESSAGE</code> , <code>PLAIN_MESSAGE</code>
optionType	One of <code>DEFAULT_OPTION</code> , <code>YES_NO_OPTION</code> , <code>YES_NO_CANCEL_OPTION</code> , <code>OK_CANCEL_OPTION</code>
icon	An icon to show instead of one of the standard icons

- `static int showOptionDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon, Object[] options, Object default)`
- `static int showInternalOptionDialog(Component parent, Object message, String title, int optionType, int messageType, Icon icon, Object[] options, Object default)`  
shows an option dialog or an internal option dialog. (An internal dialog is rendered entirely within its owner frame.) Returns the index of the option selected by the user, or `CLOSED_OPTION` if the user canceled the dialog.

*Parameters:*

parent	The parent component (can be null)
message	The message to show on the dialog (can be a string, icon, component, or an array of them)
title	The string in the title bar of the dialog
messageType	One of <code>ERROR_MESSAGE</code> , <code>INFORMATION_MESSAGE</code> , <code>WARNING_MESSAGE</code> , <code>QUESTION_MESSAGE</code> , <code>PLAIN_MESSAGE</code>
optionType	One of <code>DEFAULT_OPTION</code> , <code>YES_NO_OPTION</code> , <code>YES_NO_CANCEL_OPTION</code> , <code>OK_CANCEL_OPTION</code>
icon	An icon to show instead of one of the standard icons
options	An array of options (can be strings, icons, or components)
default	The default option to present to the user

- static Object showInputDialog(Component parent, Object message, String title, int messageType, Icon icon, Object[] values, Object default)
  - static String showInputDialog(Component parent, Object message, String title, int messageType)
  - static String showInputDialog(Component parent, Object message)
  - static String showInputDialog(Object message)
  - static String showInputDialog(Component parent, Object message, Object default) **1.4**
  - static String showInputDialog(Object message, Object default) **1.4**
  - static Object showInternalInputDialog(Component parent, Object message, String title, int messageType, Icon icon, Object[] values, Object default)
  - static String showInternalInputDialog(Component parent, Object message, String title, int messageType)
  - static String showInternalInputDialog(Component parent, Object message)
- shows an input dialog or an internal input dialog. (An internal dialog is rendered entirely within its owner frame.) Returns the input string typed by the user, or null if the user canceled the dialog.

<i>Parameters:</i>	parent	The parent component (can be null)
	message	The message to show on the dialog (can be a string, icon, component, or an array of them)
	title	The string in the title bar of the dialog
	messageType	One of ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE, PLAIN_MESSAGE
	icon	An icon to show instead of one of the standard icons
	values	An array of values to show in a combo box
	default	The default value to present to the user

### Creating Dialogs

In the last section, you saw how to use the `JOptionPane` class to show a simple dialog. In this section, you see how to create such a dialog by hand.

Figure 9–38 shows a typical modal dialog box, a program information box that is displayed when the user clicks the About button.

To implement a dialog box, you extend the `JDialog` class. This is essentially the same process as extending `JFrame` for the main window for an application. More precisely:

1. In the constructor of your dialog box, call the constructor of the superclass `JDialog`.
2. Add the user interface components of the dialog box.
3. Add the event handlers.
4. Set the size for the dialog box.

When you call the superclass constructor, you will need to supply the *owner frame*, the title of the dialog, and the *modality*.

The owner frame controls where the dialog is displayed. You can supply `null` as the owner; then, the dialog is owned by a hidden frame.

The modality specifies which other windows of your application are blocked while the dialog is displayed. A modeless dialog does not block other windows. A modal dialog blocks all other windows of the application (except children of the dialog). You would use a modeless dialog for a toolbox that the user can always access. On the other hand, you would use a modal dialog if you want to force the user to supply required information before continuing.



**NOTE:** As of Java SE 6, there are two additional modality types. A document-modal dialog blocks all windows belonging to the same “document,” or more precisely, all windows with the same parentless root window as the dialog. This solves a problem with help systems. In older versions, users were unable to interact with the help windows when a modal dialog was popped up. A toolkit-modal dialog blocks all windows from the same “toolkit.” A toolkit is a Java program that launches multiple applications, such as the applet engine in a browser. For more information on these advanced issues, please see <http://java.sun.com/developer/technicalArticles/J2SE/Desktop/javase6/modality>.



**Figure 9-38** An About dialog box

Here’s the code for a dialog box:

```
public AboutDialog extends JDialog
{
    public AboutDialog(JFrame owner)
    {
        super(owner, "About DialogTest", true);
        add(new JLabel(
            "<html><h1><i>Core Java</i></h1><hr>By Cay Horstmann and Gary Cornell</html>"),
            BorderLayout.CENTER);

        JPanel panel = new JPanel();
        JButton ok = new JButton("Ok");

        ok.addActionListener(new
            ActionListener()
            {
                public void actionPerformed(ActionEvent event)
                {
                    setVisible(false);
                }
            });
    }
}
```

```

        panel.add(ok);
        add(panel, BorderLayout.SOUTH);

        setSize(250, 150);
    }
}

```

As you can see, the constructor adds user interface elements: in this case, labels and a button. It adds a handler to the button and sets the size of the dialog.

To display the dialog box, you create a new dialog object and make it visible:

```

JDialog dialog = new AboutDialog(this);
dialog.setVisible(true);

```

Actually, in the sample code below, we create the dialog box only once, and we can reuse it whenever the user clicks the About button.

```

if (dialog == null) // first time
    dialog = new AboutDialog(this);
dialog.setVisible(true);

```

When the user clicks the Ok button, the dialog box should close. This is handled in the event handler of the Ok button:

```

ok.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            setVisible(false);
        }
    });

```

When the user closes the dialog by clicking on the Close box, then the dialog is also hidden. Just as with a JFrame, you can override this behavior with the `setDefaultCloseOperation` method.

Listing 9–15 is the code for the About dialog box test program.

#### Listing 9–15 DialogTest.java

```

1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.33 2007-06-12
7.  * @author Cay Horstmann
8.  */
9. public class DialogTest
10. {
11.     public static void main(String[] args)
12.     {
13.         EventQueue.invokeLater(new Runnable()
14.         {

```



**Listing 9-15** DialogTest.java (continued)

```
15.         public void run()
16.         {
17.             DialogFrame frame = new DialogFrame();
18.             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.             frame.setVisible(true);
20.         }
21.     });
22. }
23. }
24.
25. /**
26.  * A frame with a menu whose File->About action shows a dialog.
27.  */
28. class DialogFrame extends JFrame
29. {
30.     public DialogFrame()
31.     {
32.         setTitle("DialogTest");
33.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34.
35.         // construct a File menu
36.
37.         JMenuBar menuBar = new JMenuBar();
38.         setJMenuBar(menuBar);
39.         JMenu fileMenu = new JMenu("File");
40.         menuBar.add(fileMenu);
41.
42.         // add About and Exit menu items
43.
44.         // The About item shows the About dialog
45.
46.         JMenuItem aboutItem = new JMenuItem("About");
47.         aboutItem.addActionListener(new ActionListener()
48.         {
49.             public void actionPerformed(ActionEvent event)
50.             {
51.                 if (dialog == null) // first time
52.                     dialog = new AboutDialog(DialogFrame.this);
53.                 dialog.setVisible(true); // pop up dialog
54.             }
55.         });
56.         fileMenu.add(aboutItem);
57.
58.         // The Exit item exits the program
59.
60.         JMenuItem exitItem = new JMenuItem("Exit");
61.         exitItem.addActionListener(new ActionListener()
62.         {
```

**Listing 9–15** DialogTest.java (continued)

```
63.         public void actionPerformed(ActionEvent event)
64.         {
65.             System.exit(0);
66.         }
67.     });
68.     fileMenu.add(exitItem);
69. }
70.
71. public static final int DEFAULT_WIDTH = 300;
72. public static final int DEFAULT_HEIGHT = 200;
73.
74. private AboutDialog dialog;
75. }
76.
77. /**
78.  * A sample modal dialog that displays a message and waits for the user to click the Ok button.
79.  */
80. class AboutDialog extends JDialog
81. {
82.     public AboutDialog(JFrame owner)
83.     {
84.         super(owner, "About DialogTest", true);
85.
86.         // add HTML label to center
87.
88.         add(
89.             new JLabel(
90.                 "<html><h1><i>Core Java</i></h1><hr>By Cay Horstmann and Gary Cornell</html>"),
91.             BorderLayout.CENTER);
92.
93.         // Ok button closes the dialog
94.
95.         JButton ok = new JButton("Ok");
96.         ok.addActionListener(new ActionListener()
97.         {
98.             public void actionPerformed(ActionEvent event)
99.             {
100.                 setVisible(false);
101.             }
102.         });
103.
104.         // add Ok button to southern border
105.
106.         JPanel panel = new JPanel();
107.         panel.add(ok);
108.         add(panel, BorderLayout.SOUTH);
109.
110.         setSize(250, 150);
111.     }
112. }
```

**API** javax.swing.JDialog 1.2

- `public JDialog(Frame parent, String title, boolean modal)`  
constructs a dialog. The dialog is not visible until it is explicitly shown.

<i>Parameters:</i>	<code>parent</code>	The frame that is the owner of the dialog
	<code>title</code>	The title of the dialog
	<code>modal</code>	True for modal dialogs (a modal dialog blocks input to other windows)

**Data Exchange**

The most common reason to put up a dialog box is to get information from the user. You have already seen how easy it is to make a dialog box object: Give it initial data and then call `setVisible(true)` to display the dialog box on the screen. Now let us see how to transfer data in and out of a dialog box.

Consider the dialog box in Figure 9–39 that could be used to obtain a user name and a password to connect to some on-line service.



**Figure 9–39 Password dialog box**

Your dialog box should provide methods to set default data. For example, the `PasswordChooser` class of the example program has a method, `setUser`, to place default values into the next fields:


```
public void setUser(User u)
{
    username.setText(u.getName());
}
```

Once you set the defaults (if desired), you show the dialog by calling `setVisible(true)`. The dialog is now displayed.

The user then fills in the information and clicks the `Ok` or `Cancel` button. The event handlers for both buttons call `setVisible(false)`, which terminates the call to `setVisible(true)`. Alternatively, the user may close the dialog. If you did not install a window listener for the dialog, then the default window closing operation applies: The dialog becomes invisible, which also terminates the call to `setVisible(true)`.

The important issue is that the call to `setVisible(true)` blocks until the user has dismissed the dialog. This makes it easy to implement modal dialogs.

You want to know whether the user has accepted or canceled the dialog. Our sample code sets the `ok` flag to `false` before showing the dialog. Only the event handler for the `Ok` button sets the `ok` flag to `true`. In that case, you can retrieve the user input from the dialog.

 **NOTE:** Transferring data out of a modeless dialog is not as simple. When a modeless dialog is displayed, the call to `setVisible(true)` does not block and the program continues running while the dialog is displayed. If the user selects items on a modeless dialog and then clicks “Ok,” the dialog needs to send an event to some listener in the program.

The example program contains another useful improvement. When you construct a `JDialog` object, you need to specify the owner frame. However, quite often you want to show the same dialog with different owner frames. It is better to pick the owner frame *when you are ready to show the dialog*, not when you construct the `PasswordChooser` object.

The trick is to have the `PasswordChooser` extend `JPanel` instead of `JDialog`. Build a `JDialog` object on the fly in the `showDialog` method:

```
public boolean showDialog(Frame owner, String title)
{
    ok = false;

    if (dialog == null || dialog.getOwner() != owner)
    {
        dialog = new JDialog(owner, true);
        dialog.add(this);
        dialog.pack();
    }

    dialog.setTitle(title);
    dialog.setVisible(true);
    return ok;
}
```

Note that it is safe to have `owner` equal to `null`.

You can do even better. Sometimes, the owner frame isn’t readily available. It is easy enough to compute it from any parent component, like this:

```
Frame owner;
if (parent instanceof Frame)
    owner = (Frame) parent;
else
    owner = (Frame) SwingUtilities.getAncestorOfClass(Frame.class, parent);
```

We use this enhancement in our sample program. The `JOptionPane` class also uses this mechanism.

Many dialogs have a *default button*, which is automatically selected if the user presses a trigger key (`ENTER` in most “look and feel” implementations). The default button is specially marked, often with a thick outline.

You set the default button in the *root pane* of the dialog:

```
dialog.getRootPane().setDefaultButton(okButton);
```

If you follow our suggestion of laying out the dialog in a panel, then you must be careful to set the default button only after you wrapped the panel into a dialog. The panel itself has no root pane.

Listing 9–16 is the complete code that illustrates the data flow into and out of a dialog box.

**Listing 9–16** DataExchangeTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.33 2007-06-12
7.  * @author Cay Horstmann
8.  */
9. public class DataExchangeTest
10. {
11.     public static void main(String[] args)
12.     {
13.         EventQueue.invokeLater(new Runnable()
14.         {
15.             public void run()
16.             {
17.                 DataExchangeFrame frame = new DataExchangeFrame();
18.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.                 frame.setVisible(true);
20.             }
21.         });
22.     }
23. }
24.
25. /**
26.  * A frame with a menu whose File->Connect action shows a password dialog.
27.  */
28. class DataExchangeFrame extends JFrame
29. {
30.     public DataExchangeFrame()
31.     {
32.         setTitle("DataExchangeTest");
33.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
34.
35.         // construct a File menu
36.
37.         JMenuBar mbar = new JMenuBar();
38.         setJMenuBar(mbar);
39.         JMenu fileMenu = new JMenu("File");
40.         mbar.add(fileMenu);
41.
42.         // add Connect and Exit menu items
43.     }
```

**Listing 9-16** DataExchangeTest.java (continued)

```
44. JMenuItem connectItem = new JMenuItem("Connect");
45. connectItem.addActionListener(new ConnectAction());
46. fileMenu.add(connectItem);
47.
48. // The Exit item exits the program
49.
50. JMenuItem exitItem = new JMenuItem("Exit");
51. exitItem.addActionListener(new ActionListener()
52. {
53.     public void actionPerformed(ActionEvent event)
54.     {
55.         System.exit(0);
56.     }
57. });
58. fileMenu.add(exitItem);
59.
60. textArea = new JTextArea();
61. add(new JScrollPane(textArea), BorderLayout.CENTER);
62. }
63.
64. public static final int DEFAULT_WIDTH = 300;
65. public static final int DEFAULT_HEIGHT = 200;
66.
67. private PasswordChooser dialog = null;
68. private JTextArea textArea;
69.
70. /**
71.  * The Connect action pops up the password dialog.
72.  */
73.
74. private class ConnectAction implements ActionListener
75. {
76.     public void actionPerformed(ActionEvent event)
77.     {
78.         // if first time, construct dialog
79.
80.         if (dialog == null) dialog = new PasswordChooser();
81.
82.         // set default values
83.         dialog.setUser(new User("yourname", null));
84.
85.         // pop up dialog
86.         if (dialog.showDialog(DataExchangeFrame.this, "Connect")
87.         {
88.             // if accepted, retrieve user input
89.             User u = dialog.getUser();
90.             textArea.append("user name = " + u.getName() + ", password = "
91.                 + (new String(u.getPassword())) + "\n");
92.         }
93.     }
```

**Listing 9-16** DataExchangeTest.java (continued)

```
94.     }
95. }
96.
97. /**
98.  * A password chooser that is shown inside a dialog
99.  */
100. class PasswordChooser extends JPanel
101. {
102.     public PasswordChooser()
103.     {
104.         setLayout(new BorderLayout());
105.
106.         // construct a panel with user name and password fields
107.
108.         JPanel panel = new JPanel();
109.         panel.setLayout(new GridLayout(2, 2));
110.         panel.add(new JLabel("User name:"));
111.         panel.add(username = new JTextField(""));
112.         panel.add(new JLabel("Password:"));
113.         panel.add(password = new JPasswordField(""));
114.         add(panel, BorderLayout.CENTER);
115.
116.         // create Ok and Cancel buttons that terminate the dialog
117.
118.         okButton = new JButton("Ok");
119.         okButton.addActionListener(new ActionListener()
120.         {
121.             public void actionPerformed(ActionEvent event)
122.             {
123.                 ok = true;
124.                 dialog.setVisible(false);
125.             }
126.         });
127.
128.         JButton cancelButton = new JButton("Cancel");
129.         cancelButton.addActionListener(new ActionListener()
130.         {
131.             public void actionPerformed(ActionEvent event)
132.             {
133.                 dialog.setVisible(false);
134.             }
135.         });
136.
137.         // add buttons to southern border
138.
139.         JPanel buttonPanel = new JPanel();
140.         buttonPanel.add(okButton);
141.         buttonPanel.add(cancelButton);
142.         add(buttonPanel, BorderLayout.SOUTH);
143.     }
```

**Listing 9-16** DataExchangeTest.java (continued)

```
144.
145.  /**
146.   * Sets the dialog defaults.
147.   * @param u the default user information
148.   */
149. public void setUser(User u)
150. {
151.     username.setText(u.getName());
152. }
153.
154.  /**
155.   * Gets the dialog entries.
156.   * @return a User object whose state represents the dialog entries
157.   */
158. public User getUser()
159. {
160.     return new User(username.getText(), password.getPassword());
161. }
162.
163.  /**
164.   * Show the chooser panel in a dialog
165.   * @param parent a component in the owner frame or null
166.   * @param title the dialog window title
167.   */
168. public boolean showDialog(Component parent, String title)
169. {
170.     ok = false;
171.
172.     // locate the owner frame
173.
174.     Frame owner = null;
175.     if (parent instanceof Frame) owner = (Frame) parent;
176.     else owner = (Frame) SwingUtilities.getAncestorOfClass(Frame.class, parent);
177.
178.     // if first time, or if owner has changed, make new dialog
179.
180.     if (dialog == null || dialog.getOwner() != owner)
181.     {
182.         dialog = new JDialog(owner, true);
183.         dialog.add(this);
184.         dialog.getRootPane().setDefaultButton(okButton);
185.         dialog.pack();
186.     }
187.
188.     // set title and show dialog
189.
190.     dialog.setTitle(title);
191.     dialog.setVisible(true);
192.     return ok;
193. }
```



**Listing 9-16** DataExchangeTest.java (continued)

```
194.
195. private JTextField username;
196. private JPasswordField password;
197. private JButton okButton;
198. private boolean ok;
199. private JDialog dialog;
200. }
201.
202. /**
203.  * A user has a name and password. For security reasons, the password is stored as a char[],
204.  * not a String.
205.  */
206. class User
207. {
208.     public User(String aName, char[] aPassword)
209.     {
210.         name = aName;
211.         password = aPassword;
212.     }
213.
214.     public String getName()
215.     {
216.         return name;
217.     }
218.
219.     public char[] getPassword()
220.     {
221.         return password;
222.     }
223.
224.     public void setName(String aName)
225.     {
226.         name = aName;
227.     }
228.
229.     public void setPassword(char[] aPassword)
230.     {
231.         password = aPassword;
232.     }
233.
234.     private String name;
235.     private char[] password;
236. }
```

**API** javax.swing.SwingUtilities 1.2

- Container getAncestorOfClass(Class c, Component comp)  
returns the innermost parent container of the given component that belongs to the given class or one of its subclasses.

**API** javax.swing.JComponent 1.2

- `JRootPane getRootPane()`  
gets the root pane enclosing this component, or `null` if this component does not have an ancestor with a root pane.

**API** javax.swing.JRootPane 1.2

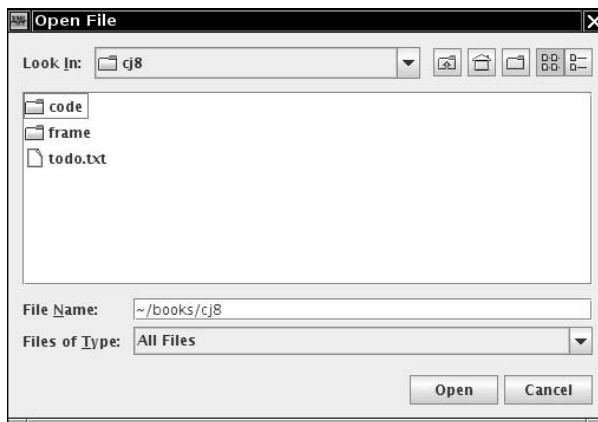
- `void setDefaultButton(JButton button)`  
sets the default button for this root pane. To deactivate the default button, call this method with a `null` parameter.

**API** javax.swing.JButton 1.2

- `boolean isDefaultButton()`  
returns true if this button is the default button of its root pane.

**File Dialogs**

When you write an application, you often want to be able to open and save files. A good file dialog box that shows files and directories and lets the user navigate the file system is hard to write, and you definitely don't want to reinvent that wheel. Fortunately, Swing provides a `JFileChooser` class that allows you to display a file dialog box similar to the one that most native applications use. `JFileChooser` dialogs are always modal. Note that the `JFileChooser` class is not a subclass of `JDialog`. Instead of calling `setVisible(true)`, you call `showOpenDialog` to display a dialog for opening a file or you call `showSaveDialog` to display a dialog for saving a file. The button for accepting a file is then automatically labeled `Open` or `Save`. You can also supply your own button label with the `showDialog` method. Figure 9-40 shows an example of the file chooser dialog box.

**Figure 9-40** File chooser dialog box

Here are the steps needed to put up a file dialog box and recover what the user chooses from the box:

1. Make a `JFileChooser` object. Unlike the constructor for the `JDialog` class, you do not supply the parent component. This allows you to reuse a file chooser dialog with multiple frames.

For example:

```
JFileChooser chooser = new JFileChooser();
```



**TIP:** Reusing a file chooser object is a good idea because the `JFileChooser` constructor can be quite slow, especially on Windows if the user has many mapped network drives.

2. Set the directory by calling the `setCurrentDirectory` method.  
For example, to use the current working directory  

```
chooser.setCurrentDirectory(new File("."));
```

you need to supply a `File` object. `File` objects are explained in detail in Chapter 12. All you need to know for now is that the constructor `File(String filename)` turns a file or directory name into a `File` object.
3. If you have a default file name that you expect the user to choose, supply it with the `setSelectedFile` method:  

```
chooser.setSelectedFile(new File(filename));
```
4. To enable the user to select multiple files in the dialog, call the `setMultiSelectionEnabled` method. This is, of course, entirely optional and not all that common.  

```
chooser.setMultiSelectionEnabled(true);
```
5. If you want to restrict the display of files in the dialog to those of a particular type (for example, all files with extension `.gif`), then you need to set a *file filter*. We discuss file filters later in this section.
6. By default, a user can select only files with a file chooser. If you want the user to select directories, use the `setFileSelectionMode` method. Call it with `JFileChooser.FILES_ONLY` (the default), `JFileChooser.DIRECTORIES_ONLY`, or `JFileChooser.FILES_AND_DIRECTORIES`.
7. Show the dialog box by calling the `showOpenDialog` or `showSaveDialog` method. You must supply the parent component in these calls:

```
int result = chooser.showOpenDialog(parent);
```

or

```
int result = chooser.showSaveDialog(parent);
```

The only difference between these calls is the label of the “approve button,” the button that the user clicks to finish the file selection. You can also call the `showDialog` method and pass an explicit text for the approve button:

```
int result = chooser.showDialog(parent, "Select");
```

These calls return only when the user has approved, canceled, or dismissed the file dialog. The return value is `JFileChooser.APPROVE_OPTION`, `JFileChooser.CANCEL_OPTION`, or `JFileChooser.ERROR_OPTION`

8. You get the selected file or files with the `getSelectedFile()` or `getSelectedFiles()` method. These methods return either a single `File` object or an array of `File` objects. If you just need the name of the file object, call its `getPath` method. For example:

```
String filename = chooser.getSelectedFile().getPath();
```

For the most part, these steps are simple. The major difficulty with using a file dialog is to specify a subset of files from which the user should choose. For example, suppose the user should choose a GIF image file. Then, the file chooser should only display files with extension `.gif`. It should also give the user some kind of feedback that the displayed files are of a particular category, such as “GIF Images.” But the situation can be more complex. If the user should choose a JPEG image file, then the extension can be either `.jpg` or `.jpeg`. Rather than coming up with a mechanism to codify these complexities, the designers of the file chooser supply a more elegant mechanism: to restrict the displayed files, you supply an object that extends the abstract class `javax.swing.filechooser.FileFilter`. The file chooser passes each file to the file filter and displays only the files that the file filter accepts.

At the time of this writing, two such subclasses are supplied: the default filter that accepts all files, and a filter that accepts all files with a given extension. Moreover, it is easy to write ad hoc file filters. You simply implement the two abstract methods of the `FileFilter` superclass:

```
public boolean accept(File f);
public String getDescription();
```

The first method tests whether a file should be accepted. The second method returns a description of the file type that can be displayed in the file chooser dialog.



**NOTE:** An unrelated `FileFilter` interface in the `java.io` package has a single method, `boolean accept(File f)`. It is used in the `listFiles` method of the `File` class to list files in a directory. We do not know why the designers of Swing didn't extend this interface—perhaps the Java class library has now become so complex that even the programmers at Sun are no longer aware of all the standard classes and interfaces.

You will need to resolve the name conflict between these two identically named types if you import both the `java.io` and the `javax.swing.filechooser` package. The simplest remedy is to import `javax.swing.filechooser.FileFilter`, not `javax.swing.filechooser.*`.

Once you have a file filter object, you use the `setFileFilter` method of the `JFileChooser` class to install it into the file chooser object:

```
chooser.setFileFilter(new FileNameExtensionFilter("Image files", "gif", "jpg");
```

You can install multiple filters to the file chooser by calling

```
chooser.addChoosableFileFilter(filter1);
chooser.addChoosableFileFilter(filter2);
. . .
```

The user selects a filter from the combo box at the bottom of the file dialog. By default, the “All files” filter is always present in the combo box. This is a good idea, just in case a user of your program needs to select a file with a nonstandard extension. However, if you want to suppress the “All files” filter, call

```
chooser.setAcceptAllFileFilterUsed(false)
```



**CAUTION:** If you reuse a single file chooser for loading and saving different file types, call `chooser.resetChoosableFilters()` to clear any old file filters before adding new ones.

Finally, you can customize the file chooser by providing special icons and file descriptions for each file that the file chooser displays. You do this by supplying an object of a class extending the `FileView` class in the `javax.swing.filechooser` package. This is definitely an advanced technique. Normally, you don't need to supply a file view—the pluggable look and feel supplies one for you. But if you want to show different icons for special file types, you can install your own file view. You need to extend the `FileView` class and implement five methods:

```
Icon getIcon(File f);
String getName(File f);
String getDescription(File f);
String getTypeDescription(File f);
Boolean isTraversable(File f);
```

Then you use the `setFileView` method to install your file view into the file chooser.

The file chooser calls your methods for each file or directory that it wants to display. If your method returns `null` for the icon, name, or description, the file chooser then consults the default file view of the look and feel. That is good, because it means you need to deal only with the file types for which you want to do something different.

The file chooser calls the `isTraversable` method to decide whether to open a directory when a user clicks on it. Note that this method returns a `Boolean` object, not a `boolean` value! This seems weird, but it is actually convenient—if you aren't interested in deviating from the default file view, just return `null`. The file chooser will then consult the default file view. In other words, the method returns a `Boolean` to let you choose among three options: `true` (`Boolean.TRUE`), `false` (`Boolean.FALSE`), and don't care (`null`).

The example program contains a simple file view class. That class shows a particular icon whenever a file matches a file filter. We use it to display a palette icon for all image files.

```
class FileIconView extends FileView
{
    public FileIconView(FileFilter aFilter, Icon anIcon)
    {
        filter = aFilter;
        icon = anIcon;
    }

    public Icon getIcon(File f)
    {
        if (!f.isDirectory() && filter.accept(f))
            return icon;
        else return null;
    }
}
```

```

    private FileFilter filter;
    private Icon icon;
}

```

You install this file view into your file chooser with the `setFileView` method:

```

chooser.setFileView(new FileIconView(filter,
    new ImageIcon("palette.gif")));

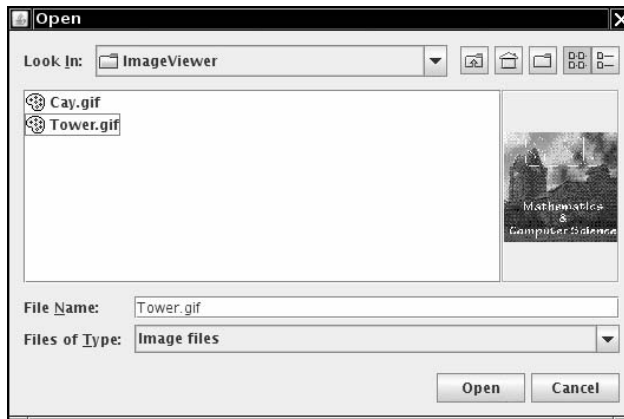
```

The file chooser will then show the palette icon next to all files that pass the filter and use the default file view to show all other files. Naturally, we use the same filter that we set in the file chooser.



**TIP:** You can find a more useful `ExampleFileView` class in the `demo/jfc/FileChooserDemo` directory of the JDK. That class lets you associate icons and descriptions with arbitrary extensions.

Finally, you can customize a file dialog by adding an *accessory* component. For example, Figure 9-41 shows a preview accessory next to the file list. This accessory displays a thumbnail view of the currently selected file.



**Figure 9-41** A file dialog with a preview accessory

An accessory can be any Swing component. In our case, we extend the `JLabel` class and set its icon to a scaled copy of the graphics image:

```

class ImagePreviewer extends JLabel
{
    public ImagePreviewer(JFileChooser chooser)
    {
        setPreferredSize(new Dimension(100, 100));
        setBorder(BorderFactory.createEtchedBorder());
    }

    public void loadImage(File f)

```

```

    {
        ImageIcon icon = new ImageIcon(f.getPath());
        if(icon.getIconWidth() > getWidth())
            icon = new ImageIcon(icon.getImage().getScaledInstance(
                getWidth(), -1, Image.SCALE_DEFAULT));
        setIcon(icon);
        repaint();
    }
}

```

There is just one challenge. We want to update the preview image whenever the user selects a different file. The file chooser uses the “JavaBeans” mechanism of notifying interested listeners whenever one of its properties changes. The selected file is a property that you can monitor by installing a `PropertyChangeListener`. We discuss this mechanism in greater detail in Chapter 8 of Volume II. Here is the code that you need to trap the notifications:

```

chooser.addPropertyChangeListener(new
    PropertyChangeListener()
    {
        public void propertyChange(PropertyChangeEvent event)
        {
            if (event.getPropertyName() == JFileChooser.SELECTED_FILE_CHANGED_PROPERTY)
            {
                File newFile = (File) event.getNewValue()
                // update the accessory
                . . .
            }
        }
    });

```

In our example program, we add this code to the `ImagePreviewer` constructor.

Listing 9–17 contains a modification of the `ImageViewer` program from Chapter 2, in which the file chooser has been enhanced by a custom file view and a preview accessory.

#### Listing 9–17 FileChooserTest.java

```

1. import java.awt.*;
2. import java.awt.event.*;
3. import java.beans.*;
4. import java.util.*;
5. import java.io.*;
6. import javax.swing.*;
7. import javax.swing.filechooser.*;
8. import javax.swing.filechooser.FileFilter;
9.
10. /**
11.  * @version 1.23 2007-06-12
12.  * @author Cay Horstmann
13.  */

```

**Listing 9-17** FileChooserTest.java (continued)

```
14. public class FileChooserTest
15. {
16.     public static void main(String[] args)
17.     {
18.         EventQueue.invokeLater(new Runnable()
19.             {
20.                 public void run()
21.                 {
22.                     ImageViewerFrame frame = new ImageViewerFrame();
23.                     frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24.                     frame.setVisible(true);
25.                 }
26.             });
27.     }
28. }
29.
30. /**
31.  * A frame that has a menu for loading an image and a display area for the loaded image.
32.  */
33. class ImageViewerFrame extends JFrame
34. {
35.     public ImageViewerFrame()
36.     {
37.         setTitle("FileChooserTest");
38.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
39.
40.         // set up menu bar
41.         JMenuBar menuBar = new JMenuBar();
42.         setJMenuBar(menuBar);
43.
44.         JMenu menu = new JMenu("File");
45.         menuBar.add(menu);
46.
47.         JMenuItem openItem = new JMenuItem("Open");
48.         menu.add(openItem);
49.         openItem.addActionListener(new FileOpenListener());
50.
51.         JMenuItem exitItem = new JMenuItem("Exit");
52.         menu.add(exitItem);
53.         exitItem.addActionListener(new ActionListener()
54.             {
55.                 public void actionPerformed(ActionEvent event)
56.                 {
57.                     System.exit(0);
58.                 }
59.             });
60.
61.         // use a label to display the images
62.         label = new JLabel();
63.         add(label);
```



**Listing 9-17** FileChooserTest.java (continued)

```
64.
65.     // set up file chooser
66.     chooser = new JFileChooser();
67.
68.     // accept all image files ending with .jpg, .jpeg, .gif
69.     /*
70.     final ExtensionFileFilter filter = new ExtensionFileFilter();
71.     filter.addExtension("jpg");
72.     filter.addExtension("jpeg");
73.     filter.addExtension("gif");
74.     filter.setDescription("Image files");
75.     */
76.     FileNameExtensionFilter filter = new FileNameExtensionFilter(
77.         "Image files", "jpg", "jpeg", "gif");
78.     chooser.setFileFilter(filter);
79.
80.     chooser.setAccessory(new ImagePreviewer(chooser));
81.
82.     chooser.setFileView(new FileIconView(filter, new ImageIcon("palette.gif")));
83. }
84.
85. /**
86.  * This is the listener for the File->Open menu item.
87.  */
88. private class FileOpenListener implements ActionListener
89. {
90.     public void actionPerformed(ActionEvent event)
91.     {
92.         chooser.setCurrentDirectory(new File("."));
93.
94.         // show file chooser dialog
95.         int result = chooser.showOpenDialog(ImageViewerFrame.this);
96.
97.         // if image file accepted, set it as icon of the label
98.         if (result == JFileChooser.APPROVE_OPTION)
99.         {
100.             String name = chooser.getSelectedFile().getPath();
101.             label.setIcon(new ImageIcon(name));
102.         }
103.     }
104. }
105.
106. public static final int DEFAULT_WIDTH = 300;
107. public static final int DEFAULT_HEIGHT = 400;
108.
109. private JLabel label;
110. private JFileChooser chooser;
111. }
112.
```

**Listing 9-17** FileChooserTest.java (continued)

```
113. /**
114.  * A file view that displays an icon for all files that match a file filter.
115.  */
116. class FileIconView extends FileView
117. {
118.     /**
119.      * Constructs a FileIconView.
120.      * @param aFilter a file filter--all files that this filter accepts will be shown with the
121.      * icon. @param anIcon--the icon shown with all accepted files.
122.      */
123.     public FileIconView(FileFilter aFilter, Icon anIcon)
124.     {
125.         filter = aFilter;
126.         icon = anIcon;
127.     }
128.
129.     public Icon getIcon(File f)
130.     {
131.         if (!f.isDirectory() && filter.accept(f)) return icon;
132.         else return null;
133.     }
134.
135.     private FileFilter filter;
136.     private Icon icon;
137. }
138.
139. /**
140.  * A file chooser accessory that previews images.
141.  */
142. class ImagePreviewer extends JLabel
143. {
144.     /**
145.      * Constructs an ImagePreviewer.
146.      * @param chooser the file chooser whose property changes trigger an image change in this
147.      * previewer
148.      */
149.     public ImagePreviewer(JFileChooser chooser)
150.     {
151.         setPreferredSize(new Dimension(100, 100));
152.         setBorder(BorderFactory.createEtchedBorder());
153.
154.         chooser.addPropertyChangeListener(new PropertyChangeListener()
155.         {
156.             public void propertyChange(PropertyChangeEvent event)
157.             {
158.                 if (event.getPropertyName() == JFileChooser.SELECTED_FILE_CHANGED_PROPERTY)
159.                 {
160.                     // the user has selected a new file
161.                     File f = (File) event.getNewValue();
162.                     if (f == null)
```

**Listing 9-17** FileChooserTest.java (continued)

```
163.         {
164.             setIcon(null);
165.             return;
166.         }
167.
168.         // read the image into an icon
169.         ImageIcon icon = new ImageIcon(f.getPath());
170.
171.         // if the icon is too large to fit, scale it
172.         if (icon.getIconWidth() > getWidth()) icon = new ImageIcon(icon.getImage()
173.             .getScaledInstance(getWidth(), -1, Image.SCALE_DEFAULT));
174.
175.         setIcon(icon);
176.     }
177. }
178. });
179. }
180. }
```

**API** javax.swing.JFileChooser 1.2

- JFileChooser()  
creates a file chooser dialog box that can be used for multiple frames.
- void setCurrentDirectory(File dir)  
sets the initial directory for the file dialog box.
- void setSelectedFile(File file)
- void setSelectedFiles(File[] file)  
sets the default file choice for the file dialog box.
- void setMultiSelectionEnabled(boolean b)  
sets or clears multiple selection mode.
- void setFileSelectionMode(int mode)  
lets the user select files only (the default), directories only, or both files and directories. The mode parameter is one of JFileChooser.FILES\_ONLY, JFileChooser.DIRECTORIES\_ONLY, and JFileChooser.FILES\_AND\_DIRECTORIES.
- int showOpenDialog(Component parent)
- int showSaveDialog(Component parent)
- int showDialog(Component parent, String approveButtonText)  
shows a dialog in which the approve button is labeled “Open”, “Save”, or with the approveButtonText string. Returns APPROVE\_OPTION, CANCEL\_OPTION (if the user selected the cancel button or dismissed the dialog), or ERROR\_OPTION (if an error occurred).
- File getSelectedFile()
- File[] getSelectedFiles()  
gets the file or files that the user selected (or returns null if the user didn’t select any file).

- `void setFileFilter(FileFilter filter)`  
sets the file mask for the file dialog box. All files for which `filter.accept` returns true will be displayed. Also adds the filter to the list of choosable filters.
- `void addChoosableFileFilter(FileFilter filter)`  
adds a file filter to the list of choosable filters.
- `void setAcceptAllFileFilterUsed(boolean b)`  
includes or suppresses an “All files” filter in the filter combo box.
- `void resetChoosableFileFilters()`  
clears the list of choosable filters. Only the “All files” filter remains unless it is explicitly suppressed.
- `void setFileView(FileView view)`  
sets a file view to provide information about the files that the file chooser displays.
- `void setAccessory(JComponent component)`  
sets an accessory component.

**API** `javax.swing.filechooser.FileFilter` 1.2

- `boolean accept(File f)`  
returns true if the file chooser should display this file.
- `String getDescription()`  
returns a description of this file filter, for example, “Image files (\*.gif,\*.jpeg)”.

**API** `javax.swing.filechooser.FileNameExtensionFilter` 6

- `FileNameExtensionFilter(String description, String ... extensions)`  
constructs a file filter with the given descriptions that accepts all directories and all files whose names end in a period followed by one of the given extension strings.

**API** `javax.swing.filechooser.FileView` 1.2

- `String getName(File f)`  
returns the name of the file `f`, or `null`. Normally, this method simply returns `f.getName()`.
- `String getDescription(File f)`  
returns a humanly readable description of the file `f`, or `null`. For example, if `f` is an HTML document, this method might return its title.
- `String getTypeDescription(File f)`  
returns a humanly readable description of the type of the file `f`, or `null`. For example, if `f` is an HTML document, this method might return a string “Hypertext document”.
- `Icon getIcon(File f)`  
returns an icon for the file `f`, or `null`. For example, if `f` is a JPEG file, this method might return a thumbnail icon.

- `Boolean isTraversable(File f)`  
returns `Boolean.TRUE` if `f` is a directory that the user can open. This method might return `false` if a directory is conceptually a compound document. Like all `FileView` methods, this method can return `null` to signify that the file chooser should consult the default view instead.

### Color Choosers

As you saw in the preceding section, a high-quality file chooser is an intricate user interface component that you definitely do not want to implement yourself. Many user interface toolkits provide other common dialogs: to choose a date/time, currency value, font, color, and so on. The benefit is twofold. Programmers can simply use a high-quality implementation rather than rolling their own. And users have a common experience for these selections.

At this point, Swing provides only one additional chooser, the `JColorChooser` (see Figures 9-42 through 9-44). You use it to let users pick a color value. Like the `JFileChooser` class, the color chooser is a component, not a dialog, but it contains convenience methods to create dialogs that contain a color chooser component.

Here is how you show a modal dialog with a color chooser:

```
Color selectedColor = JColorChooser.showDialog(parent, title, initialColor);
```

Alternatively, you can display a modeless color chooser dialog. You supply the following:

- A parent component
- The title of the dialog
- A flag to select either a modal or a modeless dialog
- A color chooser
- Listeners for the “OK” and “Cancel” buttons (or `null` if you don’t want a listener)

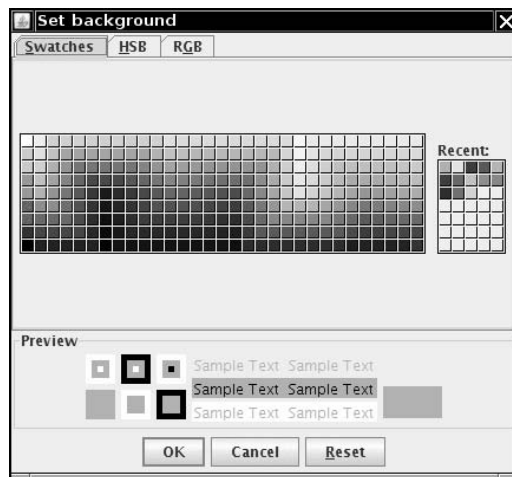


Figure 9-42 The Swatches pane of a color chooser

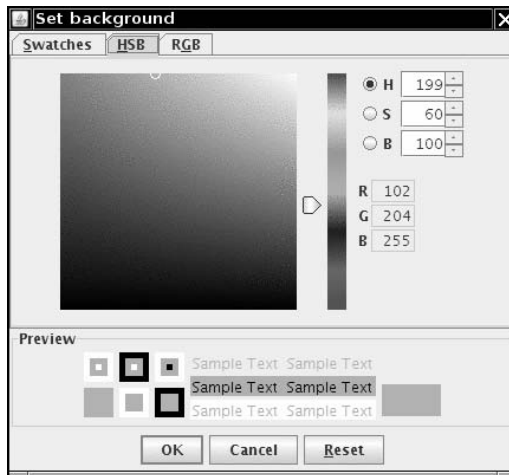


Figure 9-43 The HSB pane of a color chooser

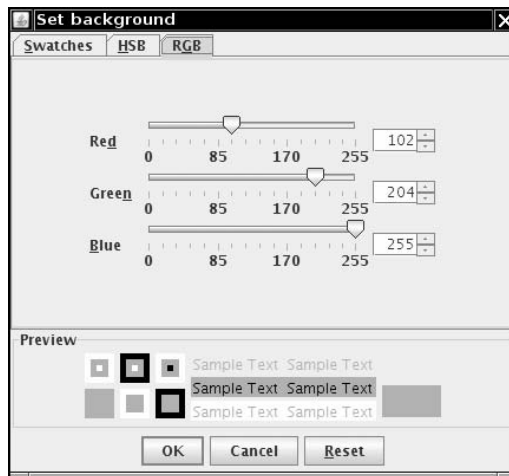


Figure 9-44 The RGB pane of a color chooser

Here is how you make a modeless dialog that sets the background color when the user clicks the OK button:

```
chooser = new JColorChooser();
dialog = JColorChooser.createDialog(
    parent,
    "Background Color",
```

```

false /* not modal */,
chooser,
new ActionListener() // OK button listener
{
    public void actionPerformed(ActionEvent event)
    {
        setBackground(chooser.getColor());
    }
},
null /* no Cancel button listener */);

```

You can do even better than that and give the user immediate feedback of the color selection. To monitor the color selections, you need to obtain the selection model of the chooser and add a change listener:

```

chooser.getSelectionModel().addChangeListener(new
ChangeListener()
{
    public void stateChanged(ChangeEvent event)
    {
        do something with chooser.getColor();
    }
});

```

In this case, there is no benefit to the OK and Cancel buttons that the color chooser dialog provides. You can just add the color chooser component directly into a modeless dialog:

```

dialog = new JDialog(parent, false /* not modal */);
dialog.add(chooser);
dialog.pack();

```

The program in Listing 9–18 shows the three types of dialogs. If you click on the Modal button, you must select a color before you can do anything else. If you click on the Modeless button, you get a modeless dialog, but the color change only happens when you click the OK button on the dialog. If you click the Immediate button, you get a modeless dialog without buttons. As soon as you pick a different color in the dialog, the background color of the panel is updated.

#### Listing 9–18 ColorChooserTest.java

```

1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4. import javax.swing.event.*;
5.
6. /**
7.  * @version 1.03 2007-06-12
8.  * @author Cay Horstmann
9.  */
10. public class ColorChooserTest
11. {

```

**Listing 9-18** ColorChooserTest.java (continued)

```
12. public static void main(String[] args)
13. {
14.     EventQueue.invokeLater(new Runnable()
15.     {
16.         public void run()
17.         {
18.             ColorChooserFrame frame = new ColorChooserFrame();
19.             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20.             frame.setVisible(true);
21.         }
22.     });
23. }
24. }
25.
26. /**
27.  * A frame with a color chooser panel
28.  */
29. class ColorChooserFrame extends JFrame
30. {
31.     public ColorChooserFrame()
32.     {
33.         setTitle("ColorChooserTest");
34.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
35.
36.         // add color chooser panel to frame
37.
38.         ColorChooserPanel panel = new ColorChooserPanel();
39.         add(panel);
40.     }
41.
42.     public static final int DEFAULT_WIDTH = 300;
43.     public static final int DEFAULT_HEIGHT = 200;
44. }
45.
46. /**
47.  * A panel with buttons to pop up three types of color choosers
48.  */
49. class ColorChooserPanel extends JPanel
50. {
51.     public ColorChooserPanel()
52.     {
53.         JButton modalButton = new JButton("Modal");
54.         modalButton.addActionListener(new ModalListener());
55.         add(modalButton);
56.
57.         JButton modelessButton = new JButton("Modeless");
58.         modelessButton.addActionListener(new ModelessListener());
59.         add(modelessButton);
60.     }

```



**Listing 9-18** ColorChooserTest.java (continued)

```
61.     JButton immediateButton = new JButton("Immediate");
62.     immediateButton.addActionListener(new ImmediateListener());
63.     add(immediateButton);
64. }
65.
66. /**
67.  * This listener pops up a modal color chooser
68.  */
69. private class ModalListener implements ActionListener
70. {
71.     public void actionPerformed(ActionEvent event)
72.     {
73.         Color defaultColor = getBackground();
74.         Color selected = JColorChooser.showDialog(ColorChooserPanel.this, "Set background",
75.             defaultColor);
76.         if (selected != null) setBackground(selected);
77.     }
78. }
79.
80. /**
81.  * This listener pops up a modeless color chooser. The panel color is changed when the user
82.  * clicks the Ok button.
83.  */
84. private class ModelessListener implements ActionListener
85. {
86.     public ModelessListener()
87.     {
88.         chooser = new JColorChooser();
89.         dialog = JColorChooser.createDialog(ColorChooserPanel.this, "Background Color",
90.             false /* not modal */, chooser, new ActionListener() // OK
91.             // button
92.             // listener
93.             {
94.                 public void actionPerformed(ActionEvent event)
95.                 {
96.                     setBackground(chooser.getColor());
97.                 }
98.             }, null /* no Cancel button listener */);
99.     }
100.
101.     public void actionPerformed(ActionEvent event)
102.     {
103.         chooser.setColor(getBackground());
104.         dialog.setVisible(true);
105.     }
106.
107.     private JDialog dialog;
108.     private JColorChooser chooser;
109. }
110.
```

**Listing 9-18** ColorChooserTest.java (continued)

```

111.  /**
112.   * This listener pops up a modeless color chooser. The panel color is changed immediately
113.   * when the user picks a new color.
114.   */
115.  private class ImmediateListener implements ActionListener
116.  {
117.      public ImmediateListener()
118.      {
119.          chooser = new JColorChooser();
120.          chooser.getSelectionModel().addChangeListener(new ChangeListener()
121.          {
122.              public void stateChanged(ChangeEvent event)
123.              {
124.                  setBackground(chooser.getColor());
125.              }
126.          });
127.
128.          dialog = new JDialog((Frame) null, false /* not modal */);
129.          dialog.add(chooser);
130.          dialog.pack();
131.      }
132.
133.      public void actionPerformed(ActionEvent event)
134.      {
135.          chooser.setColor(getBackground());
136.          dialog.setVisible(true);
137.      }
138.
139.      private JDialog dialog;
140.      private JColorChooser chooser;
141.  }
142. }

```

**API** javax.swing.JColorChooser 1.2

- JColorChooser()
  - constructs a color chooser with an initial color of white.
- Color getColor()
- void setColor(Color c)
  - gets and sets the current color of this color chooser.
- static Color showDialog(Component parent, String title, Color initialColor)
  - shows a modal dialog that contains a color chooser.

*Parameters:*

parent	The component over which to pop up the dialog
title	The title for the dialog box frame
initialColor	The initial color to show in the color chooser

- `static JDialog createDialog(Component parent, String title, boolean modal, JColorChooser chooser, ActionListener okListener, ActionListener cancelListener)`  
creates a dialog box that contains a color chooser.

<i>Parameters:</i>	<code>parent</code>	The component over which to pop up the dialog
	<code>title</code>	The title for the dialog box frame
	<code>modal</code>	true if this call should block until the dialog is closed
	<code>chooser</code>	The color chooser to add to the dialog
	<code>okListener,</code> <code>cancelListener</code>	The listeners of the OK and Cancel buttons

This ends our discussion of user interface components. The material in Chapters 7 through 9 showed you how to implement simple GUIs in Swing. Turn to Volume II for more advanced Swing components and sophisticated graphics techniques.



# *Chapter*

# 10

## DEPLOYING APPLICATIONS AND APPLETS

- ▼ JAR FILES
- ▼ JAVA WEB START
- ▼ APPLETS
- ▼ STORAGE OF APPLICATION PREFERENCES

**A**t this point, you should be comfortable with using most of the features of the Java programming language, and you have had a pretty thorough introduction to basic graphics programming in Java. Now that you are ready to create applications for your users, you will want to know how to package them for deployment on your users' computers. The traditional deployment choice—which was responsible for the unbelievable hype during the first few years of Java's life—is to use *applets*. An applet is a special kind of Java program that a Java-enabled browser can download from the Internet and then run. The hopes were that users would be freed from the hassles of installing software and that they could access their software from any Java-enabled computer or device with an Internet connection.

For a number of reasons, applets never quite lived up to these expectations. Therefore, we start this chapter with instructions for packaging applications. We then turn to the *Java Web Start* mechanism, an alternative approach for Internet-based application delivery, which fixes some of the problems of applets. Finally, we cover applets and show you in which circumstances you still want to use them.

We also discuss how your applications can store configuration information and user preferences.

### JAR Files

When you package your application, you want to give your users a single file, not a directory structure filled with class files. Java Archive (JAR) files were designed for this purpose. A JAR file can contain both class files and other file types such as image and sound files. Moreover, JAR files are compressed, using the familiar ZIP compression format.



**TIP:** Java SE 5.0 introduced a new compression scheme, called "pack200", that is specifically tuned to compress class files more efficiently than the generic ZIP compression algorithm. Sun claims a compression rate of close to 90% for class files. See <http://java.sun.com/javase/6/docs/technotes/guide/deployment/deployment-guide/pack200.html> for more information.

You use the `jar` tool to make JAR files. (In the default JDK installation, it's in the `jdk/bin` directory.) The most common command to make a new JAR file uses the following syntax:

```
jar cvf JARFileName File1 File2 . . .
```

For example:

```
jar cvf CalculatorClasses.jar *.class icon.gif
```

In general, the `jar` command has the following format:

```
jar options File1 File2 . . .
```

Table 10-1 lists all the options for the `jar` program. They are similar to the options of the UNIX `tar` command.

You can package application programs, program components (sometimes called "beans"—see Chapter 8 of Volume II), and code libraries into JAR files. For example, the runtime library of the JDK is contained in a very large file `rt.jar`.

**Table 10-1 jar Program Options**

Option	Description
c	Creates a new or empty archive and adds files to it. If any of the specified file names are directories, the jar program processes them recursively.
C	Temporarily changes the directory. For example, jar cvf JARFileName.jar -C classes *.class changes to the classes subdirectory to add class files.
e	Creates an entry point in the manifest (see “Executable JAR Files” on page 496).
f	Specifies the JAR file name as the second command-line argument. If this parameter is missing, jar will write the result to standard output (when creating a JAR file) or read it from standard input (when extracting or tabulating a JAR file).
i	Creates an index file (for speeding up lookups in a large archive).
m	Adds a <i>manifest</i> to the JAR file. A manifest is a description of the archive contents and origin. Every archive has a default manifest, but you can supply your own if you want to authenticate the contents of the archive.
M	Does not create a manifest file for the entries.
t	Displays the table of contents.
u	Updates an existing JAR file.
v	Generates verbose output.
x	Extracts files. If you supply one or more file names, only those files are extracted. Otherwise, all files are extracted.
0	Stores without ZIP compression.

### The Manifest

In addition to class files, images, and other resources, each JAR file contains a *manifest* file that describes special features of the archive.

The manifest file is called MANIFEST.MF and is located in a special META-INF subdirectory of the JAR file. The minimum legal manifest is quite boring—just

```
Manifest-Version: 1.0
```

Complex manifests can have many more entries. The manifest entries are grouped into sections. The first section in the manifest is called the *main section*. It applies to the whole JAR file. Subsequent entries can specify properties of named entities such as individual files, packages, or URLs. Those entries must begin with a `Name` entry. Sections are separated by blank lines. For example:

```
Manifest-Version: 1.0
lines describing this archive
```

```
Name: Wozzle.class
lines describing this file
```

```
Name: com/mycompany/mypkg/  
lines describing this package
```

To edit the manifest, place the lines that you want to add to the manifest into a text file. Then run

```
jar cfm JARFileName ManifestFileName . . .
```

For example, to make a new JAR file with a manifest, run

```
jar cfm MyArchive.jar manifest.mf com/mycompany/mypkg/*.class
```

To update the manifest of an existing JAR file, place the additions into a text file and use a command such as

```
jar ufm MyArchive.jar manifest-additions.mf
```



NOTE: See <http://java.sun.com/javase/6/docs/technotes/guides/jar> for more information on the JAR and manifest file formats.

---

### **Executable JAR Files**

As of Java SE 6, you can use the `e` option of the `jar` command to specify the *entry point* of your program—the class that you would normally specify when invoking the `java` program launcher:

```
jar cvfe MyProgram.jar com.mycompany.mypkg.MainAppClass files to add
```

Users can now simply start the program as

```
java -jar MyProgram.jar
```

In older versions of the JDK, you had to specify the *main class* of your program, following this procedure:

```
Main-Class: com.mycompany.mypkg.MainAppClass
```

Do not add a `.class` extension to the main class name. Then run the `jar` command:

```
jar cvfm MyProgram.jar mainclass.mf files to add
```



CAUTION: The last line in the manifest must end with a newline character. Otherwise, the manifest will not be read correctly. It is a common error to produce a text file containing just the `Main-Class` line without a line terminator.

---

Depending on the operating system configuration, you may be able to launch the application by double-clicking on the JAR file icon. Here are behaviors for various operating systems:

- On Windows, the Java runtime installer creates a file association for the `.jar` extension that launches the file with the `javaw -jar` command. (Unlike the `java` command, the `javaw` command doesn't open a shell window.)
- On Solaris, the operating system recognizes the "magic number" of a JAR file and starts it with the `java -jar` command.
- On Mac OS X, the operating system recognizes the `.jar` file extension and executes the Java program when you double-click on a JAR file.



However, a Java program in a JAR file does not have the same feel as a native application. On Windows, you can use third-party wrapper utilities that turn JAR files into Windows executables. A wrapper is a Windows program with the familiar .exe extension that locates and launches the Java virtual machine (JVM), or tells the user what to do when no JVM is found. There are a number of commercial and open source products, such as JSmooth (<http://jsmooth.sourceforge.net>) and Launch4J (<http://launch4j.sourceforge.net>). The open source installer generator IzPack (<http://izpack.org>) also contains a native launcher. For more information on this topic, see <http://www.javalobby.org/articles/java2exe>.

On the Macintosh, the situation is a bit easier. The application package utility MRJAppBuilder lets you turn a JAR file into a first-class Mac application. For more information, see <http://java.sun.com/developer/technicalArticles/JavaLP/JavaToMac3>.

### Resources

Classes that are used in both applets and applications often use associated data files, such as

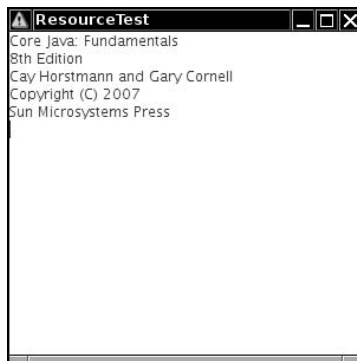
- Image and sound files;
- Text files with message strings and button labels; or
- Files with binary data, for example, to describe the layout of a map.

In Java, such an associated file is called a *resource*.



**NOTE:** In Windows, the term “resource” has a more specialized meaning. Windows resources also consist of images, button labels, and so on, but they are attached to the executable file and accessed by a standard programming interface. In contrast, Java resources are stored as separate files, not as part of class files. It is up to each program to access and interpret the resource data.

For example, consider a class, `AboutPanel`, that displays a message such as the one in Figure 10-1.



**Figure 10-1** Displaying a resource from a JAR file

Of course, the book title and copyright year in the panel will change for the next edition of the book. To make it easy to track this change, we want to put the text inside a file and not hardcode it as a string.

But where should you put a file such as `about.txt`? Of course, it would be convenient if you simply placed it with the rest of the program files inside the JAR file.

The class loader knows how to search for class files until it has located them somewhere on the class path, or in an archive, or on a web server. The resource mechanism gives you the same convenience for files that aren't class files. Here are the necessary steps:

1. Get the `Class` object of the class that has a resource, for example, `AboutPanel.class`.
2. If the resource is an image or audio file, call `getResource(filename)` to get the resource location as a URL. Then read it with the `getImage` or `getAudioClip` method.
3. For resources other than images or audio files, use the `getResourceAsStream` method to read the data in the file.

The point is that the class loader remembers how to locate the class and it can then search for the associated resource in the same location.

For example, to make an icon with the image file `about.gif`, do the following:

```
URL url = ResourceTest.class.getResource("about.gif");
Image img = Toolkit.getDefaultToolkit().getImage(url);
```

That means "locate the `about.gif` file at the same place where you find the `ResourceTest` class."

To read in the file `about.txt`, you use these commands:

```
InputStream stream = ResourceTest.class.getResourceAsStream("about.txt");
Scanner in = new Scanner(stream);
```

Instead of placing a resource file inside the same directory as the class file, you can place it in a subdirectory. You can use a hierarchical resource name such as

```
data/text/about.txt
```

This is a relative resource name, and it is interpreted relative to the package of the class that is loading the resource. Note that you must always use the `/` separator, regardless of the directory separator on the system that actually stores the resource files. For example, on the Windows file system, the resource loader automatically translates `/` to `\` separators.

A resource name starting with a `/` is called an absolute resource name. It is located in the same way that a class inside a package would be located. For example, a resource

```
/corejava/title.txt
```

is located in the `corejava` directory (which may be a subdirectory of the class path, inside a JAR file, or, for applets, on a web server).

Automating the loading of files is all that the resource loading feature does. There are no standard methods for interpreting the contents of a resource file. Each program must have its own way of interpreting the contents of its resource files.

Another common application of resources is the internationalization of programs. Language-dependent strings, such as messages and user interface labels, are stored in resource files, with one file for each language. The *internationalization API*, which is

discussed in Chapter 5 of Volume II, supports a standard method for organizing and accessing these localization files.

Listing 10–1 shows the source code of the program that demonstrates resource loading. Compile, build a JAR file, and execute it:

```
javac ResourceTest.java
jar cvfm ResourceTest.jar ResourceTest.mf *.class *.gif *.txt
java -jar ResourceTest.jar
```

Move the JAR file to a different directory and run it again to check that the program reads the resource files from the JAR file, not the current directory.

**Listing 10–1** ResourceTest.java

```
1. import java.awt.*;
2. import java.io.*;
3. import java.net.*;
4. import java.util.*;
5. import javax.swing.*;
6.
7. /**
8.  * @version 1.4 2007-04-30
9.  * @author Cay Horstmann
10. */
11. public class ResourceTest
12. {
13.     public static void main(String[] args)
14.     {
15.         EventQueue.invokeLater(new Runnable()
16.         {
17.             public void run()
18.             {
19.                 ResourceTestFrame frame = new ResourceTestFrame();
20.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21.                 frame.setVisible(true);
22.             }
23.         });
24.     }
25. }
26.
27. /**
28.  * A frame that loads image and text resources.
29.  */
30. class ResourceTestFrame extends JFrame
31. {
32.     public ResourceTestFrame()
33.     {
34.         setTitle("ResourceTest");
35.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
36.         URL aboutURL = getClass().getResource("about.gif");
```

**Listing 10-1** ResourceTest.java (continued)

```

37.     Image img = Toolkit.getDefaultToolkit().getImage(aboutURL);
38.     setIconImage(img);
39.
40.     JTextArea textArea = new JTextArea();
41.     InputStream stream = getClass().getResourceAsStream("about.txt");
42.     Scanner in = new Scanner(stream);
43.     while (in.hasNext())
44.         textArea.append(in.nextLine() + "\n");
45.     add(textArea);
46. }
47.
48.     public static final int DEFAULT_WIDTH = 300;
49.     public static final int DEFAULT_HEIGHT = 300;
50. }
```

**API** java.lang.Class 1.0

- URL getResource(String name) 1.1
- InputStream getResourceAsStream(String name) 1.1  
finds the resource in the same place as the class and then returns a URL or input stream you can use for loading the resource. Returns null if the resource isn't found, and so does not throw an exception for an I/O error.

**Sealing**

We mentioned in Chapter 4 that you can *seal* a Java language package to ensure that no further classes can add themselves to it. You would want to seal a package if you use package-visible classes, methods, and fields in your code. Without sealing, other classes can place themselves into the same package and thereby gain access to its package-visible features.

For example, if you seal the package `com.mycompany.util`, then no class outside the sealed archive can be defined with the statement

```
package com.mycompany.util;
```

To achieve this, you put all classes of the package into a JAR file. By default, packages in a JAR file are not sealed. You can change that global default by placing the line

```
Sealed: true
```

into the main section of the manifest. For each individual package, you can specify whether you want the package sealed or not, by adding another section to the JAR file manifest, like this:

```

Name: com/mycompany/util/
Sealed: true

Name: com/mycompany/misc/
Sealed: false
```

To seal a package, make a text file with the manifest instructions. Then run the `jar` command in the usual way:

```
jar cvfm MyArchive.jar manifest.mf files to add
```

### Java Web Start

Java Web Start is a technology for delivering applications over the Internet. Java Web Start applications have the following characteristics:

- They are typically delivered through a browser. Once a Java Web Start application has been downloaded, it can be started without using a browser.
- They do not live inside a browser window. The application is displayed in its own frame, outside the browser.
- They do not use the Java implementation of the browser. The browser simply launches an external application whenever it loads a Java Web Start application descriptor. That is the same mechanism that is used to launch other helper applications such as Adobe Acrobat or RealAudio.
- Digitally signed applications can be given arbitrary access rights on the local machine. Unsigned applications run in a “sandbox,” which prohibits potentially dangerous operations.

To prepare an application for delivery by Java Web Start, you package it in one or more JAR files. Then you prepare a descriptor file in Java Network Launch Protocol (JNLP) format. Place these files on a web server.

You also need to ensure that your web server reports a MIME type of `application/x-java-jnlp-file` for files with extension `.jnlp`. (Browsers use the MIME type to determine which helper application to launch.) Consult your web server documentation for details.



**TIP:** To experiment with Java Web Start, install Tomcat from <http://jakarta.apache.org/tomcat>. Tomcat is a container for servlets and JSP pages, but it also serves web pages. It is preconfigured to serve the correct MIME type for JNLP files.

Let's try out Java Web Start to deliver the calculator application from Chapter 9. Follow these steps:

1. Compile `Calculator.java`.
2. Prepare a manifest file `Calculator.mf` with the line  
`Main-Class: Calculator`
3. Produce a JAR file with the command  
`jar cvfm Calculator.jar Calculator.mf *.class`
4. Prepare the launch file `Calculator.jnlp` with the following contents:

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0+" codebase="http://localhost:8080/calculator/" href="Calculator.jnlp">
  <information>
    <title>Calculator Demo Application</title>
    <vendor>Cay S. Horstmann</vendor>
    <description>A Calculator</description>
  </information>
</jnlp>
```

```

</information>
<resources>
  <j2se version="1.5.0+"/>
  <jar href="Calculator.jar"/>
</resources>
<application-desc/>
</jnlp>

```

(Note that the version number must be 1.5.0, not 5.0. As of Java SE 6, you can use `java` instead of `j2se` for the tag name.)

The launch file format is fairly self-explanatory. For a full specification, see <http://java.sun.com/products/javawebstart/docs/developersguide.html>.

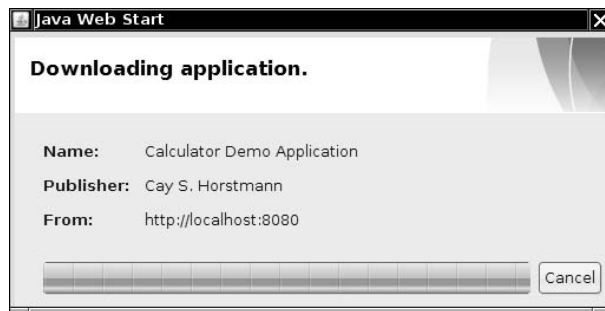
- If you use Tomcat, make a directory `tomcat/webapps/calculator`, where `tomcat` is the base directory of your Tomcat installation. Make a subdirectory `tomcat/webapps/calculator/WEB-INF`, and place the following minimal `web.xml` file inside the `WEB-INF` subdirectory:

```

<?xml version="1.0" encoding="utf-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_5.xsd">
</web-app>

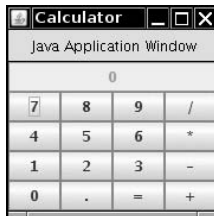
```

- Place the JAR file and the launch file on your web server so that the URL matches the codebase entry in the JNLP file. If you use Tomcat, put them into the `tomcat/webapps/calculator` directory.
- Make sure that your browser has been configured for Java Web Start, by checking that the `application/x-java-jnlp-file` MIME type is associated with the `javaws` application. If you installed the JDK, the configuration should be automatic.
- Start Tomcat.
- Point your browser to the JNLP file. For example, if you use Tomcat, go to `http://localhost:8080/calculator/Calculator.jnlp`.
- You should see the launch window for Java Web Start (see Figure 10–2).



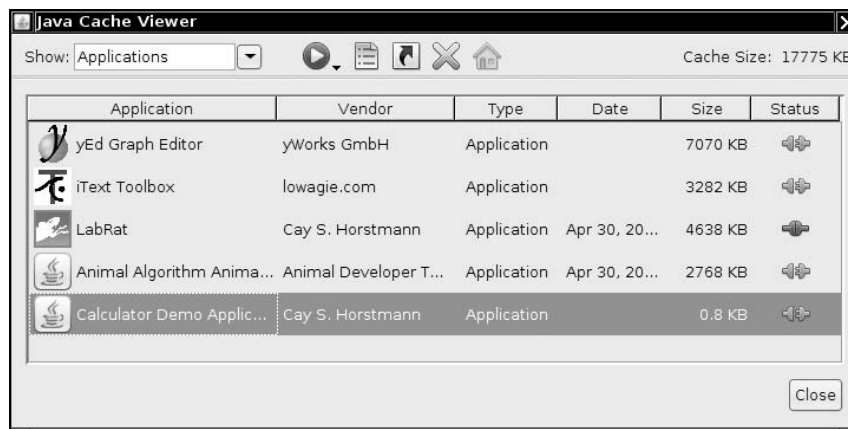
**Figure 10–2** Launching Java Web Start

11. Soon afterward, the calculator should come up, with a border marking it as a Java application (see Figure 10–3).



**Figure 10–3** The calculator delivered by Java Web Start

12. When you access the JNLP file again, the application is retrieved from the cache. You can review the cache content by using the Java Plug-in control panel (see Figure 10–4). As of Java SE 5.0, that control panel is used both for applets and Java Web Start applications. In Windows, look for the Java Plug-in control inside the Windows control panel. Under Linux, run `jdk/jre/bin/ControlPanel`.



**Figure 10–4** The application cache



**TIP:** If you don't want to run a web server while you are testing your JNLP configuration, you can temporarily override the codebase URL in the launch file by running

```
javaws -codebase file:///programDirectory JNLPfile
```

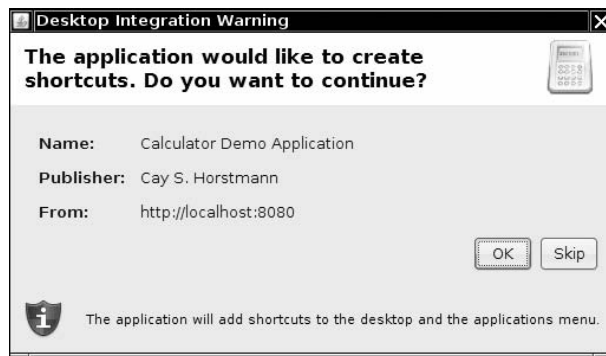
For example, in UNIX, you can simply issue this command from the directory containing the JNLP file:

```
javaws -codebase file:///`pwd` WebStartCalculator.jnlp
```

Of course, you don't want to tell your users to launch the cache viewer whenever they want to run your application again. You can have the installer offer to install desktop and menu shortcuts. Add these lines to the JNLP file:

```
<shortcut>
  <desktop/>
  <menu submenu="Accessories"/>
</shortcut>
```

When the user first downloads the application, a "desktop integration warning" is displayed (see Figure 10-5).



**Figure 10-5** The desktop integration warning

You should also supply an icon for the menu shortcut and the launch screen. Sun recommends that you supply a  $32 \times 32$  and a  $64 \times 64$  icon. Place the icon files on the web server, together with the JNLP and JAR files. Add these lines to the information section of the JNLP file:

```
<icon href="calc_icon32.png" width="32" height="32" />
<icon href="calc_icon64.png" width="64" height="64" />
```

Note that these icons are not related to the application icon. If you want the application to have an icon, you need to add a separate icon image into the JAR file and call the `setIconImage` method on the frame class. (See Listing 10-1 for an example.)

### **The Sandbox**

Whenever code is loaded from a remote site and then executed locally, security becomes vital. Clicking a single link can launch a Java Web Start application. Visiting a web page automatically starts all applets on the page. If clicking a link or visiting a web page could install arbitrary code on the user's computer, criminals would have an easy time stealing confidential information, accessing financial data, or taking over users' machines to send spam.

To ensure that the Java technology cannot be used for nefarious purposes, Java has an elaborate security model that we discuss in detail in Volume II. A *security manager* checks access to all system resources. By default, it only allows those operations that



are harmless. To allow additional operations, the code must be digitally signed and the user must approve the signing certificate.

What *can* remote code do on all platforms? It is always ok to show images and play sounds, get keystrokes and mouse clicks from the user, and send user input back to the host from which the code was loaded. That is enough functionality to show facts and figures or to get user input for placing an order. The restricted execution environment is often called the “sandbox.” Code that plays in the sandbox cannot alter the user’s system or spy on it.

In particular, programs in the sandbox have the following restrictions:

- They can *never* run any local executable program.
- They cannot read from or write to the local computer’s file system.
- They cannot find out any information about the local computer, except for the Java version used and a few harmless operating system details. In particular, code in the sandbox cannot find out the user’s name, e-mail address, and so on.
- Remotely loaded programs cannot communicate with any host other than the server from which they were downloaded; that server is called the *originating host*. This rule is often called “remote code can only phone home.” The rule protects users from code that might try to spy on intranet resources. (As of Java SE 6, a Java Web Start application can make other network connections, but the program user must consent.)
- All pop-up windows carry a warning message. This message is a security feature to ensure that users do not mistake the window for a local application. The fear is that an unsuspecting user could visit a web page, be tricked into running remote code, and then type in a password or credit card number, which can be sent back to the web server. In early versions of the JDK, that message was very ominous: “Untrusted Java Applet Window”. Every successive version watered down the warning a bit—“Unauthenticated Java Applet Window”, then “Warning: Java Applet Window”. Now it is simply “Java Applet Window” or “Java Application Window”.

### **Signed Code**

The sandbox restrictions are too restrictive for many situations. For example, on a corporate intranet, you can certainly imagine a Web Start application or applet wanting to access local files. It is possible to control in great detail which rights to grant a particular application; we discuss this in Chapter 9 of Volume II. Of course, an application can simply request to have all permissions of a desktop application, and quite a few Java Web Start applications do just that. This is accomplished by adding the following tags to the JNLP file:

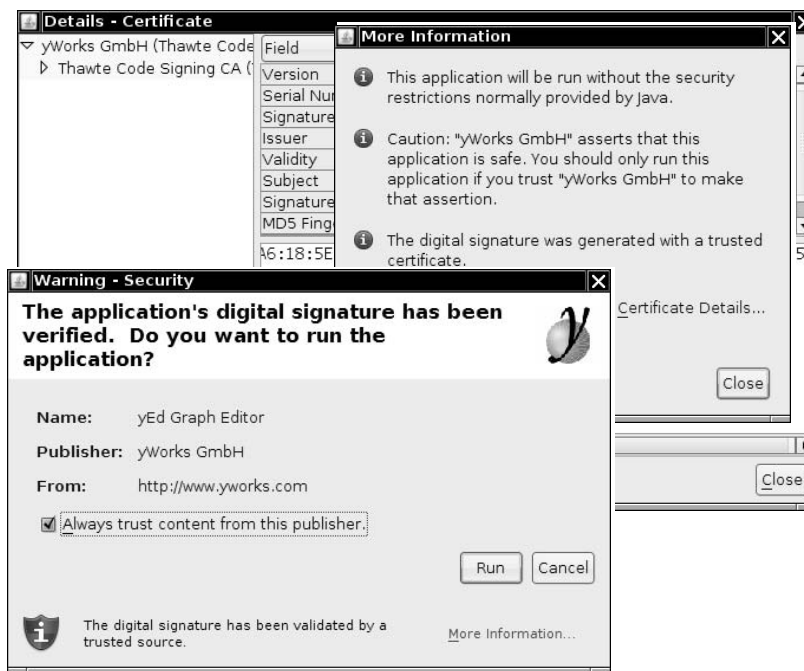
```
<security>
  <all-permissions/>
</security>
```

To run outside the sandbox, the JAR files of a Java Web Start application must be *digitally signed*. A signed JAR file carries with it a certificate that indicates the identity of the signer. Cryptographic techniques ensure that such a certificate cannot be forged, and that any effort to tamper with the signed file will be detected.

For example, suppose you receive an application that is produced and digitally signed by yWorks GmbH, using a certificate issued by Thawte (see Figure 10–6). When you receive the application, you will be assured of the following:

1. The code is exactly as it was when it was signed; no third party has tampered with it.
2. The signature really is from yWorks.
3. The certificate really was issued by Thawte. (Java Web Start knows how to check certificates from Thawte and a few other vendors.)

Unfortunately, that's all you know. You do not know that the code is inherently safe. In fact, if you click on the "More Information" link, you are told that the application will run without the security restrictions normally provided by Java. Should you install and run the application? That really depends on your trust in yWorks GmbH.



**Figure 10–6** A secure certificate

Getting a certificate from one of the supported vendors costs hundreds of dollars per year. Many developers simply generate their own and use them for code signing. Of course, Java Web Start has no way of checking the accuracy of these certificates. When you receive such an application, then you know:

1. The code is exactly as it was when it was signed; no other party has tampered with it.
2. Someone has signed the code, but Java Web Start cannot verify who it was.

This is quite worthless; anyone could have tampered with the code and then signed it, claiming to be the author. Nevertheless, Java Web Start will be perfectly happy to present the certificate for your approval (see Figure 10-7). It is theoretically possible to verify the certificate through another way, but few users have the technical savvy to do that.

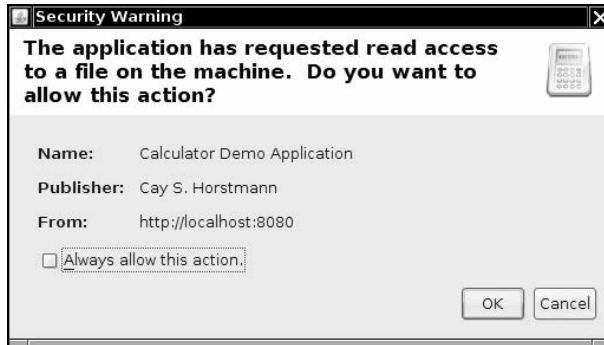


**Figure 10-7** An insecure certificate

Of course, many people download and run applications from the Internet every day. If you find that your users trust your application and your web infrastructure, go ahead and use a self-signed certificate. (See <http://java.sun.com/javase/6/docs/technotes/guides/javaws/developmentguide/development.html> for details.) If not, give your users the benefit of safety and stay within the sandbox. With the JNLP API (which we discuss in the next section), you can still allow your program to selectively access resources, subject to user approval.

### **The JNLP API**

The JNLP API allows unsigned applications to run in the sandbox and at the same time access local resources in a secure way. For example, there are services to load and save files. The application can't look at the file system and it can't specify file names. Instead, a file dialog is popped up, and the program user selects the file. Before the file dialog is popped up, the program user is alerted and must agree to proceed (see Figure 10-8). Furthermore, the API doesn't actually give the program access to a `File` object. In particular, the application has no way of finding out the file location. Thus, programmers are given the tools to implement "file open" and "file save" actions, but as much system information as possible is hidden from untrusted applications.



**Figure 10-8** A Java Web Start security advisory

The API provides the following services:

- Loading and saving files
- Accessing the clipboard
- Printing
- Downloading a file
- Displaying a document in the default browser
- Storing and retrieving persistent configuration information
- Ensuring that only a single instance of an application executes (added in Java SE 5.0)

To access a service, you use the `ServiceManager`, like this:

```
FileSaveService service = (FileSaveService) ServiceManager.lookup("javax.jnlp.FileSaveService");
```

This call throws an `UnavailableServiceException` if the service is not available.



**NOTE:** You must include the file `javaws.jar` in the class path if you want to compile programs that use the JNLP API. That file is included in the `jdk/lib` subdirectory of the JDK.

We now discuss the most useful JNLP services. To save a file, you provide suggestions for the initial path name and file extensions for the file dialog, the data to be saved, and a suggested file name. For example:

```
service.saveFileDialog(".", new String[] { "txt" }, data, "calc.txt");
```

The data must be delivered in an `InputStream`. That can be somewhat tricky to arrange.

The program in Listing 10-2 on page 511 uses the following strategy:

1. It creates a `ByteArrayOutputStream` to hold the bytes to be saved.
2. It creates a `PrintStream` that sends its data to the `ByteArrayOutputStream`.
3. It prints the information to be saved to the `PrintStream`.
4. It creates a `ByteArrayInputStream` to read the saved bytes.
5. It passes that stream to the `saveFileDialog` method.

You will learn more about streams in Chapter 1 of Volume II. For now, you can just gloss over the details in the sample program.

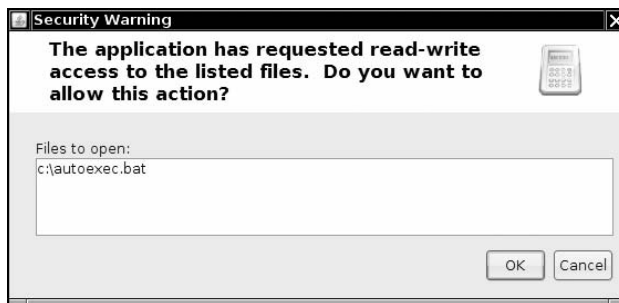
To read data from a file, you use the `FileOpenService` instead. Its `openFileDialog` receives suggestions for the initial path name and file extensions for the file dialog and returns a `FileContents` object. You can then call the `getInputStream` and `getOutputStream` methods to read and write the file data. If the user didn't choose a file, then the `openFileDialog` method returns `null`.

```
FileOpenService service = (FileOpenService) ServiceManager.lookup("javax.jnlp.FileOpenService");
FileContents contents = service.openFileDialog(".", new String[] { "txt" });
if (contents != null)
{
    InputStream in = contents.getInputStream();
    . . .
}
```

Note that your application does not know the name or location of the file. Conversely, if you want to open a specific file, you use the `ExtendedService`:

```
ExtendedService service = (ExtendedService) ServiceManager.lookup("javax.jnlp.ExtendedService");
FileContents contents = service.openFile(new File("c:\\autoexec.bat"));
if (contents != null)
{
    OutputStream out = contents.getOutputStream();
    . . .
}
```

The user of your program must agree to the file access (see Figure 10–9).



**Figure 10–9** File access warning

To display a document on the default browser, use the `BasicService` interface. Note that some systems may not have a default browser.

```
BasicService service = (BasicService) ServiceManager.lookup("javax.jnlp.BasicService");
if (service.isWebBrowserSupported())
    service.showDocument(url);
else . . .
```

A rudimentary `PersistenceService` lets an application store small amounts of configuration information and retrieve it when the application runs again. The mechanism is similar to HTTP cookies. The persistent store uses URLs as keys. The URLs don't have to point to a real web resource. The service simply uses them as a convenient hierarchical

naming scheme. For any given URL key, an application can store arbitrary binary data. (The store may restrict the size of the data block.)

So that applications are isolated from each other, a particular application can only use URL keys that start with its codebase (as specified in the JNLP file). For example, if an application is downloaded from `http://myserver.com/apps`, then it can only use keys of the form `http://myserver.com/apps/subkey1/subkey2/...` Attempts to access other keys will fail.

An application can call the `getCodeBase` method of the `BasicService` to find its codebase.

You create a new key with the `create` method of the `PersistenceService`.

```
URL url = new URL(codeBase, "mykey");
service.create(url, maxSize);
```

To access the information associated with a particular key, call the `get` method. That method returns a `FileContents` object through which you can read and write the key data. For example:

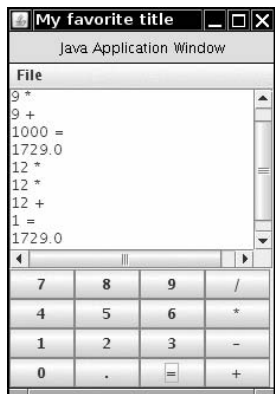
```
FileContents contents = service.get(url);
InputStream in = contents.getInputStream();
OutputStream out = contents.getOutputStream(true); // true = overwrite
```

Unfortunately, there is no convenient way to find out whether a key already exists or whether you need to create it. You can hope that the key exists and call `get`. If the call throws a `FileNotFoundException`, then you need to create the key.



**NOTE:** Starting with Java SE 5.0, both Java Web Start applications and applets can print, using the normal printing API. A security dialog pops up, asking the user for permission to access the printer. For more information on the printing API, turn to Chapter 7 of Volume II.

The program in Listing 10–2 is a simple enhancement of the calculator application. This calculator has a virtual paper tape that keeps track of all calculations. You can save and load the calculation history. To demonstrate the persistent store, the application lets you set the frame title. If you run the application again, it retrieves your title choice from the persistent store (see Figure 10–10).



**Figure 10–10** The WebStartCalculator application

**Listing 10-2** WebStartCalculator.java

```
1. import java.awt.EventQueue;
2. import java.awt.event.*;
3. import java.io.*;
4. import java.net.*;
5. import javax.swing.*;
6. import javax.jnlp.*;
7.
8. /**
9.  * A calculator with a calculation history that can be deployed as a Java Web Start application.
10.  * @version 1.02 2007-06-12
11.  * @author Cay Horstmann
12.  */
13. public class WebStartCalculator
14. {
15.     public static void main(String[] args)
16.     {
17.         EventQueue.invokeLater(new Runnable()
18.         {
19.             public void run()
20.             {
21.                 CalculatorFrame frame = new CalculatorFrame();
22.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23.                 frame.setVisible(true);
24.             }
25.         });
26.     }
27. }
28.
29. /**
30.  * A frame with a calculator panel and a menu to load and save the calculator history.
31.  */
32. class CalculatorFrame extends JFrame
33. {
34.     public CalculatorFrame()
35.     {
36.         setTitle();
37.         panel = new CalculatorPanel();
38.         add(panel);
39.
40.         JMenu fileMenu = new JMenu("File");
41.
42.         JMenuItem openItem = fileMenu.add("Open");
43.         openItem.addActionListener(new ActionListener()
44.         {
45.             public void actionPerformed(ActionEvent event)
46.             {
47.                 open();
48.             }
49.         });
```

**Listing 10-2** WebStartCalculator.java (continued)

```
50.
51.     JMenuItem saveItem = fileMenu.add("Save");
52.     saveItem.addActionListener(new ActionListener()
53.     {
54.         public void actionPerformed(ActionEvent event)
55.         {
56.             save();
57.         }
58.     });
59.     JMenuBar menuBar = new JMenuBar();
60.     menuBar.add(fileMenu);
61.     setJMenuBar(menuBar);
62.
63.     pack();
64. }
65.
66. /**
67.  * Gets the title from the persistent store or asks the user for the title if there is no
68.  * prior entry.
69.  */
70. public void setTitle()
71. {
72.     try
73.     {
74.         String title = null;
75.
76.         BasicService basic = (BasicService) ServiceManager.lookup("javax.jnlp.BasicService");
77.         URL codeBase = basic.getCodeBase();
78.
79.         PersistenceService service = (PersistenceService) ServiceManager
80.             .lookup("javax.jnlp.PersistenceService");
81.         URL key = new URL(codeBase, "title");
82.
83.         try
84.         {
85.             FileContents contents = service.get(key);
86.             InputStream in = contents.getInputStream();
87.             BufferedReader reader = new BufferedReader(new InputStreamReader(in));
88.             title = reader.readLine();
89.         }
90.         catch (FileNotFoundException e)
91.         {
92.             title = JOptionPane.showInputDialog("Please supply a frame title:");
93.             if (title == null) return;
94.
95.             service.create(key, 100);
96.             FileContents contents = service.get(key);
97.             OutputStream out = contents.getOutputStream(true);
98.             PrintStream printOut = new PrintStream(out);
99.             printOut.print(title);
```



**Listing 10-2** WebStartCalculator.java (continued)

```
100.     }
101.     setTitle(title);
102.     }
103.     catch (UnavailableServiceException e)
104.     {
105.         JOptionPane.showMessageDialog(this, e);
106.     }
107.     catch (MalformedURLException e)
108.     {
109.         JOptionPane.showMessageDialog(this, e);
110.     }
111.     catch (IOException e)
112.     {
113.         JOptionPane.showMessageDialog(this, e);
114.     }
115. }
116.
117. /**
118.  * Opens a history file and updates the display.
119.  */
120. public void open()
121. {
122.     try
123.     {
124.         FileOpenService service = (FileOpenService) ServiceManager
125.             .lookup("javax.jnlp.FileOpenService");
126.         FileContents contents = service.openFileDialog(".", new String[] { "txt" });
127.
128.         JOptionPane.showMessageDialog(this, contents.getName());
129.         if (contents != null)
130.         {
131.             InputStream in = contents.getInputStream();
132.             BufferedReader reader = new BufferedReader(new InputStreamReader(in));
133.             String line;
134.             while ((line = reader.readLine()) != null)
135.             {
136.                 panel.append(line);
137.                 panel.append("\n");
138.             }
139.         }
140.     }
141.     catch (UnavailableServiceException e)
142.     {
143.         JOptionPane.showMessageDialog(this, e);
144.     }
145.     catch (IOException e)
146.     {
147.         JOptionPane.showMessageDialog(this, e);
148.     }
149. }
```

**Listing 10-2** WebStartCalculator.java (continued)

```
150.
151.  /**
152.   * Saves the calculator history to a file.
153.   */
154.  public void save()
155.  {
156.      try
157.      {
158.          ByteArrayOutputStream out = new ByteArrayOutputStream();
159.          PrintStream printOut = new PrintStream(out);
160.          printOut.print(panel.getText());
161.          InputStream data = new ByteArrayInputStream(out.toByteArray());
162.          FileSaveService service = (FileSaveService) ServiceManager
163.              .lookup("javax.jnlp.FileSaveService");
164.          service.saveFileDialog(".", new String[] { "txt" }, data, "calc.txt");
165.      }
166.      catch (UnavailableServiceException e)
167.      {
168.          JOptionPane.showMessageDialog(this, e);
169.      }
170.      catch (IOException e)
171.      {
172.          JOptionPane.showMessageDialog(this, e);
173.      }
174.  }
175.
176.  private CalculatorPanel panel;
177. }
```

---

**API** javax.jnlp.ServiceManager

- static String[] getServiceNames()  
returns the names of all available services.
- static Object lookup(String name)  
returns a service with a given name.

**API** javax.jnlp.BasicService

- URL getCodeBase()  
returns the codebase of this application.
- boolean isWebBrowserSupported()  
returns true if the Web Start environment can launch a web browser.
- boolean showDocument(URL url)  
attempts to show the given URL in a browser. Returns true if the request succeeded.

**API** javax.jnlp.FileContents

- `InputStream getInputStream()`  
returns an input stream to read the contents of the file.
- `OutputStream getOutputStream(boolean overwrite)`  
returns an output stream to write to the file. If `overwrite` is `true`, then the existing contents of the file are overwritten.
- `String getName()`  
returns the file name (but not the full directory path).
- `boolean canRead()`
- `boolean canWrite()`  
returns `true` if the underlying file is readable or writable.

**API** javax.jnlp.FileOpenService

- `FileContents openFileDialog(String pathHint, String[] extensions)`
- `FileContents[] openMultiFileDialog(String pathHint, String[] extensions)`  
displays a user warning and a file chooser. Returns content descriptors of the file or files that the user selected, or `null` if the user didn't choose a file.

**API** javax.jnlp.FileSaveService

- `FileContents saveFileDialog(String pathHint, String[] extensions, InputStream data, String nameHint)`
- `FileContents saveAsFileDialog(String pathHint, String[] extensions, FileContents data)`  
displays a user warning and a file chooser. Writes the data and returns content descriptors of the file or files that the user selected, or `null` if the user didn't choose a file.

**API** javax.jnlp.PersistenceService

- `long create(URL key, long maxsize)`  
creates a persistent store entry for the given key. Returns the maximum size granted by the persistent store.
- `void delete(URL key)`  
deletes the entry for the given key.
- `String[] getNames(URL url)`  
returns the relative key names of all keys that start with the given URL.
- `FileContents get(URL key)`  
gets a content descriptor through which you can modify the data associated with the given key. If no entry exists for the key, a `FileNotFoundException` is thrown.

## Applets

Applets are Java programs that are included in an HTML page. The HTML page must tell the browser which applets to load and then where to put each applet on the web page. As you might expect, the tag needed to use an applet must tell the browser where to get the class files, and how the applet is positioned on the web page (size, location, and so on). The browser then retrieves the class files from the Internet (or from a directory on the user's machine) and automatically runs the applet.

When applets were first developed, you had to use Sun's HotJava browser to view web pages that contained applets. Naturally, few users were willing to use a separate browser just to enjoy a new web feature. Java applets became really popular when Netscape included a Java virtual machine in its Navigator browser. Microsoft Internet Explorer soon followed suit. Unfortunately, two problems happened. Netscape didn't keep up with more modern versions of Java, and Microsoft vacillated between reluctantly supporting outdated Java versions and dropping Java support altogether.

To overcome this problem, Sun released a tool called the "Java Plug-in." Using the various extension mechanisms, it seamlessly plugs in to a variety of browsers and enables them to execute Java applets by using an external Java runtime environment that Sun supplies. By keeping the Plug-in up-to-date, you can always take advantage of the latest and greatest features of Java.



**NOTE:** To run the applets in this chapter in a browser, you need to install the current version of the Java Plug-in and make sure your browser is connected with the Plug-in. Go to <http://java.com> for download and configuration information.

---

## A Simple Applet

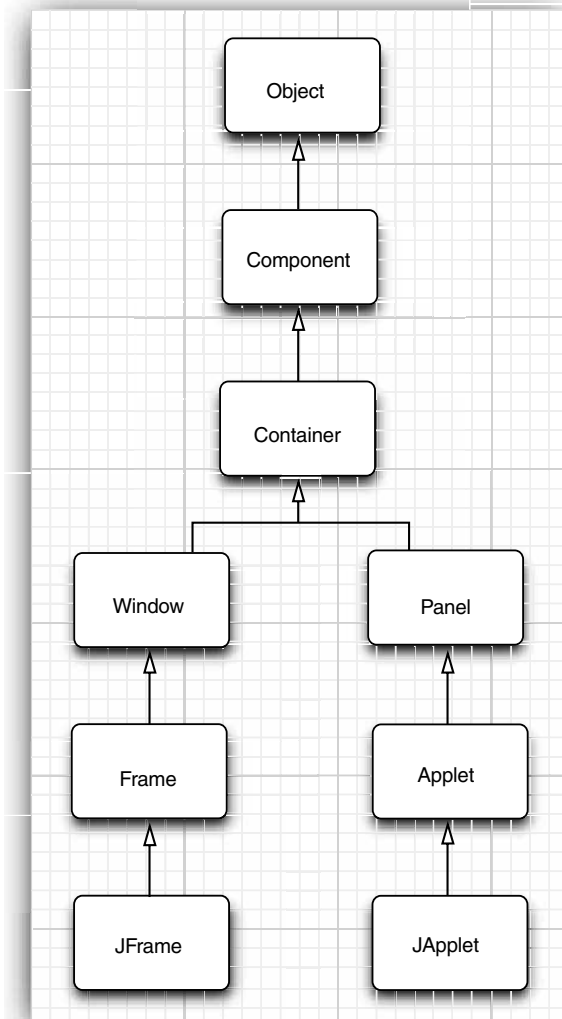
For tradition's sake, let's write a `NotHelloWorld` program as an applet. An applet is simply a Java class that extends the `java.applet.Applet` class. In this book, we will use Swing to implement applets. All of our applets will extend the `JApplet` class, the superclass for Swing applets. As you can see in Figure 10-11, `JApplet` is an immediate subclass of the ordinary `Applet` class.



**NOTE:** If your applet contains Swing components, you must extend the `JApplet` class. Swing components inside a plain `Applet` don't paint correctly.

---

Listing 10-3 on page 518 shows the code for an applet version of "Not Hello World". Notice how similar this is to the corresponding program from Chapter 7. However, because the applet lives inside a web page, there is no need to specify a method for exiting the applet.

**Figure 10-11** Applet inheritance diagram

**Listing 10-3** NotHelloWorldApplet.java

```
1. /*
2.  * The following HTML tags are required to display this applet in a browser: <applet
3.  * code="NotHelloWorldApplet.class" width="300" height="100"> </applet>
4.  */
5.
6. import java.awt.*;
7. import javax.swing.*;
8.
9. /**
10.  * @version 1.22 2007-06-12
11.  * @author Cay Horstmann
12.  */
13. public class NotHelloWorldApplet extends JApplet
14. {
15.     public void init()
16.     {
17.        .EventQueue.invokeLater(new Runnable()
18.         {
19.             public void run()
20.             {
21.                 JLabel label = new JLabel("Not a Hello, World applet", SwingConstants.CENTER);
22.                 add(label);
23.             }
24.         });
25.     }
26. }
```

To execute the applet, you carry out two steps:

1. Compile your Java source files into class files.
2. Create an HTML file that tells the browser which class file to load first and how to size the applet.

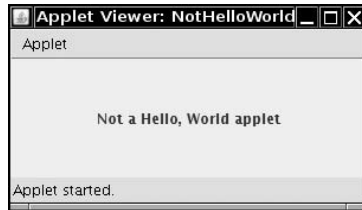
It is customary (but not necessary) to give the HTML file the same name as that of the applet class inside. So, following this tradition, we call the file `NotHelloWorldApplet.html`. Here are the contents of the file:

```
<applet code="NotHelloWorldApplet.class" width="300" height="300">
</applet>
```

Before you view the applet in a browser, it is a good idea to test it in the *applet viewer* program that is a part of the JDK. To use the applet viewer in our example, enter

```
appletviewer NotHelloWorldApplet.html
```

at the command line. The command-line argument for the applet viewer program is the name of the HTML file, not the class file. Figure 10-12 shows the applet viewer displaying this applet.



**Figure 10-12** Viewing an applet in the applet viewer



**TIP:** Here is a weird trick to avoid the additional HTML file. Add an applet tag as a *comment* inside the source file:

```
/*
  <applet code="MyApplet.class" width="300" height="300">
  </applet>
*/
public class MyApplet extends JApplet
. . .
```

Then run the applet viewer *with the source file* as its command-line argument:

```
appletviewer NotHelloWorldApplet.java
```

We aren't recommending this as standard practice, but it can come in handy if you want to minimize the number of files that you need to worry about during testing.



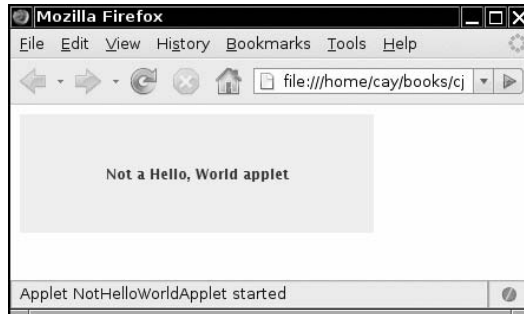
**TIP:** You can also run applets from inside your integrated environment. In Eclipse, you select Run -> Run as -> Java Applet from the menu.

The applet viewer is good for the first stage of testing, but at some point you need to run your applets in a browser to see them in the same way a user might use them. In particular, the applet viewer program shows you only the applet, not the surrounding HTML text. If an HTML file contains multiple applet tags, the applet viewer pops up multiple windows.

To properly view the applet, simply load the HTML file into the browser (see Figure 10-13). If the applet doesn't show up, you need to install the Java Plug-in.



**TIP:** If you make a change to your applet and recompile, you need to restart the browser so that it loads the new class files. Simply refreshing the HTML page will not load the new code. This is a hassle when you are debugging an applet. You can avoid the painful browser restart from the *Java console*. Launch the console and issue the `x` command, which clears the classloader cache. Then you can reload the HTML page, and the new applet code is used. Under Windows, open the Java Plug-in control in the Windows control panel. Under Linux, run `jcontrol` and request that the Java console be displayed. The console will pop up whenever an applet is loaded.




**Figure 10-13** Viewing an applet in a browser

### Converting Applications to Applets

It is easy to convert a graphical Java application into an applet that you can embed in a web page. Essentially, all of the user interface code can stay the same. Here are the specific steps:

1. Make an HTML page with the appropriate tag to load the applet code.
2. Supply a subclass of the `JApplet` class. Make this class `public`. Otherwise, the applet cannot be loaded.
3. Eliminate the `main` method in the application. Do not construct a frame window for the application. Your application will be displayed inside the browser.
4. Move any initialization code from the frame window constructor to the `init` method of the applet. You don't need to explicitly construct the applet object—the browser instantiates it for you and calls the `init` method.
5. Remove the call to `setSize`; for applets, sizing is done with the `width` and `height` parameters in the HTML file.
6. Remove the call to `setDefaultCloseOperation`. An applet cannot be closed; it terminates when the browser exits.
7. If the application calls `setTitle`, eliminate the call to the method. Applets cannot have title bars. (You can, of course, title the web page itself, using the HTML `title` tag.)
8. Don't call `setVisible(true)`. The applet is displayed automatically.

 **NOTE:** On page 532, you will see how to implement a program that is both an applet and an application.

#### **API** `java.applet.Applet 1.0`

- `void init()`  
is called when the applet is first loaded. Override this method and place all initialization code here.



- `void start()`  
override this method for code that needs to be executed *every time* the user visits the browser page containing this applet. A typical action is to reactivate a thread.
- `void stop()`  
override this method for code that needs to be executed *every time* the user leaves the browser page containing this applet. A typical action is to deactivate a thread.
- `void destroy()`  
override this method for code that needs to be executed when the user exits the browser.
- `void resize(int width, int height)`  
requests that the applet be resized. This would be a great method if it worked on web pages; unfortunately, it does not work in current browsers because it interferes with their page-layout mechanisms.

### **The Applet HTML Tag and Its Attributes**

In its most basic form, an example for using the applet tag looks like this:

```
<applet code="NotHelloWorldApplet.class" width="300" height="100">
```

As you have seen, the `code` attribute gives the name of the class file and must include the `.class` extension; the `width` and `height` attributes size the window that will hold the applet. Both are measured in pixels. You also need a matching `</applet>` tag that marks the end of the HTML tagging needed for an applet. The text between the `<applet>` and `</applet>` tags is displayed only if the browser cannot show applets. The `code`, `width`, and `height` attributes are required. If any are missing, the browser cannot load your applet.

All this information would usually be embedded in an HTML page that, at the very least, might look like this:

```
<html>
  <head>
    <title>NotHelloWorldApplet</title>
  </head>
  <body>
    <p>The next line of text is displayed under the auspices of Java:</p>
    <applet code="NotHelloWorldApplet.class" width="100" height="100">
      If your browser could show Java, you would see an applet here.
    </applet>
  </body>
</html>
```

You can use the following attributes within the applet tag:

- `width`, `height`  
These attributes are required and give the width and height of the applet, measured in pixels. In the applet viewer, this is the initial size of the applet. You can resize any window that the applet viewer creates. In a browser, you *cannot* resize the applet. You will need to make a good guess about how much space your applet requires to show up well for all users.
- `align`  
This attribute specifies the alignment of the applet. The attribute values are the same as for the `align` attribute of the HTML `img` tag.

- `vspace`, `hspace`  
These optional attributes specify the number of pixels above and below the applet (`vspace`) and on each side of the applet (`hspace`).
- `code`  
This attribute gives the name of the applet's class file. This name is taken relative to the codebase (see below) or relative to the current page if the codebase is not specified.  
The path name must match the package of the applet class. For example, if the applet class is in the package `com.mycompany`, then the attribute is `code="com/mycompany/MyApplet.class"`. The alternative `code="com.mycompany.MyApplet.class"` is also permitted. But you cannot use absolute path names here. Use the `codebase` attribute if your class file is located elsewhere.  
The `code` attribute specifies only the name of the class that contains the applet class. Of course, your applet may contain other class files. Once the browser's class loader loads the class containing the applet, it will realize that it needs more class files and will load them.  
Either the `code` or the `object` attribute (see below) is required.
- `codebase`  
This optional attribute is a URL for locating the class files. You can use an absolute URL, even to a different server. Most commonly, though, this is a relative URL that points to a subdirectory. For example, if the file layout looks like this:  

```

aDirectory/
├── MyPage.html
└── myApplets/
    └── MyApplet.class

```

then you use the following tag in `MyPage.html`:  

```

<applet code="MyApplet.class" codebase="myApplets" width="100" height="150">

```
- `archive`  
This optional attribute lists the JAR file or files containing classes and other resources for the applet. These files are fetched from the web server before the applet is loaded. This technique speeds up the loading process significantly because only one HTTP request is necessary to load a JAR file that contains many smaller files. The JAR files are separated by commas. For example:  

```

<applet code="MyApplet.class"
  archive="MyClasses.jar,corejava/CoreJavaClasses.jar"
  width="100" height="150">

```
- `object`  
This tag lets you specify the name of a file that contains the *serialized* applet object. (An object is *serialized* when you write all its instance fields to a file. We discuss serialization in Chapter 1 of Volume II.) To display the applet, the object is deserialized from the file to return it to its previous state. When you use this attribute, the `init` method is *not* called, but the applet's `start` method is called. Before serializing an applet object, you should call its `stop` method. This feature is useful for implementing a persistent browser that automatically reloads its applets and has them return to the same state that they were in when the browser was closed. This is a specialized feature, not normally encountered by web page designers.

Either `code` or `object` must be present in every applet tag. For example:

```
<applet object="MyApplet.ser" width="100" height="150">
```

- **name**

Scripters will want to give the applet a `name` attribute that they can use to refer to the applet when scripting. Both Netscape and Internet Explorer let you call methods of an applet on a page through JavaScript. This is not a book on JavaScript, so we only give you a brief idea of the code that is required to call Java code from JavaScript.



**NOTE:** JavaScript is a scripting language that can be used inside web pages, invented by Netscape and originally called LiveScript. It has little to do with Java, except for some similarity in syntax. It was a marketing move to call it JavaScript. A subset (with the catchy name of ECMAScript) is standardized as ECMA-262. But, to nobody's surprise, Netscape and Microsoft support incompatible extensions of that standard in their browsers. For more information on JavaScript, we recommend the book *JavaScript: The Definitive Guide* by David Flanagan (O'Reilly Media, Inc., 2006).

To access an applet from JavaScript, you first have to give it a name.

```
<applet code="MyApplet.class" width="100" height="150" name="mine">
</applet>
```

You can then refer to the object as `document.applets.appletname`. For example:

```
var myApplet = document.applets.mine;
```

Through the magic of the integration between Java and JavaScript that both Netscape and Internet Explorer provide, you can call applet methods:

```
myApplet.init();
```

The `name` attribute is also essential when you want two applets on the same page to communicate with each other directly. You specify a name for each current applet instance. You pass this string to the `getApplet` method of the `AppletContext` class. We discuss this mechanism, called *inter-applet communication*, later in this chapter.



**NOTE:** In <http://www.javaworld.com/javatips/jw-jvatip80.html>, Francis Lu uses JavaScript-to-Java communication to solve an age-old problem: to resize an applet so that it isn't bound by hardcoded width and height attributes. This is a good example of the integration between Java and JavaScript.

- **alt**

Java may be deactivated in the browser, perhaps by a paranoid system administrator. You can then use the `alt` attribute to display a message to these unfortunate souls.

```
<applet code="MyApplet.class" width="100" height="150"
alt="If you activated Java, you would see my applet here">
```

If a browser cannot process applets at all, it ignores the `unknownapplet` and `param` tags. All text between the `<applet>` and `</applet>` tags is displayed by the browser. Conversely, Java-aware browsers do not display any text between the `<applet>` and `</applet>` tags. You can display messages inside these tags for those poor folks that use a prehistoric browser. For example:

```
<applet code="MyApplet.class" width="100" height="150">
  If your browser could show Java, you would see my applet here.
</applet>
```

### **The object Tag**

The object tag is part of the HTML 4.0 standard, and the W3 consortium suggests that people use it instead of the applet tag. There are 35 different attributes to the object tag, most of which (such as `onkeydown`) are relevant only to people writing Dynamic HTML. The various positioning attributes such as `align` and `height` work exactly as they did for the applet tag. The key attribute in the object tag for your Java applets is the `classid` attribute. This attribute specifies the location of the object. Of course, object tags can load different kinds of objects, such as Java applets or ActiveX components like the Java Plug-in itself. In the `codetype` attribute, you specify the nature of the object. For example, Java applets have a code type of `application/java`. Here is an object tag to load a Java applet:

```
<object
  codetype="application/java"
  classid="java:MyApplet.class"
  width="100" height="150">
```

Note that the `classid` attribute can be followed by a `codebase` attribute that works exactly as it did with the applet tag.

### **Use of Parameters to Pass Information to Applets**

Just as applications can use command-line information, applets can use parameters that are embedded in the HTML file. This is done by the HTML tag called `param` along with attributes that you define. For example, suppose you want to let the web page determine the style of the font to use in your applet. You could use the following HTML tags:

```
<applet code="FontParamApplet.class" width="200" height="200">
  <param name="font" value="Helvetica"/>
</applet>
```

You then pick up the value of the parameter, using the `getParameter` method of the `Applet` class, as in the following example:

```
public class FontParamApplet extends JApplet
{
  public void init()
  {
    String fontName = getParameter("font");
    . . .
  }
  . . .
}
```



**NOTE:** You can call the `getParameter` method only in the `init` method of the applet, not in the constructor. When the applet constructor is executed, the parameters are not yet prepared. Because the layout of most nontrivial applets is determined by parameters, we recommend that you don't supply constructors to applets. Simply place all initialization code into the `init` method.

---

Parameters are always returned as strings. You need to convert the string to a numeric type if that is what is called for. You do this in the standard way by using the appropriate method, such as `parseInt` of the `Integer` class.

For example, if we wanted to add a size parameter for the font, then the HTML code might look like this:

```
<applet code="FontParamApplet.class" width="200" height="200">
  <param name="font" value="Helvetica"/>
  <param name="size" value="24"/>
</applet>
```

The following source code shows how to read the integer parameter:

```
public class FontParamApplet extends JApplet
{
    public void init()
    {
        String fontName = getParameter("font");
        int fontSize = Integer.parseInt(getParameter("size"));
        . . .
    }
}
```



**NOTE:** A case-insensitive comparison is used when matching the name attribute value in the param tag and the argument of the `getParameter` method.

In addition to ensuring that the parameters match in your code, you should find out whether or not the size parameter was left out. You do this with a simple test for `null`.

For example:

```
int fontsize;
String sizeString = getParameter("size");
if (sizeString == null) fontsize = 12;
else fontsize = Integer.parseInt(sizeString);
```

Here is a useful applet that uses parameters extensively. The applet draws a bar chart, shown in Figure 10–14.

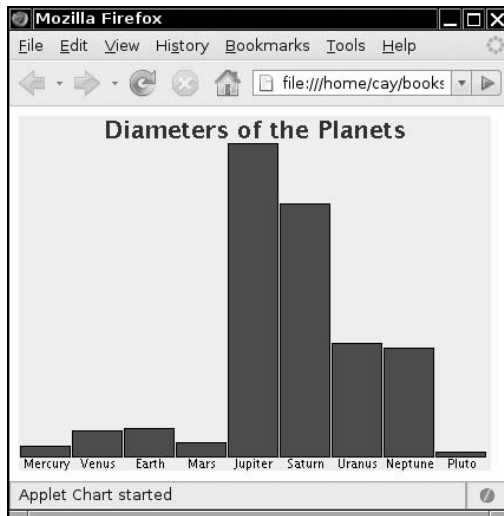
This applet takes the labels and the heights of the bars from the param values in the HTML file. Here is what the HTML for Figure 10–14 looks like:

```
<applet code="Chart.class" width="400" height="300">
  <param name="title" value="Diameters of the Planets"/>
  <param name="values" value="9"/>
  <param name="name.1" value="Mercury"/>
  <param name="name.2" value="Venus"/>
  <param name="name.3" value="Earth"/>
  <param name="name.4" value="Mars"/>
  <param name="name.5" value="Jupiter"/>
  <param name="name.6" value="Saturn"/>
  <param name="name.7" value="Uranus"/>
  <param name="name.8" value="Neptune"/>
  <param name="name.9" value="Pluto"/>
  <param name="value.1" value="3100"/>
```

```

<param name="value.2" value="7500"/>
<param name="value.3" value="8000"/>
<param name="value.4" value="4200"/>
<param name="value.5" value="88000"/>
<param name="value.6" value="71000"/>
<param name="value.7" value="32000"/>
<param name="value.8" value="30600"/>
<param name="value.9" value="1430"/>
</applet>

```



**Figure 10-14** A chart applet

You could have set up an array of strings and an array of numbers in the applet, but there are two advantages to using the parameter mechanism instead. You can have multiple copies of the same applet on your web page, showing different graphs: just put two applet tags with different sets of parameters on the page. And you can change the data that you want to chart. Admittedly, the diameters of the planets will stay the same for quite some time, but suppose your web page contains a chart of weekly sales data. It is easy to update the web page because it is plain text. Editing and recompiling a Java file weekly is more tedious.

In fact, there are commercial JavaBeans components (beans) that make much fancier graphs than the one in our chart applet. If you buy one, you can drop it into your web page and feed it parameters without ever needing to know how the applet renders the graphs.

Listing 10-4 is the source code of our chart applet. Note that the `init` method reads the parameters, and the `paintComponent` method draws the chart.

**Listing 10-4** Chart.java

```
1. import java.awt.*;
2. import java.awt.font.*;
3. import java.awt.geom.*;
4. import javax.swing.*;
5.
6. /**
7.  * @version 1.33 2007-06-12
8.  * @author Cay Horstmann
9.  */
10. public class Chart extends JApplet
11. {
12.     public void init()
13.     {
14.         EventQueue.invokeLater(new Runnable()
15.         {
16.             public void run()
17.             {
18.                 String v = getParameter("values");
19.                 if (v == null) return;
20.                 int n = Integer.parseInt(v);
21.                 double[] values = new double[n];
22.                 String[] names = new String[n];
23.                 for (int i = 0; i < n; i++)
24.                 {
25.                     values[i] = Double.parseDouble(getParameter("value." + (i + 1)));
26.                     names[i] = getParameter("name." + (i + 1));
27.                 }
28.
29.                 add(new ChartComponent(values, names, getParameter("title")));
30.             }
31.         });
32.     }
33. }
34.
35. /**
36.  * A component that draws a bar chart.
37.  */
38. class ChartComponent extends JComponent
39. {
40.     /**
41.      * Constructs a ChartComponent.
42.      * @param v the array of values for the chart
43.      * @param n the array of names for the values
44.      * @param t the title of the chart
45.      */
46.     public ChartComponent(double[] v, String[] n, String t)
47.     {
48.         values = v;
49.         names = n;
50.         title = t;
```

**Listing 10-4** Chart.java (continued)

```
51. }
52.
53. public void paintComponent(Graphics g)
54. {
55.     Graphics2D g2 = (Graphics2D) g;
56.
57.     // compute the minimum and maximum values
58.     if (values == null) return;
59.     double minValue = 0;
60.     double maxValue = 0;
61.     for (double v : values)
62.     {
63.         if (minValue > v) minValue = v;
64.         if (maxValue < v) maxValue = v;
65.     }
66.     if (maxValue == minValue) return;
67.
68.     int panelWidth = getWidth();
69.     int panelHeight = getHeight();
70.
71.     Font titleFont = new Font("SansSerif", Font.BOLD, 20);
72.     Font labelFont = new Font("SansSerif", Font.PLAIN, 10);
73.
74.     // compute the extent of the title
75.     FontRenderContext context = g2.getFontRenderContext();
76.     Rectangle2D titleBounds = titleFont.getStringBounds(title, context);
77.     double titleWidth = titleBounds.getWidth();
78.     double top = titleBounds.getHeight();
79.
80.     // draw the title
81.     double y = -titleBounds.getY(); // ascent
82.     double x = (panelWidth - titleWidth) / 2;
83.     g2.setFont(titleFont);
84.     g2.drawString(title, (float) x, (float) y);
85.
86.     // compute the extent of the bar labels
87.     LineMetrics labelMetrics = labelFont.getLineMetrics("", context);
88.     double bottom = labelMetrics.getHeight();
89.
90.     y = panelHeight - labelMetrics.getDescent();
91.     g2.setFont(labelFont);
92.
93.     // get the scale factor and width for the bars
94.     double scale = (panelHeight - top - bottom) / (maxValue - minValue);
95.     int barWidth = panelWidth / values.length;
96.
97.     // draw the bars
98.     for (int i = 0; i < values.length; i++)
99.     {
```



**Listing 10-4** Chart.java (continued)

```

100.    // get the coordinates of the bar rectangle
101.    double x1 = i * barWidth + 1;
102.    double y1 = top;
103.    double height = values[i] * scale;
104.    if (values[i] >= 0) y1 += (maxValue - values[i]) * scale;
105.    else
106.    {
107.        y1 += maxValue * scale;
108.        height = -height;
109.    }
110.
111.    // fill the bar and draw the bar outline
112.    Rectangle2D rect = new Rectangle2D.Double(x1, y1, barWidth - 2, height);
113.    g2.setPaint(Color.RED);
114.    g2.fill(rect);
115.    g2.setPaint(Color.BLACK);
116.    g2.draw(rect);
117.
118.    // draw the centered label below the bar
119.    Rectangle2D labelBounds = labelFont.getStringBounds(names[i], context);
120.
121.    double labelWidth = labelBounds.getWidth();
122.    x = x1 + (barWidth - labelWidth) / 2;
123.    g2.drawString(names[i], (float) x, (float) y);
124. }
125. }
126.
127. private double[] values;
128. private String[] names;
129. private String title;
130. }

```

**API** java.applet.Applet 1.0

- `public String getParameter(String name)`  
gets the value of a parameter defined with a `param` tag in the web page loading the applet. The string `name` is case sensitive.
- `public String getAppletInfo()`  
is a method that many applet authors override to return a string that contains information about the author, version, and copyright of the current applet. You need to create this information by overriding this method in your applet class.
- `public String[][] getParameterInfo()`  
is a method that you can override to return an array of `param` tag options that this applet supports. Each row contains three entries: the name, the type, and a description of the parameter. Here is an example:  

```

"fps", "1-10", "frames per second"
"repeat", "boolean", "repeat image loop?"
"images", "url", "directory containing images"

```

**Accessing Image and Audio Files**

Applets can handle both images and audio. As we write this, images must be in GIF, PNG, or JPEG form, audio files in AU, AIFF, WAV, or MIDI. Animated GIFs are supported, and the animation is displayed.

You specify the locations of image and audio files with relative URLs. The base URL is usually obtained by calling the `getDocumentBase` or `getCodeBase` method. The former gets the URL of the HTML page in which the applet is contained, the latter the URL of the applet's codebase directory.



NOTE: In prior versions of Java SE, there was considerable confusion about these methods—see bug #4456393 on the Java bug parade (<http://bugs.sun.com/bugdatabase/index.jsp>). The documentation was finally clarified in Java SE 5.0.

Give the base URL and the file location to the `getImage` or `getAudioClip` method. For example:

```
Image cat = getImage(getCodeBase(), "images/cat.gif");
AudioClip meow = getAudioClip(getCodeBase(), "audio/meow.au");
```

You saw in Chapter 7 how to display an image. To play an audio clip, simply invoke its `play` method. You can also call the `play` method of the `Applet` class without first loading the audio clip.

```
play(getCodeBase(), "audio/meow.au");
```

**API java.applet.Applet 1.0**

- `URL getDocumentBase()`  
gets the URL of the web page containing this applet.
- `URL getCodeBase()`  
gets the URL of the codebase directory from which this applet is loaded. That is either the absolute URL of the directory referenced by the `codebase` attribute or the directory of the HTML file if no `codebase` is specified.
- `void play(URL url)`
- `void play(URL url, String name)`  
The first form plays an audio file specified by the URL. The second form uses the string to provide a path relative to the URL in the first parameter. Nothing happens if the audio clip cannot be found.
- `AudioClip getAudioClip(URL url)`
- `AudioClip getAudioClip(URL url, String name)`  
The first form gets an audio clip from the given URL. The second form uses the string to provide a path relative to the URL in the first argument. The methods return `null` if the audio clip cannot be found.
- `Image getImage(URL url)`
- `Image getImage(URL url, String name)`  
returns an image object that encapsulates the image specified by the URL. If the image does not exist, immediately returns `null`. Otherwise, a separate thread is launched to load the image.

### The Applet Context

An applet runs inside a browser or the applet viewer. An applet can ask the browser to do things for it, for example, fetch an audio clip, show a short message in the status line, or display a different web page. The ambient browser can carry out these requests, or it can ignore them. For example, if an applet running inside the applet viewer asks the applet viewer program to display a web page, nothing happens.

To communicate with the browser, an applet calls the `getAppletContext` method. That method returns an object that implements an interface of type `AppletContext`. You can think of the concrete implementation of the `AppletContext` interface as a communication path between the applet and the ambient browser. In addition to `getAudioClip` and `getImage`, the `AppletContext` interface contains several useful methods, which we discuss in the next few sections.

### Inter-Applet Communication

A web page can contain more than one applet. If a web page contains multiple applets from the same codebase, they can communicate with each other. Naturally, this is an advanced technique that you probably will not need very often.

If you give `name` attributes to each applet in the HTML file, you can use the `getApplet` method of the `AppletContext` interface to get a reference to the applet. For example, if your HTML file contains the tag

```
<applet code="Chart.class" width="100" height="100" name="Chart1">
```

then the call

```
Applet chart1 = getAppletContext().getApplet("Chart1");
```

gives you a reference to the applet. What can you do with the reference? Provided you give the `Chart` class a method to accept new data and redraw the chart, you can call this method by making the appropriate cast.

```
((Chart) chart1).setData(3, "Earth", 9000);
```

You can also list all applets on a web page, whether or not they have a `name` attribute. The `getApplets` method returns an *enumeration object*. (You learn more about enumeration objects in Chapter 13.) Here is a loop that prints the class names of all applets on the current page:

```
Enumeration<Applet> e = getAppletContext().getApplets();
while (e.hasMoreElements())
{
    Applet a = e.nextElement();
    System.out.println(a.getClass().getName());
}
```

An applet cannot communicate with an applet on a different web page.

### Display of Items in the Browser

You have access to two areas of the ambient browsers: the status line and the web page display area. Both use methods of the `AppletContext` class.

You can display a string in the status line at the bottom of the browser with the `showStatus` message. For example:

```
showStatus("Loading data . . . please wait");
```



TIP: In our experience, `showStatus` is of limited use. The browser is also using the status line, and, more often than not, it will overwrite your precious message with chatter like “Applet running.” Use the status line for fluff messages like “Loading data . . . please wait,” but not for something that the user cannot afford to miss.

You can tell the browser to show a different web page with the `showDocument` method. There are several ways to do this. The simplest is with a call to `showDocument` with one argument, the URL you want to show.

```
URL u = new URL("http://java.sun.com/index.html");
getAppletContext().showDocument(u);
```

The problem with this call is that it opens the new web page in the same window as your current page, thereby displacing your applet. To return to your applet, the user must click the Back button of the browser.

You can tell the browser to show the document in another window by giving a second parameter in the call to `showDocument` (see Table 10–2). If you supply the special string “\_blank”, the browser opens a new window with the document, instead of displacing the current document. More important, if you take advantage of the frame feature in HTML, you can split a browser window into multiple frames, each of which has a name. You can put your applet into one frame and have it show documents in other frames. We show you an example of how to do this in the next section.

**Table 10–2 The `showDocument` Method**

Target Parameter	Location
“_self” or none	Show the document in the current frame.
“_parent”	Show the document in the parent frame.
“_top”	Show the document in the topmost frame.
“_blank”	Show in new, unnamed, top-level window.
Any other string	Show in the frame with that name. If no frame with that name exists, open a new window and give it that name.



NOTE: Sun’s applet viewer does not show web pages. The `showDocument` method is ignored in the applet viewer.

### **It’s an Applet. It’s an Application. It’s Both!**

Quite a few years ago, a *Saturday Night Live* skit poked fun at a television commercial, showing a couple that argued about a gelatinous substance. The husband said, “It’s a dessert topping.” The wife said, “It’s a floor wax.” And the announcer concluded triumphantly, “It’s both!”

Well, in this section, we show you how to turn an applet into a Java program that is *both* an applet and an application. That is, you can load the program with the applet viewer or a browser, or you can start it from the command line with the `java` program launcher. We are not sure how often this comes up—we found it interesting that this could be done at all and thought you would, too.

The screen shots in Figures 10–15 and 10–16 show the *same* program, viewed inside the applet viewer as an applet and launched from the command line as an application.



**Figure 10–15** It's an applet!



**Figure 10–16** It's an application!

A GUI application constructs a frame object and invokes `setVisible(true)` on it. Since you cannot call `setVisible` on a naked applet, the applet must be placed inside a frame. We provide a class `AppletFrame` whose constructor adds the applet into the content pane:

```
public class AppletFrame extends JFrame
{
    public AppletFrame(Applet anApplet)
    {
        applet = anApplet;
        add(applet);
        . . .
    }
    . . .
}
```

In the main method of the applet/application, we construct and show an `AppletFrame`.

```
class MyAppletApplication extends MyApplet
{
    public static void main(String args[])
    {
        AppletFrame frame = new AppletFrame(new MyApplet());
        frame.setTitle("MyAppletApplication");
        frame.setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

(Note that we simply extend the existing applet.)

When the applet starts, its `init` and `start` methods must be called. We achieve this by overriding the `setVisible` method of the `AppletFrame` class:

```
public void setVisible(boolean b)
{
    if (b)
    {
        applet.init();
        super.setVisible(true);
        applet.start();
    }
    else
    {
        applet.stop();
        super.setVisible(false);
        applet.destroy();
    }
}
```

There is one catch. If the program is started as an application, and it calls `getAppletContext`, it gets a `null` pointer because it has not been launched inside a browser. This causes a runtime crash whenever we have code like

```
getAppletContext().showStatus(message);
```

While we do not want to write a full-fledged browser, we do need to supply the bare minimum to make calls like this work. The call displays no message, but at least it will not crash the program. It turns out that all we need to do is implement two interfaces: `AppletStub` and `AppletContext`. The major purpose of the `AppletStub` interface is to locate the applet context. Every applet has an applet stub (set with the `setStub` method of the `Applet` class). We supply the bare minimum functionality that is necessary to implement these two interfaces.

```
public class AppletFrame extends JFrame
    implements AppletStub, AppletContext
{
    . . .
    // AppletStub methods
    public boolean isActive() { return true; }
    public URL getDocumentBase() { return null; }
```

```

public URL getCodeBase() { return null; }
public String getParameter(String name) { return ""; }
public AppletContext getAppletContext() { return this; }
public void appletResize(int width, int height) {}

// AppletContext methods
public AudioClip getAudioClip(URL url) { return null; }
public Image getImage(URL url) { return Toolkit.getDefaultToolkit().getImage(url); }
public Applet getApplet(String name) { return null; }
public Enumeration<Applet> getApplets() { return null; }
public void showDocument(URL url) {}
public void showDocument(URL url, String target) {}
public void showStatus(String status) {}
public void setStream(String key, InputStream stream) {}
public InputStream getStream(String key) { return null; }
public Iterator<String> getStreamKeys() { return null; }
}

```

Next, the constructor of the frame class calls `setStub` on the applet to make itself its stub.

```

public AppletFrame(Applet anApplet)
{
    applet = anApplet
    Container contentPane = getContentPane();
    contentPane.add(applet);
    applet.setStub(this);
}

```

Just for fun, we use the previously mentioned trick of adding the `applet` tag as a comment to the source file. Then, you can invoke the applet viewer with the source file without requiring an additional HTML file.

Listings 10–5 and 10–6 list the code. Try running both the applet and the application:

```

appletviewer NotHelloAppletApplication.java
java NotHelloAppletApplication

```

#### Listing 10–5 AppletFrame.java

```

1. import java.awt.*;
2. import java.applet.*;
3. import java.io.*;
4. import java.net.*;
5. import java.util.*;
6. import javax.swing.*;
7.
8. /**
9.  * @version 1.32 2007-06-12
10.  * @author Cay Horstmann
11.  */
12. public class AppletFrame extends JFrame implements AppletStub, AppletContext
13. {
14.     public AppletFrame(Applet anApplet)
15.     {

```

**Listing 10-5** AppletFrame.java (continued)

```
16.     applet = anApplet;
17.     add(applet);
18.     applet.setStub(this);
19. }
20.
21. public void setVisible(boolean b)
22. {
23.     if (b)
24.     {
25.         applet.init();
26.         super.setVisible(true);
27.         applet.start();
28.     }
29.     else
30.     {
31.         applet.stop();
32.         super.setVisible(false);
33.         applet.destroy();
34.     }
35. }
36.
37. // AppletStub methods
38. public boolean isActive()
39. {
40.     return true;
41. }
42.
43. public URL getDocumentBase()
44. {
45.     return null;
46. }
47.
48. public URL getCodeBase()
49. {
50.     return null;
51. }
52.
53. public String getParameter(String name)
54. {
55.     return "";
56. }
57.
58. public AppletContext getAppletContext()
59. {
60.     return this;
61. }
62.
63. public void appletResize(int width, int height)
64. {
65. }
```



**Listing 10-5** AppletFrame.java (continued)

```
66.
67. // AppletContext methods
68. public AudioClip getAudioClip(URL url)
69. {
70.     return null;
71. }
72.
73. public Image getImage(URL url)
74. {
75.     return Toolkit.getDefaultToolkit().getImage(url);
76. }
77.
78. public Applet getApplet(String name)
79. {
80.     return null;
81. }
82.
83. public Enumeration<Applet> getApplets()
84. {
85.     return null;
86. }
87.
88. public void showDocument(URL url)
89. {
90. }
91.
92. public void showDocument(URL url, String target)
93. {
94. }
95.
96. public void showStatus(String status)
97. {
98. }
99.
100. public void setStream(String key, InputStream stream)
101. {
102. }
103.
104. public InputStream getStream(String key)
105. {
106.     return null;
107. }
108.
109. public Iterator<String> getStreamKeys()
110. {
111.     return null;
112. }
113.
114. private Applet applet;
115. }
```

**Listing 10-6** AppletApplication.java

```
1. /*
2. * The applet viewer reads the tags below if you call it with appletviewer AppletApplication.java
3. * No separate HTML file is required. <applet code="AppletApplication.class"
4. * width="200" height="200">
5. * </applet>
6. */
7.
8. import java.awt.EventQueue;
9.
10. import javax.swing.*;
11.
12. /**
13. * It's an applet. It's an application. It's BOTH!
14. * @version 1.32 2007-04-28
15. * @author Cay Horstmann
16. */
17. public class AppletApplication extends NotHelloWorldApplet
18. {
19.     public static void main(String[] args)
20.     {
21.         EventQueue.invokeLater(new Runnable()
22.         {
23.             public void run()
24.             {
25.                 AppletFrame frame = new AppletFrame(new NotHelloWorldApplet());
26.                 frame.setTitle("NotHelloWorldApplet");
27.                 frame.setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
28.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
29.                 frame.setVisible(true);
30.             }
31.         });
32.     }
33.
34.     public static final int DEFAULT_WIDTH = 200;
35.     public static final int DEFAULT_HEIGHT = 200;
36. }
```

**API** java.applet.Applet 1.2

- `public AppletContext getAppletContext()`  
gives you a handle to the applet's browser environment. On most browsers, you can use this information to control the browser in which the applet is running.
- `void showStatus(String msg)`  
shows the string specified in the status line of the browser.

**API** java.applet.AppletContext 1.0

- Enumeration<Applet> getApplets()  
returns an enumeration (see Chapter 13) of all the applets in the same context, that is, the same web page.
- Applet getApplet(String name)  
returns the applet in the current context with the given name; returns null if none exists. Only the current web page is searched.
- void showDocument(URL url)
- void showDocument(URL url, String target)  
shows a new web page in a frame in the browser. In the first form, the new page displaces the current page. The second form uses the target parameter to identify the target frame (see Table 10–2 on page 532).

**Storage of Application Preferences**

Users of your applications will usually expect that their preferences and customizations are saved, and later restored when the application starts again. First, we cover the simple approach for storing configuration information in property files that Java applications have traditionally taken. We then turn to the powerful preferences mechanism that was introduced in Java SE 1.4.

**Property Maps**

A *property map* is a data structure that stores key/value pairs. Property maps are often used for storing configuration information. Property maps have three particular characteristics:

- The keys and values are strings.
- The set can easily be saved to a file and loaded from a file.
- There is a secondary table for default values.

The Java class that implements a property map is called `Properties`.

Property maps are useful in specifying configuration options for programs. For example:

```
Properties settings = new Properties();
settings.put("width", "200");
settings.put("title", "Hello, World!");
```

Use the `store` method to save this list of properties to a file. Here, we just save the property map in the file `program.properties`. The second argument is a comment that is included in the file.

```
FileOutputStream out = new FileOutputStream("program.properties");
settings.store(out, "Program Properties");
```

The sample set gives the following output:

```
#Program Properties
#Mon Apr 30 07:22:52 2007
width=200
title=Hello, World!
```

To load the properties from a file, use

```
FileInputStream in = new FileInputStream("program.properties");
settings.load(in);
```

It is customary to store program properties in a subdirectory of the user's home directory. The directory name is often chosen to start with a dot—on a UNIX system, this convention indicates a system directory that is hidden from the user. Our sample program follows this convention.

To find the user's home directory, you can call the `System.getProperties` method, which, as it happens, also uses a `Properties` object to describe the system information. The home directory has the key `"user.home"`. There is also a convenience method to read a single key:

```
String userDir = System.getProperty("user.home");
```

It is a good idea to provide defaults for our program properties, in case a user edits the file by hand. The `Properties` class has two mechanisms for providing defaults. First, whenever you look up the value of a string, you can specify a default that should be used automatically when the key is not present.

```
String title = settings.getProperty("title", "Default title");
```

If there is a `"title"` property in the property map, `title` is set to that string. Otherwise, `title` is set to `"Default title"`.

If you find it too tedious to specify the default in every call to `getProperty`, then you can pack all the defaults into a secondary property map and supply that map in the constructor of your primary property map.

```
Properties defaultSettings = new Properties();
defaultSettings.put("width", "300");
defaultSettings.put("height", "200");
defaultSettings.put("title", "Default title");
. . .
Properties settings = new Properties(defaultSettings);
```

Yes, you can even specify defaults to defaults if you give another property map parameter to the `defaultSettings` constructor, but it is not something one would normally do.

Listing 10-7 shows how you can use properties for storing and loading program state. The program remembers the frame position, size, and title. You can also manually edit the file `.corejava/program.properties` in your home directory to change the program's appearance to the way *you* want



**NOTE:** Properties are simple tables without a hierarchical structure. It is common to introduce a fake hierarchy with key names such as `window.main.color`, `window.main.title`, and so on. But the `Properties` class has no methods that help organize such a hierarchy. If you store complex configuration information, you should use the `Preferences` class instead—see the next section.

**Listing 10-7** PropertiesTest.java

```
1. import java.awt.EventQueue;
2. import java.awt.event.*;
3. import java.io.*;
4. import java.util.Properties;
5.
6. import javax.swing.*;
7.
8. /**
9.  * A program to test properties. The program remembers the frame position, size, and title.
10.  * @version 1.00 2007-04-29
11.  * @author Cay Horstmann
12.  */
13. public class PropertiesTest
14. {
15.     public static void main(String[] args)
16.     {
17.         EventQueue.invokeLater(new Runnable()
18.         {
19.             public void run()
20.             {
21.                 PropertiesFrame frame = new PropertiesFrame();
22.                 frame.setVisible(true);
23.             }
24.         });
25.     }
26. }
27.
28. /**
29.  * A frame that restores position and size from a properties file and updates the properties
30.  * upon exit.
31.  */
32. class PropertiesFrame extends JFrame
33. {
34.     public PropertiesFrame()
35.     {
36.         // get position, size, title from properties
37.
38.         String userDir = System.getProperty("user.home");
39.         File propertiesDir = new File(userDir, ".corejava");
40.         if (!propertiesDir.exists()) propertiesDir.mkdir();
41.         propertiesFile = new File(propertiesDir, "program.properties");
42.
43.         Properties defaultSettings = new Properties();
44.         defaultSettings.put("left", "0");
45.         defaultSettings.put("top", "0");
46.         defaultSettings.put("width", "" + DEFAULT_WIDTH);
47.         defaultSettings.put("height", "" + DEFAULT_HEIGHT);
48.         defaultSettings.put("title", "");
49.     }
50. }
```

**Listing 10-7** PropertiesTest.java (continued)

```
50.     settings = new Properties(defaultSettings);
51.
52.     if (propertiesFile.exists()) try
53.     {
54.         FileInputStream in = new FileInputStream(propertiesFile);
55.         settings.load(in);
56.     }
57.     catch (IOException ex)
58.     {
59.         ex.printStackTrace();
60.     }
61.
62.     int left = Integer.parseInt(settings.getProperty("left"));
63.     int top = Integer.parseInt(settings.getProperty("top"));
64.     int width = Integer.parseInt(settings.getProperty("width"));
65.     int height = Integer.parseInt(settings.getProperty("height"));
66.     setBounds(left, top, width, height);
67.
68.     // if no title given, ask user
69.
70.     String title = settings.getProperty("title");
71.     if (title.equals("")) title = JOptionPane.showInputDialog("Please supply a frame title:");
72.     if (title == null) title = "";
73.     setTitle(title);
74.
75.     addWindowListener(new WindowAdapter()
76.     {
77.         public void windowClosing(WindowEvent event)
78.         {
79.             settings.put("left", "" + getX());
80.             settings.put("top", "" + getY());
81.             settings.put("width", "" + getWidth());
82.             settings.put("height", "" + getHeight());
83.             settings.put("title", getTitle());
84.             try
85.             {
86.                 FileOutputStream out = new FileOutputStream(propertiesFile);
87.                 settings.store(out, "Program Properties");
88.             }
89.             catch (IOException ex)
90.             {
91.                 ex.printStackTrace();
92.             }
93.             System.exit(0);
94.         }
95.     });
96. }
97.
```

**Listing 10–7** PropertiesTest.java (continued)

```
98. private File propertiesFile;
99. private Properties settings;
100.
101. public static final int DEFAULT_WIDTH = 300;
102. public static final int DEFAULT_HEIGHT = 200;
103. }
```

**API** java.util.Properties 1.0

- Properties()  
creates an empty property map.
- Properties(Properties defaults)  
creates an empty property map with a set of defaults.  
*Parameters:* defaults      The defaults to use for lookups
- String getProperty(String key)  
gets a property map. Returns the string associated with the key, or the string associated with the key in the default table if it wasn't present in the table, or null if the key wasn't present in the default table either.  
*Parameters:* key      The key whose associated string to get
- String getProperty(String key, String defaultValue)  
gets a property with a default value if the key is not found. Returns the string associated with the key, or the default string if it wasn't present in the table.  
*Parameters:* key      The key whose associated string to get  
defaultValue      The string to return if the key is not present
- void load(InputStream in) throws IOException  
loads a property map from an input stream.  
*Parameters:* in      The input stream
- void store(OutputStream out, String header) 1.2  
saves a property map to an output stream.  
*Parameters:* out      The output stream  
header      The header in the first line of the stored file

**API** java.lang.System 1.0

- Properties getProperties()  
retrieves all system properties. The application must have permission to retrieve all properties or a security exception is thrown.

- String `getProperty(String key)`  
retrieves the system property with the given key name. The application must have permission to retrieve the property or a security exception is thrown. The following properties can always be retrieved:

```
java.version
java.vendor
java.vendor.url
java.class.version
os.name
os.version
os.arch
file.separator
path.separator
line.separator
java.specification.version
java.vm.specification.version
java.vm.specification.vendor
java.vm.specification.name
java.vm.version
java.vm.vendor
java.vm.name
```



NOTE: You can find the names of the freely accessible system properties in the file `security/java.policy` in the directory of the Java runtime.

### **The Preferences API**

As you have seen, the `Properties` class makes it simple to load and save configuration information. However, using property files has these disadvantages:

- Configuration files cannot always be stored in the user's home directory. Some operating systems (such as Windows 9x) have no concept of a home directory.
- There is no standard convention for naming configuration files, increasing the likelihood of name clashes as users install multiple Java applications.

Some operating systems have a central repository for configuration information. The best-known example is the registry in Microsoft Windows. The `Preferences` class of Java SE 1.4 provides such a central repository in a platform-independent manner. In Windows, the `Preferences` class uses the registry for storage; on Linux, the information is stored in the local file system instead. Of course, the repository implementation is transparent to the programmer using the `Preferences` class.

The `Preferences` repository has a tree structure, with node path names such as `/com/mycompany/myapp`. As with package names, name clashes are avoided as long as programmers start the paths with reversed domain names. In fact, the designers of the API suggest that the configuration node paths match the package names in your program.

Each node in the repository has a separate table of key/value pairs that you can use to store numbers, strings, or byte arrays. No provision is made for storing serializable objects. The API designers felt that the serialization format is too fragile for long-term storage. Of course, if you disagree, you can save serialized objects in byte arrays.



For additional flexibility, there are multiple parallel trees. Each program user has one tree, and an additional tree, called the system tree, is available for settings that are common to all users. The `Preferences` class uses the operating system notion of the “current user” for accessing the appropriate user tree.

To access a node in the tree, start with the user or system root:

```
Preferences root = Preferences.userRoot();
```

or

```
Preferences root = Preferences.systemRoot();
```

Then access the node. You can simply provide a node path name:

```
Preferences node = root.node("/com/mycompany/myapp");
```

A convenient shortcut gets a node whose path name equals the package name of a class. Simply take an object of that class and call

```
Preferences node = Preferences.userNodeForPackage(obj.getClass());
```

or

```
Preferences node = Preferences.systemNodeForPackage(obj.getClass());
```

Typically, `obj` will be the `this` reference.

Once you have a node, you can access the key/value table with methods

```
String get(String key, String defval)
int getInt(String key, int defval)
long getLong(String key, long defval)
float getFloat(String key, float defval)
double getDouble(String key, double defval)
boolean getBoolean(String key, boolean defval)
byte[] getByteArray(String key, byte[] defval)
```

Note that you must specify a default value when reading the information, in case the repository data is not available. Defaults are required for several reasons. The data might be missing because the user never specified a preference. Certain resource-constrained platforms might not have a repository, and mobile devices might be temporarily disconnected from the repository.

Conversely, you can write data to the repository with `put` methods such as

```
put(String key, String value)
putInt(String key, int value)
```

and so on.

You can enumerate all keys stored in a node with the method

```
String[] keys
```

But there is currently no way to find out the type of the value of a particular key.

Central repositories such as the Windows registry traditionally suffer from two problems:

- They turn into a “dumping ground,” filled with obsolete information.
- Configuration data gets entangled into the repository, making it difficult to move preferences to a new platform.

The Preferences class has a solution for the second problem. You can export the preferences of a subtree (or, less commonly, a single node) by calling the methods

```
void exportSubtree(OutputStream out)
void exportNode(OutputStream out)
```

The data are saved in XML format. You can import them into another repository by calling

```
void importPreferences(InputStream in)
```

Here is a sample file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE preferences SYSTEM "http://java.sun.com/dtd/preferences.dtd">
<preferences EXTERNAL_XML_VERSION="1.0">
  <root type="user">
    <map/>
    <node name="com">
      <map/>
      <node name="horstmann">
        <map/>
        <node name="corejava">
          <map>
            <entry key="left" value="11"/>
            <entry key="top" value="9"/>
            <entry key="width" value="453"/>
            <entry key="height" value="365"/>
            <entry key="title" value="Hello, World!"/>
          </map>
        </node>
      </node>
    </node>
  </root>
</preferences>
```

If your program uses preferences, you should give your users the opportunity of exporting and importing them, so they can easily migrate their settings from one computer to another. The program in Listing 10–8 demonstrates this technique. The program simply saves the position, size, and title of the main window. Try resizing the window, then exit and restart the application. The window will be just like you left it when you exited.

#### Listing 10–8 PreferencesTest.java

```
1. import java.awt.EventQueue;
2. import java.awt.event.*;
3. import java.io.*;
4. import java.util.prefs.*;
5. import javax.swing.*;
6.
7. /**
8.  * A program to test preference settings. The program remembers the frame position, size,
9.  * and title.
```

**Listing 10-8** PreferencesTest.java (continued)

```
10. * @version 1.02 2007-06-12
11. * @author Cay Horstmann
12. */
13. public class PreferencesTest
14. {
15.     public static void main(String[] args)
16.     {
17.         EventQueue.invokeLater(new Runnable()
18.         {
19.             public void run()
20.             {
21.                 PreferencesFrame frame = new PreferencesFrame();
22.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23.                 frame.setVisible(true);
24.             }
25.         });
26.     }
27. }
28.
29. /**
30.  * A frame that restores position and size from user preferences and updates the preferences
31.  * upon exit.
32.  */
33. class PreferencesFrame extends JFrame
34. {
35.     public PreferencesFrame()
36.     {
37.         // get position, size, title from preferences
38.
39.         Preferences root = Preferences.userRoot();
40.         final Preferences node = root.node("/com/horstmann/corejava");
41.         int left = node.getInt("left", 0);
42.         int top = node.getInt("top", 0);
43.         int width = node.getInt("width", DEFAULT_WIDTH);
44.         int height = node.getInt("height", DEFAULT_HEIGHT);
45.         setBounds(left, top, width, height);
46.
47.         // if no title given, ask user
48.
49.         String title = node.get("title", "");
50.         if (title.equals("")) title = JOptionPane.showInputDialog("Please supply a frame title:");
51.         if (title == null) title = "";
52.         setTitle(title);
53.
54.         // set up file chooser that shows XML files
55.
56.         final JFileChooser chooser = new JFileChooser();
57.         chooser.setCurrentDirectory(new File("."));
58.
```

**Listing 10-8** PreferencesTest.java (continued)

```
59. // accept all files ending with .xml
60. chooser.setFileFilter(new javax.swing.filechooser.FileFilter()
61.     {
62.         public boolean accept(File f)
63.         {
64.             return f.getName().toLowerCase().endsWith(".xml") || f.isDirectory();
65.         }
66.
67.         public String getDescription()
68.         {
69.             return "XML files";
70.         }
71.     });
72.
73. // set up menus
74. JMenuBar menuBar = new JMenuBar();
75. setJMenuBar(menuBar);
76. JMenu menu = new JMenu("File");
77. menuBar.add(menu);
78.
79. JMenuItem exportItem = new JMenuItem("Export preferences");
80. menu.add(exportItem);
81. exportItem.addActionListener(new ActionListener()
82.     {
83.         public void actionPerformed(ActionEvent event)
84.         {
85.             if (chooser.showSaveDialog(PreferencesFrame.this) == JFileChooser.APPROVE_OPTION)
86.             {
87.                 try
88.                 {
89.                     OutputStream out = new FileOutputStream(chooser.getSelectedFile());
90.                     node.exportSubtree(out);
91.                     out.close();
92.                 }
93.                 catch (Exception e)
94.                 {
95.                     e.printStackTrace();
96.                 }
97.             }
98.         }
99.     });
100.
101. JMenuItem importItem = new JMenuItem("Import preferences");
102. menu.add(importItem);
103. importItem.addActionListener(new ActionListener()
104.     {
105.         public void actionPerformed(ActionEvent event)
106.         {
```

**Listing 10–8** PreferencesTest.java (continued)

```

107.         if (chooser.showOpenDialog(PreferencesFrame.this) == JFileChooser.APPROVE_OPTION)
108.         {
109.             try
110.             {
111.                 InputStream in = new FileInputStream(chooser.getSelectedFile());
112.                 Preferences.importPreferences(in);
113.                 in.close();
114.             }
115.             catch (Exception e)
116.             {
117.                 e.printStackTrace();
118.             }
119.         }
120.     }
121. });
122.
123. JMenuItem exitItem = new JMenuItem("Exit");
124. menu.add(exitItem);
125. exitItem.addActionListener(new ActionListener()
126. {
127.     public void actionPerformed(ActionEvent event)
128.     {
129.         node.putInt("left", getX());
130.         node.putInt("top", getY());
131.         node.putInt("width", getWidth());
132.         node.putInt("height", getHeight());
133.         node.put("title", getTitle());
134.         System.exit(0);
135.     }
136. });
137. }
138.
139. public static final int DEFAULT_WIDTH = 300;
140. public static final int DEFAULT_HEIGHT = 200;
141. }

```

**API** java.util.prefs.Preferences 1.4

- Preferences userRoot()
 

returns the root preferences node of the user of the calling program.
- Preferences systemRoot()
 

returns the systemwide root preferences node.
- Preferences node(String path)
 

returns a node that can be reached from the current node by the given path. If path is absolute (that is, starts with a /), then the node is located starting from the root of the tree containing this preference node. If there isn't a node with the given path, it is created.

- Preferences `userNodeForPackage(Class c1)`
- Preferences `systemNodeForPackage(Class c1)`  
returns a node in the current user's tree or the system tree whose absolute node path corresponds to the package name of the class `c1`.
- `String[] keys()`  
returns all keys belonging to this node.
- `String get(String key, String defval)`
- `int getInt(String key, int defval)`
- `long getLong(String key, long defval)`
- `float getFloat(String key, float defval)`
- `double getDouble(String key, double defval)`
- `boolean getBoolean(String key, boolean defval)`
- `byte[] getByteArray(String key, byte[] defval)`  
returns the value associated with the given key, or the supplied default value if no value is associated with the key, or the associated value is not of the correct type, or the preferences store is unavailable.
- `void put(String key, String value)`
- `void putInt(String key, int value)`
- `void putLong(String key, long value)`
- `void putFloat(String key, float value)`
- `void putDouble(String key, double value)`
- `void putBoolean(String key, boolean value)`
- `void putByteArray(String key, byte[] value)`  
stores a key/value pair with this node.
- `void exportSubtree(OutputStream out)`  
writes the preferences of this node and its children to the specified stream.
- `void exportNode(OutputStream out)`  
writes the preferences of this node (but not its children) to the specified stream.
- `void importPreferences(InputStream in)`  
imports the preferences contained in the specified stream.

This concludes our discussion of Java software deployment. In the next chapter, you learn how to use exceptions to tell your programs what to do when problems arise at runtime. We also give you tips and techniques for testing and debugging so that not too many things will go wrong when your programs run.

# *Chapter*

# 11

## EXCEPTIONS, LOGGING, ASSERTIONS, AND DEBUGGING

- ▼ DEALING WITH ERRORS
- ▼ CATCHING EXCEPTIONS
- ▼ TIPS FOR USING EXCEPTIONS
- ▼ USING ASSERTIONS
- ▼ LOGGING
- ▼ DEBUGGING TIPS
- ▼ USING A DEBUGGER

In a perfect world, users would never enter data in the wrong form, files they choose to open would always exist, and code would never have bugs. So far, we have mostly presented code as though we lived in this kind of perfect world. It is now time to turn to the mechanisms the Java programming language has for dealing with the real world of bad data and buggy code.

Encountering errors is unpleasant. If a user loses all the work he or she did during a program session because of a programming mistake or some external circumstance, that user may forever turn away from your program. At the very least, you must

- Notify the user of an error;
- Save all work; and
- Allow users to gracefully exit the program.

For exceptional situations, such as bad input data with the potential to bomb the program, Java uses a form of error trapping called, naturally enough, *exception handling*. Exception handling in Java is similar to that in C++ or Delphi. The first part of this chapter covers Java's exceptions.

During testing, you want to run lots of checks to make sure your program does the right thing. But those checks can be time-consuming and unnecessary after testing has completed. You could just remove the checks and stick them back in when additional testing is required, but that is tedious. The second part of this chapter shows you how to use the assertion facility for selectively activating checks.

When your program does the wrong thing, you can't always communicate with the user or terminate. Instead, you may want to record the problem for later analysis. The third part of this chapter discusses the logging facility of Java SE.

Finally, we give you some tips on how to get useful information out of a running Java application, and how to use the debugger in an IDE.

### Dealing with Errors

Suppose an error occurs while a Java program is running. The error might be caused by a file containing wrong information, a flaky network connection, or (we hate to mention it) use of an invalid array index or an attempt to use an object reference that hasn't yet been assigned to an object. Users expect that programs will act sensibly when errors happen. If an operation cannot be completed because of an error, the program ought to either

- Return to a safe state and enable the user to execute other commands; or
- Allow the user to save all work and terminate the program gracefully.

This may not be easy to do, because the code that detects (or even causes) the error condition is usually far removed from the code that can roll back the data to a safe state or the code that can save the user's work and exit cheerfully. The mission of exception handling is to transfer control from where the error occurred to an error handler that can deal with the situation. To handle exceptional situations in your program, you must take into account the errors and problems that may occur. What sorts of problems do you need to consider?

- *User input errors.* In addition to the inevitable typos, some users like to blaze their own trail instead of following directions. Suppose, for example, that a user asks to connect to a URL that is syntactically wrong. Your code should check the syntax, but suppose it does not. Then the network layer will complain.



- *Device errors.* Hardware does not always do what you want it to. The printer may be turned off. A web page may be temporarily unavailable. Devices will often fail in the middle of a task. For example, a printer may run out of paper during printing.
- *Physical limitations.* Disks can fill up; you can run out of available memory.
- *Code errors.* A method may not perform correctly. For example, it could deliver wrong answers or use other methods incorrectly. Computing an invalid array index, trying to find a nonexistent entry in a hash table, and trying to pop an empty stack are all examples of a code error.

The traditional reaction to an error in a method is to return a special error code that the calling method analyzes. For example, methods that read information back from files often return a `-1` end-of-file value marker rather than a standard character. This can be an efficient method for dealing with many exceptional conditions. Another common return value to denote an error condition is the `null` reference. In Chapter 10, you saw an example of this with the `getParameter` method of the `Applet` class that returns `null` if the queried parameter is not present.

Unfortunately, it is not always possible to return an error code. There may be no obvious way of distinguishing valid and invalid data. A method returning an integer cannot simply return `-1` to denote the error—the value `-1` might be a perfectly valid result.

Instead, as we mentioned back in Chapter 5, Java allows every method an alternative exit path if it is unable to complete its task in the normal way. In this situation, the method does not return a value. Instead, it *throws* an object that encapsulates the error information. Note that the method exits immediately; it does not return its normal (or any) value. Moreover, execution does not resume at the code that called the method; instead, the exception-handling mechanism begins its search for an *exception handler* that can deal with this particular error condition.

Exceptions have their own syntax and are part of a special inheritance hierarchy. We take up the syntax first and then give a few hints on how to use this language feature effectively.

### **The Classification of Exceptions**

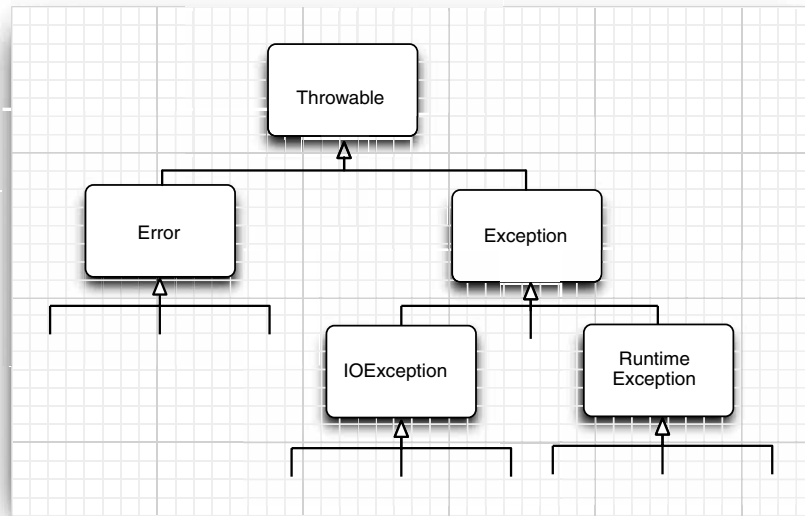
In the Java programming language, an exception object is always an instance of a class derived from `Throwable`. As you will soon see, you can create your own exception classes if the ones built into Java do not suit your needs.

Figure 11-1 is a simplified diagram of the exception hierarchy in Java.

Notice that all exceptions descend from `Throwable`, but the hierarchy immediately splits into two branches: `Error` and `Exception`.

The `Error` hierarchy describes internal errors and resource exhaustion inside the Java runtime system. You should not throw an object of this type. There is little you can do if such an internal error occurs, beyond notifying the user and trying to terminate the program gracefully. These situations are quite rare.

When doing Java programming, you focus on the `Exception` hierarchy. The `Exception` hierarchy also splits into two branches: exceptions that derive from `RuntimeException` and those that do not. The general rule is this: A `RuntimeException` happens because you made a programming error. Any other exception occurs because a bad thing, such as an I/O error, happened to your otherwise good program.



**Figure 11-1 Exception hierarchy in Java**

Exceptions that inherit from `RuntimeException` include such problems as

- A bad cast
- An out-of-bounds array access
- A null pointer access

Exceptions that do not inherit from `RuntimeException` include

- Trying to read past the end of a file
- Trying to open a malformed URL
- Trying to find a `Class` object for a string that does not denote an existing class

The rule “If it is a `RuntimeException`, it was your fault” works pretty well. You could have avoided that `ArrayIndexOutOfBoundsException` by testing the array index against the array bounds. The `NullPointerException` would not have happened had you checked whether the variable was `null` before using it.

How about a malformed URL? Isn’t it also possible to find out whether it is “malformed” before using it? Well, different browsers can handle different kinds of URLs. For example, Netscape can deal with a `mailto:` URL, whereas the applet viewer cannot. Thus, the notion of “malformed” depends on the environment, not just on your code.

The Java Language Specification calls any exception that derives from the class `Error` or the class `RuntimeException` an *unchecked* exception. All other exceptions are called *checked* exceptions. This is useful terminology that we also adopt. The compiler checks that you provide exception handlers for all checked exceptions.



NOTE: The name `RuntimeException` is somewhat confusing. Of course, all of the errors we are discussing occur at runtime.



C++ NOTE: If you are familiar with the (much more limited) exception hierarchy of the standard C++ library, you will be really confused at this point. C++ has two fundamental exception classes, `runtime_error` and `logic_error`. The `logic_error` class is the equivalent of Java's `RuntimeException` and also denotes logical errors in the program. The `runtime_error` class is the base class for exceptions caused by unpredictable problems. It is equivalent to exceptions in Java that are not of type `RuntimeException`.

### Declaring Checked Exceptions

A Java method can throw an exception if it encounters a situation it cannot handle. The idea is simple: a method will not only tell the Java compiler what values it can return, *it is also going to tell the compiler what can go wrong*. For example, code that attempts to read from a file knows that the file might not exist or that it might be empty. The code that tries to process the information in a file therefore will need to notify the compiler that it can throw some sort of `IOException`.

The place in which you advertise that your method can throw an exception is the header of the method; the header changes to reflect the checked exceptions the method can throw. For example, here is the declaration of one of the constructors of the `FileInputStream` class from the standard library. (See Chapter 12 for more on streams.)

```
public FileInputStream(String name) throws FileNotFoundException
```

The declaration says that this constructor produces a `FileInputStream` object from a `String` parameter but that it *also* can go wrong in a special way—by throwing a `FileNotFoundException`. If this sad state should come to pass, the constructor call will not initialize a new `FileInputStream` object but instead will throw an object of the `FileNotFoundException` class. If it does, then the runtime system will begin to search for an exception handler that knows how to deal with `FileNotFoundException` objects.

When you write your own methods, you don't have to advertise every possible throwable object that your method might actually throw. To understand when (and what) you have to advertise in the `throws` clause of the methods you write, keep in mind that an exception is thrown in any of the following four situations:

- You call a method that throws a checked exception, for example, the `FileInputStream` constructor.
- You detect an error and throw a checked exception with the `throw` statement (we cover the `throw` statement in the next section).
- You make a programming error, such as `a[-1] = 0` that gives rise to an unchecked exception such as an `ArrayIndexOutOfBoundsException`.
- An internal error occurs in the virtual machine or runtime library.

If either of the first two scenarios occurs, you must tell the programmers who will use your method about the possibility of an exception. Why? Any method that throws an

exception is a potential death trap. If no handler catches the exception, the current thread of execution terminates.

As with Java methods that are part of the supplied classes, you declare that your method may throw an exception with an *exception specification* in the method header.

```
class MyAnimation
{
    . . .
    public Image loadImage(String s) throws IOException
    {
        . . .
    }
}
```

If a method might throw more than one checked exception type, you must list all exception classes in the header. Separate them by a comma as in the following example:

```
class MyAnimation
{
    . . .
    public Image loadImage(String s) throws EOFException, MalformedURLException
    {
        . . .
    }
}
```

However, you do not need to advertise internal Java errors, that is, exceptions inheriting from `Error`. Any code could potentially throw those exceptions, and they are entirely beyond your control.

Similarly, you should not advertise unchecked exceptions inheriting from `RuntimeException`.

```
class MyAnimation
{
    . . .
    void drawImage(int i) throws ArrayIndexOutOfBoundsException // bad style
    {
        . . .
    }
}
```

These runtime errors are completely under your control. If you are so concerned about array index errors, you should spend the time needed to fix them instead of advertising the possibility that they can happen.

In summary, a method must declare all the *checked* exceptions that it might throw.

Unchecked exceptions are either beyond your control (`Error`) or result from conditions that you should not have allowed in the first place (`RuntimeException`). If your method fails to faithfully declare all checked exceptions, the compiler will issue an error message.

Of course, as you have already seen in quite a few examples, instead of declaring the exception, you can also catch it. Then the exception won't be thrown out of the method, and no `throws` specification is necessary. You see later in this chapter how to decide whether to catch an exception or to enable someone else to catch it.

**X** CAUTION: If you override a method from a superclass, the checked exceptions that the subclass method declares cannot be more general than those of the superclass method. (It is ok to throw more specific exceptions, or not to throw any exceptions in the subclass method.) In particular, if the superclass method throws no checked exception at all, neither can the subclass. For example, if you override `JComponent.paintComponent`, your `paintComponent` method must not throw any checked exceptions, because the superclass method doesn't throw any.

When a method in a class declares that it throws an exception that is an instance of a particular class, then it may throw an exception of that class or of any of its subclasses. For example, the `FileInputStream` constructor could have declared that it throws an `IOException`. In that case, you would not have known what kind of `IOException`. It could be a plain `IOException` or an object of one of the various subclasses, such as `FileNotFoundException`.

**C++** C++ NOTE: The `throws` specifier is the same as the `throw` specifier in C++, with one important difference. In C++, `throws` specifiers are enforced at runtime, not at compile time. That is, the C++ compiler pays no attention to exception specifications. But if an exception is thrown in a function that is not part of the `throws` list, then the unexpected function is called, and, by default, the program terminates.

Also, in C++, a function may throw any exception if no `throws` specification is given. In Java, a method without a `throws` specifier may not throw any checked exception at all.

### How to Throw an Exception

Let us suppose something terrible has happened in your code. You have a method, `readData`, that is reading in a file whose header promised

```
Content-length: 1024
```

but you get an end of file after 733 characters. You decide this situation is so abnormal that you want to throw an exception.

You need to decide what exception type to throw. Some kind of `IOException` would be a good choice. Perusing the Java API documentation, you find an `EOFException` with the description "Signals that an EOF has been reached unexpectedly during input." Perfect. Here is how you throw it:

```
throw new EOFException();
```

or, if you prefer,

```
EOFException e = new EOFException();
throw e;
```

Here is how it all fits together:

```
String readData(Scanner in) throws EOFException
{
    . . .
    while (. . .)
    {
        if (!in.hasNext()) // EOF encountered
        {
```

```

        if (n < len)
            throw new EOFException();
    }
    . . .
}
return s;
}

```

The `EOFException` has a second constructor that takes a string argument. You can put this to good use by describing the exceptional condition more carefully.

```

String gripe = "Content-length: " + len + ", Received: " + n;
throw new EOFException(gripe);

```

As you can see, throwing an exception is easy if one of the existing exception classes works for you. In this case:

1. Find an appropriate exception class.
2. Make an object of that class.
3. Throw it.

Once a method throws an exception, the method does not return to its caller. This means that you do not have to worry about cooking up a default return value or an error code.



**C++ NOTE:** Throwing an exception is the same in C++ and in Java, with one small exception. In Java, you can throw only objects of subclasses of `Throwable`. In C++, you can throw values of any type.

### Creating Exception Classes

Your code may run into a problem that is not adequately described by any of the standard exception classes. In this case, it is easy enough to create your own exception class. Just derive it from `Exception` or from a child class of `Exception` such as `IOException`. It is customary to give both a default constructor and a constructor that contains a detailed message. (The `toString` method of the `Throwable` superclass prints that detailed message, which is handy for debugging.)

```

class FileFormatException extends IOException
{
    public FileFormatException() {}
    public FileFormatException(String gripe)
    {
        super(gripe);
    }
}

```

Now you are ready to throw your very own exception type.

```

String readData(BufferedReader in) throws FileFormatException
{
    . . .
    while (. . .)
    {
        if (ch == -1) // EOF encountered

```

```

    {
        if (n < len)
            throw new FileFormatException();
        }
        . . .
    }
    return s;
}

```

**API** `java.lang.Throwable 1.0`

- `Throwable()`  
constructs a new `Throwable` object with no detailed message.
- `Throwable(String message)`  
constructs a new `Throwable` object with the specified detailed message. By convention, all derived exception classes support both a default constructor and a constructor with a detailed message.
- `String getMessage()`  
gets the detailed message of the `Throwable` object.

**Catching Exceptions**

You now know how to throw an exception. It is pretty easy. You throw it and you forget it. Of course, some code has to catch the exception. Catching exceptions requires more planning.

If an exception occurs that is not caught anywhere, the program will terminate and print a message to the console, giving the type of the exception and a stack trace. Graphics programs (both applets and applications) catch exceptions, print stack trace messages, and then go back to the user interface processing loop. (When you are debugging a graphically based program, it is a good idea to keep the console available on the screen and not minimized.)

To catch an exception, you set up a `try/catch` block. The simplest form of the `try` block is as follows:

```

try
{
    code
    more code
    more code
}
catch (ExceptionType e)
{
    handler for this type
}

```

If any of the code inside the `try` block throws an exception of the class specified in the `catch` clause, then

1. The program skips the remainder of the code in the `try` block.
2. The program executes the handler code inside the `catch` clause.

If none of the code inside the try block throws an exception, then the program skips the catch clause.

If any of the code in a method throws an exception of a type other than the one named in the catch clause, this method exits immediately. (Hopefully, one of its callers has already coded a catch clause for that type.)

To show this at work, we show some fairly typical code for reading in data:

```
public void read(String filename)
{
    try
    {
        InputStream in = new FileInputStream(filename);
        int b;
        while ((b = in.read()) != -1)
        {
            process input
        }
    }
    catch (IOException exception)
    {
        exception.printStackTrace();
    }
}
```

Notice that most of the code in the try clause is straightforward: it reads and processes bytes until we encounter the end of the file. As you can see by looking at the Java API, there is the possibility that the `read` method will throw an `IOException`. In that case, we skip out of the entire `while` loop, enter the catch clause and generate a stack trace. For a toy program, that seems like a reasonable way to deal with this exception. What other choice do you have?

Often, the best choice is to do nothing at all and simply pass the exception on to the caller. If an error occurs in the `read` method, let the caller of the `read` method worry about it! If we take that approach, then we have to advertise the fact that the method may throw an `IOException`.

```
public void read(String filename) throws IOException
{
    InputStream in = new FileInputStream(filename);
    int b;
    while ((b = in.read()) != -1)
    {
        process input
    }
}
```

Remember, the compiler strictly enforces the `throws` specifiers. If you call a method that throws a checked exception, you must either handle it or pass it on.

Which of the two is better? As a general rule, you should catch those exceptions that you know how to handle and propagate those that you do not know how to handle.



When you propagate an exception, you must add a `throws` specifier to alert the caller that an exception may be thrown.

Look at the Java API documentation to see what methods throw which exceptions. Then decide whether you should handle them or add them to the `throws` list. There is nothing embarrassing about the latter choice. It is better to direct an exception to a competent handler than to squelch it.

Please keep in mind that there is one exception to this rule, as we mentioned earlier. If you are writing a method that overrides a superclass method that throws no exceptions (such as `paintComponent` in `JComponent`), then you *must* catch each checked exception in the method's code. You are not allowed to add more `throws` specifiers to a subclass method than are present in the superclass method.



**C++ NOTE:** Catching exceptions is almost the same in Java and in C++. Strictly speaking, the analog of

```
catch (Exception e) // Java
```

is

```
catch (Exception& e) // C++
```

There is no analog to the C++ `catch (...)`. This is not needed in Java because all exceptions derive from a common superclass.

### Catching Multiple Exceptions

You can catch multiple exception types in a try block and handle each type differently. You use a separate `catch` clause for each type as in the following example:

```
try
{
    code that might throw exceptions
}
catch (MalformedURLException e1)
{
    emergency action for malformed URLs
}
catch (UnknownHostException e2)
{
    emergency action for unknown hosts
}
catch (IOException e3)
{
    emergency action for all other I/O problems
}
```

The exception object (`e1`, `e2`, `e3`) may contain information about the nature of the exception. To find out more about the object, try

```
e3.getMessage()
```

to get the detailed error message (if there is one), or

```
e3.getClass().getName()
```

to get the actual type of the exception object.

**Rethrowing and Chaining Exceptions**

You can throw an exception in a catch clause. Typically, you do this because you want to change the exception type. If you build a subsystem that other programmers use, it makes a lot of sense to use an exception type that indicates a failure of the subsystem. An example of such an exception type is the `ServletException`. The code that executes a servlet may not want to know in minute detail what went wrong, but it definitely wants to know that the servlet was at fault.

Here is how you can catch an exception and rethrow it:

```
try
{
    access the database
}
catch (SQLException e)
{
    throw new ServletException("database error: " + e.getMessage());
}
```

Here, the `ServletException` is constructed with the message text of the exception. As of Java SE 1.4, you can do better than that and set the original exception as the “cause” of the new exception:

```
try
{
    access the database
}
catch (SQLException e)
{
    Throwable se = new ServletException("database error");
    se.initCause(e);
    throw se;
}
```

When the exception is caught, the original exception can be retrieved:

```
Throwable e = se.getCause();
```

This wrapping technique is highly recommended. It allows you to throw high-level exceptions in subsystems without losing the details of the original failure.



**TIP:** The wrapping technique is also useful if a checked exception occurs in a method that is not allowed to throw a checked exception. You can catch the checked exception and wrap it into a runtime exception.

---



**NOTE:** A number of exception classes, such as `ClassNotFoundException`, `InvocationTargetException`, and `RuntimeException`, have had their own chaining schemes. As of Java SE 1.4, these have been brought into conformance with the “cause” mechanism. You can still retrieve the chained exception in the historical way or just call `getCause`.

---

### The finally Clause

When your code throws an exception, it stops processing the remaining code in your method and exits the method. This is a problem if the method has acquired some local resource that only it knows about and if that resource must be cleaned up. One solution is to catch and rethrow all exceptions. But this solution is tedious because you need to clean up the resource allocation in two places, in the normal code and in the exception code.

Java has a better solution, the `finally` clause. Here we show you how to properly dispose of a `Graphics` object. If you do any database programming in Java, you will need to use the same techniques to close connections to the database. As you will see in Chapter 4 of Volume II, it is very important to close all database connections properly, even when exceptions occur.

The code in the `finally` clause executes whether or not an exception was caught. In the following example, the program will dispose of the graphics context *under all circumstances*:

```
Graphics g = image.getGraphics();
try
{
    // 1
    code that might throw exceptions
    // 2
}
catch (IOException e)
{
    // 3
    show error dialog
    // 4
}
finally
{
    // 5
    g.dispose();
}
// 6
```

Let us look at the three possible situations in which the program will execute the `finally` clause.

1. The code throws no exceptions. In this event, the program first executes all the code in the `try` block. Then, it executes the code in the `finally` clause. Afterwards, execution continues with the first statement after the `finally` clause. In other words, execution passes through points 1, 2, 5, and 6.
2. The code throws an exception that is caught in a `catch` clause, in our case, an `IOException`. For this, the program executes all code in the `try` block, up to the point at which the exception was thrown. The remaining code in the `try` block is skipped. The program then executes the code in the matching `catch` clause, then the code in the `finally` clause.

If the `catch` clause does not throw an exception, the program executes the first line after the `finally` clause. In this scenario, execution passes through points 1, 3, 4, 5, and 6.

If the catch clause throws an exception, then the exception is thrown back to the caller of this method, and execution passes through points 1, 3, and 5 only.

3. The code throws an exception that is not caught in any catch clause. For this, the program executes all code in the try block until the exception is thrown. The remaining code in the try block is skipped. Then, the code in the finally clause is executed, and the exception is thrown back to the caller of this method. Execution passes through points 1 and 5 only.

You can use the finally clause without a catch clause. For example, consider the following try statement:

```
InputStream in = ...;
try
{
    code that might throw exceptions
}
finally
{
    in.close();
}
```

The `in.close()` statement in the finally clause is executed whether or not an exception is encountered in the try block. Of course, if an exception is encountered, it is rethrown and must be caught in another catch clause.

In fact, as explained in the following tip, we think it is a very good idea to use the finally clause in this way whenever you need to close a resource.



**TIP:** We strongly suggest that you *decouple* try/catch and try/finally blocks. This makes your code far less confusing. For example:

```
InputStream in = ...;
try
{
    try
    {
        code that might throw exceptions
    }
    finally
    {
        in.close();
    }
}
catch (IOException e)
{
    show error dialog
}
```

The inner try block has a single responsibility: to make sure that the input stream is closed. The outer try block has a single responsibility: to ensure that errors are reported. Not only is this solution clearer, it is also more functional: errors in the finally clause are reported.

---

**X** CAUTION: A `finally` clause can yield unexpected results when it contains return statements. Suppose you exit the middle of a try block with a return statement. Before the method returns, the contents of the `finally` block are executed. If the `finally` block also contains a return statement, then it masks the original return value. Consider this contrived example:

```
public static int f(int n)
{
    try
    {
        int r = n * n;
        return r;
    }
    finally
    {
        if (n == 2) return 0;
    }
}
```

If you call `f(2)`, then the try block computes `r = 4` and executes the return statement. However, the `finally` clause is executed before the method actually returns. The `finally` clause causes the method to return 0, ignoring the original return value of 4.

Sometimes the `finally` clause gives you grief, namely if the cleanup method can also throw an exception. A typical case is closing a stream. (See Chapter 1 of Volume II for more information on streams.) Suppose you want to make sure that you close a stream when an exception hits in the stream processing code.

```
InputStream in = ...;
try
{
    code that might throw exceptions
}
finally
{
    in.close();
}
```

Now suppose that the code in the try block throws some exception *other than an* `IOException` that is of interest to the caller of the code. The `finally` block executes, and the `close` method is called. That method can itself throw an `IOException`! When it does, then the original exception is lost and the `IOException` is thrown instead. That is very much against the spirit of exception handling.

It is always a good idea—unfortunately not one that the designers of the `InputStream` class chose to follow—to throw no exceptions in cleanup operations such as `dispose`, `close`, and so on, that you expect users to call in `finally` blocks.

**C++** C++ NOTE: There is one fundamental difference between C++ and Java with regard to exception handling. Java has no destructors; thus, there is no stack unwinding as in C++. This means that the Java programmer must manually place code to reclaim resources in `finally` blocks. Of course, because Java does garbage collection, there are far fewer resources that require manual deallocation.

### Analyzing Stack Trace Elements

A *stack trace* is a listing of all pending method calls at a particular point in the execution of a program. You have almost certainly seen stack trace listings—they are displayed whenever a Java program terminates with an uncaught exception.

Before Java SE 1.4, you could access the text description of a stack trace by calling the `printStackTrace` method of the `Throwable` class. Now you can call the `getStackTrace` method to get an array of `StackTraceElement` objects that you can analyze in your program. For example:

```
Throwable t = new Throwable();
StackTraceElement[] frames = t.getStackTrace();
for (StackTraceElement frame : frames)
    analyze frame
```

The `StackTraceElement` class has methods to obtain the file name and line number, as well as the class and method name, of the executing line of code. The `toString` method yields a formatted string containing all of this information.

Java SE 5.0 added the static `Thread.getAllStackTraces` method that yields the stack traces of all threads. Here is how you use that method:

```
Map<Thread, StackTraceElement[]> map = Thread.getAllStackTraces();
for (Thread t : map.keySet())
{
    StackTraceElement[] frames = map.get(t);
    analyze frames
}
```

See Chapters 13 and 14 for more information on the `Map` interface and threads.

Listing 11–1 prints the stack trace of a recursive factorial function. For example, if you compute `factorial(3)`, the printout is

```
factorial(3):
StackTraceTest.factorial(StackTraceTest.java:18)
StackTraceTest.main(StackTraceTest.java:34)
factorial(2):
StackTraceTest.factorial(StackTraceTest.java:18)
StackTraceTest.factorial(StackTraceTest.java:24)
StackTraceTest.main(StackTraceTest.java:34)
factorial(1):
StackTraceTest.factorial(StackTraceTest.java:18)
StackTraceTest.factorial(StackTraceTest.java:24)
StackTraceTest.factorial(StackTraceTest.java:24)
StackTraceTest.main(StackTraceTest.java:34)
return 1
return 2
return 6
```

**Listing 11-1** StackTraceTest.java

```
1. import java.util.*;
2.
3. /**
4.  * A program that displays a trace feature of a recursive method call.
5.  * @version 1.01 2004-05-10
6.  * @author Cay Horstmann
7.  */
8. public class StackTraceTest
9. {
10.     /**
11.      * Computes the factorial of a number
12.      * @param n a nonnegative integer
13.      * @return n! = 1 * 2 * . . . * n
14.      */
15.     public static int factorial(int n)
16.     {
17.         System.out.println("factorial(" + n + "):");
18.         Throwable t = new Throwable();
19.         StackTraceElement[] frames = t.getStackTrace();
20.         for (StackTraceElement f : frames)
21.             System.out.println(f);
22.         int r;
23.         if (n <= 1) r = 1;
24.         else r = n * factorial(n - 1);
25.         System.out.println("return " + r);
26.         return r;
27.     }
28.
29.     public static void main(String[] args)
30.     {
31.         Scanner in = new Scanner(System.in);
32.         System.out.print("Enter n: ");
33.         int n = in.nextInt();
34.         factorial(n);
35.     }
36. }
```

**API** java.lang.Throwable 1.0

- Throwable(Throwable cause) **1.4**
- Throwable(String message, Throwable cause) **1.4**  
constructs a Throwable with a given cause.
- Throwable initCause(Throwable cause) **1.4**  
sets the cause for this object or throws an exception if this object already has a cause. Returns this.

- `Throwable getCause()` **1.4**  
gets the exception object that was set as the cause for this object, or `null` if no cause was set.
- `StackTraceElement[] getStackTrace()` **1.4**  
gets the trace of the call stack at the time this object was constructed.

**API** `java.lang.Exception` **1.0**

- `Exception(Throwable cause)` **1.4**
- `Exception(String message, Throwable cause)`  
constructs an `Exception` with a given cause.

**API** `java.lang.RuntimeException` **1.0**

- `RuntimeException(Throwable cause)` **1.4**
- `RuntimeException(String message, Throwable cause)` **1.4**  
constructs a `RuntimeException` with a given cause.

**API** `java.lang.StackTraceElement` **1.4**

- `String getFileName()`  
gets the name of the source file containing the execution point of this element, or `null` if the information is not available.
- `int getLineNumber()`  
gets the line number of the source file containing the execution point of this element, or `-1` if the information is not available.
- `String getClassName()`  
gets the fully qualified name of the class containing the execution point of this element.
- `String getMethodName()`  
gets the name of the method containing the execution point of this element. The name of a constructor is `<init>`. The name of a static initializer is `<clinit>`. You can't distinguish between overloaded methods with the same name.
- `boolean isNativeMethod()`  
returns `true` if the execution point of this element is inside a native method.
- `String toString()`  
returns a formatted string containing the class and method name and the file name and line number, if available.

**Tips for Using Exceptions**

There is a certain amount of controversy about the proper use of exceptions. Some programmers believe that all checked exceptions are a nuisance, others can't seem to throw enough of them. We think that exceptions (even checked exceptions) have their place, and offer you these tips for their proper use.



1. *Exception handling is not supposed to replace a simple test.*

As an example of this, we wrote some code that tries 10,000,000 times to pop an empty stack. It first does this by finding out whether the stack is empty.

```
if (!s.empty()) s.pop();
```

Next, we tell it to pop the stack no matter what. Then, we catch the `EmptyStackException` that tells us that we should not have done that.

```
try()
{
    s.pop();
}
catch (EmptyStackException e)
{
}
```

On our test machine, we got the timing data in Table 11–1.

**Table 11–1 Timing Data**

Test	Throw/Catch
646 milliseconds	21,739 milliseconds

As you can see, it took far longer to catch an exception than it did to perform a simple test. The moral is: Use exceptions for exceptional circumstances only.

2. *Do not micromanage exceptions.*

Many programmers wrap every statement in a separate try block.

```
OutputStream out;
Stack s;

for (i = 0; i < 100; i++)
{
    try
    {
        n = s.pop();
    }
    catch (EmptyStackException s)
    {
        // stack was empty
    }
    try
    {
        out.writeInt(n);
    }
    catch (IOException e)
    {
        // problem writing to file
    }
}
```

This approach blows up your code dramatically. Think about the task that you want the code to accomplish. Here we want to pop 100 numbers off a stack and save them to a file. (Never mind why—it is just a toy example.) There is nothing we can do if a problem rears its ugly head. If the stack is empty, it will not become occupied. If the file contains an error, the error will not magically go away. It therefore makes sense to wrap the *entire task* in a try block. If any one operation fails, you can then abandon the task.

```
try
{
    for (i = 0; i < 100; i++)
    {
        n = s.pop();
        out.writeInt(n);
    }
}
catch (IOException e)
{
    // problem writing to file
}
catch (EmptyStackException s)
{
    // stack was empty
}
```

This code looks much cleaner. It fulfills one of the promises of exception handling, to *separate* normal processing from error handling.

3. *Make good use of the exception hierarchy.*

Don't just throw a `RuntimeException`. Find an appropriate subclass or create your own. Don't just catch `Throwable`. It makes your code hard to read and maintain.

Respect the difference between checked and unchecked exceptions. Checked exceptions are inherently burdensome—don't throw them for logic errors. (For example, the reflection library gets this wrong. Callers often need to catch exceptions that they know can never happen.)

Do not hesitate to turn an exception into another exception that is more appropriate. For example, when you parse an integer in a file, catch the `NumberFormatException` and turn it into a subclass of `IOException` or `MySubsystemException`.

4. *Do not squelch exceptions.*

In Java, there is the tremendous temptation to shut up exceptions. You write a method that calls a method that might throw an exception once a century. The compiler whines because you have not declared the exception in the `throws` list of your method. You do not want to put it in the `throws` list because then the compiler will whine about all the methods that call your method. So you just shut it up:

```
public Image loadImage(String s)
{
    try
    {
        code that threatens to throw checked exceptions
    }
    catch (Exception e)
    {} // so there
}
```

Now your code will compile without a hitch. It will run fine, except when an exception occurs. Then, the exception will be silently ignored. If you believe that exceptions are at all important, you should make some effort to handle them right.

5. *When you detect an error, “tough love” works better than indulgence.*

Some programmers worry about throwing exceptions when they detect errors.

Maybe it would be better to return a dummy value rather than throw an exception when a method is called with invalid parameters. For example, should `Stack.pop` return `null` rather than throw an exception when a stack is empty? We think it is better to throw a `EmptyStackException` at the point of failure than to have a `NullPointerException` occur at later time.

6. *Propagating exceptions is not a sign of shame.*

Many programmers feel compelled to catch all exceptions that are thrown. If they call a method that throws an exception, such as the `FileInputStream` constructor or the `readLine` method, they instinctively catch the exception that may be generated. Often, it is actually better to *propagate* the exception instead of catching it:

```
public void readStuff(String filename) throws IOException // not a sign of shame!
{
    InputStream in = new FileInputStream(filename);
    . . .
}
```

Higher-level methods are often better equipped to inform the user of errors or to abandon unsuccessful commands.



NOTE: Rules 5 and 6 can be summarized as “throw early, catch late.”

## Using Assertions

Assertions are a commonly used idiom for defensive programming. Suppose you are convinced that a particular property is fulfilled, and you rely on that property in your code. For example, you may be computing

```
double y = Math.sqrt(x);
```

You are certain that `x` is not negative. Perhaps it is the result of another computation that can't have a negative result, or it is a parameter of a method that requires its callers to supply only positive inputs. Still, you want to double-check rather than having confusing “not a number” floating-point values creep into your computation. You could, of course, throw an exception:

```
if (x < 0) throw new IllegalArgumentException("x < 0");
```

But this code stays in the program, even after testing is complete. If you have lots of checks of this kind, the program runs quite a bit slower than it should.

The assertion mechanism allows you to put in checks during testing and to have them automatically removed in the production code.

As of Java SE 1.4, the Java language has a keyword `assert`. There are two forms:

```
assert condition;
```

and

```
assert condition : expression;
```

Both statements evaluate the condition and throw an `AssertionError` if it is false. In the second statement, the expression is passed to the constructor of the `AssertionError` object and turned into a message string.



**NOTE:** The sole purpose of the *expression* part is to produce a message string. The `AssertionError` object does not store the actual expression value, so you can't query it later. As the JDK documentation states with paternalistic charm, doing so "would encourage programmers to attempt to recover from assertion failure, which defeats the purpose of the facility."

To assert that `x` is nonnegative, you can simply use the statement

```
assert x >= 0;
```

Or you can pass the actual value of `x` into the `AssertionError` object, so that it gets displayed later.

```
assert x >= 0 : x;
```



**C++ NOTE:** The `assert` macro of the C language turns the assertion condition into a string that is printed if the assertion fails. For example, if `assert(x >= 0)` fails, it prints that "`x >= 0`" is the failing condition. In Java, the condition is not automatically part of the error report. If you want to see it, you have to pass it as a string into the `AssertionError` object: `assert x >= 0 : "x >= 0"`.

### ***Assertion Enabling and Disabling***

By default, assertions are disabled. You enable them by running the program with the `-enableassertions` or `-ea` option:

```
java -enableassertions MyApp
```

Note that you do not have to recompile your program to enable or disable assertions. Enabling or disabling assertions is a function of the *class loader*. When assertions are disabled, the class loader strips out the assertion code so that it won't slow execution.

You can even turn on assertions in specific classes or in entire packages. For example:

```
java -ea:MyClass -ea:com.mycompany.mylib... MyApp
```

This command turns on assertions for the class `MyClass` and all classes in the `com.mycompany.mylib` package *and its subpackages*. The option `-ea...` turns on assertions in all classes of the default package.

You can also disable assertions in certain classes and packages with the `-disableassertions` or `-da` option:

```
java -ea:... -da:MyClass MyApp
```

Some classes are not loaded by a class loader but directly by the virtual machine. You can use these switches to selectively enable or disable assertions in those classes.

However, the `-ea` and `-da` switches that enable or disable all assertions do not apply to the “system classes” without class loaders. Use the `-enableassertions/-esa` switch to enable assertions in system classes.

It is also possible to programmatically control the assertion status of class loaders. See the API notes at the end of this section.

### **Using Assertions for Parameter Checking**

The Java language gives you three mechanisms to deal with system failures:

- Throwing an exception
- Logging
- Using assertions

When should you choose assertions? Keep these points in mind:

- Assertion failures are intended to be fatal, unrecoverable errors.
- Assertion checks are turned on only during development and testing. (This is sometimes jokingly described as “wearing a life jacket when you are close to shore, and throwing it overboard once you are in the middle of the ocean.”)

Therefore, you would not use assertions for signaling recoverable conditions to another part of the program or for communicating problems to the program user. Assertions should only be used to locate internal program errors during testing.

Let’s look at a common scenario—the checking of method parameters. Should you use assertions to check for illegal index values or `null` references? To answer that question, you have to look at the documentation of the method. Suppose you implement a sorting method.

```
/**
 * Sorts the specified range of the specified array into ascending numerical order.
 * The range to be sorted extends from fromIndex, inclusive, to toIndex, exclusive.
 * @param a the array to be sorted.
 * @param fromIndex the index of the first element (inclusive) to be sorted.
 * @param toIndex the index of the last element (exclusive) to be sorted.
 * @throws IllegalArgumentException if fromIndex > toIndex
 * @throws ArrayIndexOutOfBoundsException if fromIndex < 0 or toIndex > a.length
 */
static void sort(int[] a, int fromIndex, int toIndex)
```

The documentation states that the method throws an exception if the index values are incorrect. That behavior is part of the contract that the method makes with its callers. If you implement the method, you have to respect that contract and throw the indicated exceptions. It would not be appropriate to use assertions instead.

Should you assert that `a` is not `null`? That is not appropriate either. The method documentation is silent on the behavior of the method when `a` is `null`. The callers have the right to assume that the method will return successfully in that case and not throw an assertion error.

However, suppose the method contract had been slightly different:

```
@param a the array to be sorted. (Must not be null)
```

Now the callers of the method have been put on notice that it is illegal to call the method with a `null` array. Then the method may start with the assertion

```
assert a != null;
```

Computer scientists call this kind of contract a *precondition*. The original method had no preconditions on its parameters—it promised a well-defined behavior in all cases. The revised method has a single precondition: that `a` is not `null`. If the caller fails to fulfill the precondition, then all bets are off and the method can do anything it wants. In fact, with the assertion in place, the method has a rather unpredictable behavior when it is called illegally. It sometimes throws an assertion error, and sometimes a null pointer exception, depending on how its class loader is configured.

### Using Assertions for Documenting Assumptions

Many programmers use comments to document their underlying assumptions. Consider this example from <http://java.sun.com/javase/6/docs/technotes/guides/language/assert.html>:

```
if (i % 3 == 0)
    . . .
else if (i % 3 == 1)
    . . .
else // (i % 3 == 2)
    . . .
```

In this case, it makes a lot of sense to use an assertion instead.

```
if (i % 3 == 0)
    . . .
else if (i % 3 == 1)
    . . .
else
{
    assert i % 3 == 2;
    . . .
}
```

Of course, it would make even more sense to think through the issue a bit more thoroughly. What are the possible values of `i % 3`? If `i` is positive, the remainders must be 0, 1, or 2. If `i` is negative, then the remainders can be `-1` or `-2`. Thus, the real assumption is that `i` is not negative. A better assertion would be

```
assert i >= 0;
```

before the `if` statement.

At any rate, this example shows a good use of assertions as a self-check for the programmer. As you can see, assertions are a tactical tool for testing and debugging. In contrast, logging is a strategic tool for the entire life cycle of a program. We will examine logging in the next section.

#### API `java.lang.ClassLoader` 1.0

- `void setDefaultAssertionStatus(boolean b)` 1.4  
enables or disables assertions for all classes loaded by this class loader that don't have an explicit class or package assertion status.

- `void setClassAssertionStatus(String className, boolean b)` **1.4**  
enables or disables assertions for the given class and its inner classes.
- `void setPackageAssertionStatus(String packageName, boolean b)` **1.4**  
enables or disables assertions for all classes in the given package and its subpackages.
- `void clearAssertionStatus()` **1.4**  
removes all explicit class and package assertion status settings and disables assertions for all classes loaded by this class loader.

## Logging

Every Java programmer is familiar with the process of inserting calls to `System.out.println` into troublesome code to gain insight into program behavior. Of course, once you have figured out the cause of trouble, you remove the print statements, only to put them back in when the next problem surfaces. The logging API is designed to overcome this problem. Here are the principal advantages of the API:

- It is easy to suppress all log records or just those below a certain level, and just as easy to turn them back on.
- Suppressed logs are very cheap, so that there is only a minimal penalty for leaving the logging code in your application.
- Log records can be directed to different handlers, for display in the console, for storage in a file, and so on.
- Both loggers and handlers can filter records. Filters discard boring log entries, using any criteria supplied by the filter implementor.
- Log records can be formatted in different ways, for example, in plain text or XML.
- Applications can use multiple loggers, with hierarchical names such as `com.mycompany.myapp`, similar to package names.
- By default, the logging configuration is controlled by a configuration file. Applications can replace this mechanism if desired.

### Basic Logging

Let's get started with the simplest possible case. The logging system manages a default logger `Logger.global` that you can use instead of `System.out`. Use the `info` method to log an information message:

```
Logger.global.info("File->Open menu item selected");
```

By default, the record is printed like this:

```
May 10, 2004 10:12:15 PM LoggingImageViewer fileOpen  
INFO: File->Open menu item selected
```

(Note that the time and the names of the calling class and method are automatically included.) But if you call

```
Logger.global.setLevel(Level.OFF);
```

at an appropriate place (such as the beginning of `main`), then all logging is suppressed.

### Advanced Logging

Now that you have seen “logging for dummies,” let’s go on to industrial-strength logging. In a professional application, you wouldn’t want to log all records to a single global logger. Instead, you can define your own loggers.

When you request a logger with a given name for the first time, it is created.

```
Logger myLogger = Logger.getLogger("com.mycompany.myapp");
```

Subsequent calls to the same name yield the same logger object.

Similar to package names, logger names are hierarchical. In fact, they are *more* hierarchical than packages. There is no semantic relationship between a package and its parent, but logger parents and children share certain properties. For example, if you set the log level on the logger "com.mycompany", then the child loggers inherit that level.

There are seven logging levels:

- SEVERE
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST

By default, the top three levels are actually logged. You can set a different level, for example,

```
logger.setLevel(Level.FINE);
```

Now all levels of FINE and higher are logged.

You can also use `Level.ALL` to turn on logging for all levels or `Level.OFF` to turn all logging off.

There are logging methods for all levels, such as

```
logger.warning(message);  
logger.fine(message);
```

and so on. Alternatively, you can use the `log` method and supply the level, such as

```
logger.log(Level.FINE, message);
```



**TIP:** The default logging configuration logs all records with level of INFO or higher. Therefore, you should use the levels CONFIG, FINE, FINER, and FINEST for debugging messages that are useful for diagnostics but meaningless to the program user.

---



**CAUTION:** If you set the logging level to a value finer than INFO, then you also need to change the log handler configuration. The default log handler suppresses messages below INFO. See the next section for details.

---

The default log record shows the name of the class and method that contain the logging call, as inferred from the call stack. However, if the virtual machine optimizes execution,



accurate call information may not be available. You can use the `logp` method to give the precise location of the calling class and method. The method signature is

```
void logp(Level l, String className, String methodName, String message)
```

There are convenience methods for tracing execution flow:

```
void entering(String className, String methodName)
void entering(String className, String methodName, Object param)
void entering(String className, String methodName, Object[] params)
void exiting(String className, String methodName)
void exiting(String className, String methodName, Object result)
```

For example:

```
int read(String file, String pattern)
{
    logger.entering("com.mycompany.mylib.Reader", "read",
        new Object[] { file, pattern });
    . . .
    logger.exiting("com.mycompany.mylib.Reader", "read", count);
    return count;
}
```

These calls generate log records of level `FINER` that start with the strings `ENTRY` and `RETURN`.



**NOTE:** At some point in the future, the logging methods with an `Object[]` parameter will be rewritten to support variable parameter lists (“varargs”). Then, you will be able to make calls such as `logger.entering("com.mycompany.mylib.Reader", "read", file, pattern)`.

A common use for logging is to log unexpected exceptions. Two convenience methods include a description of the exception in the log record.

```
void throwing(String className, String methodName, Throwable t)
void log(Level l, String message, Throwable t)
```

Typical uses are

```
if (. . .)
{
    IOException exception = new IOException(". . .");
    logger.throwing("com.mycompany.mylib.Reader", "read", exception);
    throw exception;
}
```

and

```
try
{
    . . .
}
catch (IOException e)
{
    Logger.getLogger("com.mycompany.myapp").log(Level.WARNING, "Reading image", e);
}
```

The `throwing` call logs a record with level `FINER` and a message that starts with `THROW`.

**Changing the Log Manager Configuration**

You can change various properties of the logging system by editing a configuration file. The default configuration file is located at

```
jre/lib/logging.properties
```

To use another file, set the `java.util.logging.config.file` property to the file location by starting your application with

```
java -Djava.util.logging.config.file=configFile MainClass
```



**CAUTION:** Calling `System.setProperty("java.util.logging.config.file", file)` in `main` has no effect because the log manager is initialized during VM startup, before `main` executes.

To change the default logging level, edit the configuration file and modify the line

```
.level=INFO
```

You can specify the logging levels for your own loggers by adding lines such as

```
com.mycompany.myapp.level=FINE
```

That is, append the `.level` suffix to the logger name.

As you see later in this section, the loggers don't actually send the messages to the console—that is the job of the handlers. Handlers also have levels. To see `FINE` messages on the console, you also need to set

```
java.util.logging.ConsoleHandler.level=FINE
```



**CAUTION:** The settings in the log manager configuration are *not* system properties. Starting a program with `-Dcom.mycompany.myapp.level=FINE` does not have any influence on the logger.



**CAUTION:** At least up to Java SE 6, the API documentation of the `LogManager` class claims that you can set the `java.util.logging.config.class` and `java.util.logging.config.file` properties via the Preferences API. This is false—see [http://bugs.sun.com/bugdatabase/view\\_bug.do?bug\\_id=4691587](http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4691587).



**NOTE:** The logging properties file is processed by the `java.util.logging.LogManager` class. It is possible to specify a different log manager by setting the `java.util.logging.manager` system property to the name of a subclass. Alternatively, you can keep the standard log manager and still bypass the initialization from the logging properties file. Set the `java.util.logging.config.class` system property to the name of a class that sets log manager properties in some other way. See the API documentation for the `LogManager` class for more information.

It is also possible to change logging levels in a running program by using the `jconsole` program. See <http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html#LoggingControl> for information.

### Localization

You may want to localize logging messages so that they are readable for international users. Internationalization of applications is the topic of Chapter 5 of Volume II. Briefly, here are the points to keep in mind when localizing logging messages.

Localized applications contain locale-specific information in *resource bundles*. A resource bundle consists of a set of mappings for various locales (such as United States or Germany). For example, a resource bundle may map the string "readingFile" into strings "Reading file" in English or "Achtung! Datei wird eingelesen" in German.

A program may contain multiple resource bundles, perhaps one for menus and another for log messages. Each resource bundle has a name (such as "com.mycompany.logmessages"). To add mappings to a resource bundle, you supply a file for each locale. English message mappings are in a file `com/mycompany/logmessages_en.properties`, and German message mappings are in a file `com/mycompany/logmessages_de.properties`. (The `en`, `de` codes are the language codes.) You place the files together with the class files of your application, so that the `ResourceBundle` class will automatically locate them. These files are plain text files, consisting of entries such as

```
readingFile=Achtung! Datei wird eingelesen
renamingFile=Datei wird umbenannt
...
```

When requesting a logger, you can specify a resource bundle:

```
Logger logger = Logger.getLogger(loggerName, "com.mycompany.logmessages");
```

Then you specify the resource bundle key, not the actual message string, for the log message.

```
logger.info("readingFile");
```

You often need to include arguments into localized messages. Then the message should contain placeholders `{0}`, `{1}`, and so on. For example, to include the file name with a log message, include the placeholder like this:

```
Reading file {0}.
Achtung! Datei {0} wird eingelesen.
```

You then pass values into the placeholders by calling one of the following methods:

```
logger.log(Level.INFO, "readingFile", fileName);
logger.log(Level.INFO, "renamingFile", new Object[] { oldName, newName });
```

### Handlers

By default, loggers send records to a `ConsoleHandler` that prints them to the `System.err` stream. Specifically, the logger sends the record to the parent handler, and the ultimate ancestor (with name `""`) has a `ConsoleHandler`.

Like loggers, handlers have a logging level. For a record to be logged, its logging level must be above the threshold of *both* the logger and the handler. The log manager configuration file sets the logging level of the default console handler as

```
java.util.logging.ConsoleHandler.level=INFO
```

To log records with level `FINE`, change both the default logger level and the handler level in the configuration. Alternatively, you can bypass the configuration file altogether and install your own handler.

```

Logger logger = Logger.getLogger("com.mycompany.myapp");
logger.setLevel(Level.FINE);
logger.setUseParentHandlers(false);
Handler handler = new ConsoleHandler();
handler.setLevel(Level.FINE);
logger.addHandler(handler);

```

By default, a logger sends records both to its own handlers and the handlers of the parent. Our logger is a child of the primordial logger (with name "") that sends all records with level INFO or higher to the console. But we don't want to see those records twice. For that reason, we set the `useParentHandlers` property to `false`.

To send log records elsewhere, add another handler. The logging API provides two useful handlers for this purpose, a `FileHandler` and a `SocketHandler`. The `SocketHandler` sends records to a specified host and port. Of greater interest is the `FileHandler` that collects records in a file.

You can simply send records to a default file handler, like this:

```

FileHandler handler = new FileHandler();
logger.addHandler(handler);

```

The records are sent to a file `java.n.log` in the user's home directory, where *n* is a number to make the file unique. If a user's system has no concept of the user's home directory (for example, in Windows 95/98/Me), then the file is stored in a default location such as `C:\Windows`. By default, the records are formatted in XML. A typical log record has the form

```

<record>
  <date>2002-02-04T07:45:15</date>
  <millis>1012837515710</millis>
  <sequence>1</sequence>
  <logger>com.mycompany.myapp</logger>
  <level>INFO</level>
  <class>com.mycompany.mylib.Reader</class>
  <method>read</method>
  <thread>10</thread>
  <message>Reading file corejava.gif</message>
</record>

```

You can modify the default behavior of the file handler by setting various parameters in the log manager configuration (see Table 11–2), or by using another constructor (see the API notes at the end of this section).

You probably don't want to use the default log file name. Therefore, you should use another pattern, such as `%h/myapp.log`. (See Table 11–3 for an explanation of the pattern variables.)

If multiple applications (or multiple copies of the same application) use the same log file, then you should turn the "append" flag on. Alternatively, use `%u` in the file name pattern so that each application creates a unique copy of the log.

It is also a good idea to turn file rotation on. Log files are kept in a rotation sequence, such as `myapp.log.0`, `myapp.log.1`, `myapp.log.2`, and so on. Whenever a file exceeds the size limit, the oldest log is deleted, the other files are renamed, and a new file with generation number 0 is created.

**Table 11-2 File Handler Configuration Parameters**

Configuration Property	Description	Default
<code>java.util.logging. FileHandler.level</code>	The handler level.	<code>Level.ALL</code>
<code>java.util.logging. FileHandler.append</code>	Controls whether the handler should append to an existing file, or open a new file for each program run.	<code>false</code>
<code>java.util.logging. FileHandler.limit</code>	The approximate maximum number of bytes to write in a file before opening another. (0 = no limit).	0 (no limit) in the <code>FileHandler</code> class, 50000 in the default log manager configuration
<code>java.util.logging. FileHandler.pattern</code>	The pattern for the log file name. See Table 11-3 for pattern variables.	<code>%h/java%.log</code>
<code>java.util.logging. FileHandler.count</code>	The number of logs in a rotation sequence.	1 (no rotation)
<code>java.util.logging. FileHandler.filter</code>	The filter class to use.	No filtering
<code>java.util.logging. FileHandler.encoding</code>	The character encoding to use.	The platform encoding
<code>java.util.logging. FileHandler.formatter</code>	The record formatter.	<code>java.util.logging. XMLFormatter</code>



TIP: Many programmers use logging as an aid for the technical support staff. If a program misbehaves in the field, then the user can send back the log files for inspection. In that case, you should turn the “append” flag on, use rotating logs, or both.

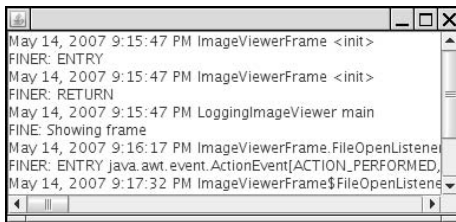
**Table 11-3 Log File Pattern Variables**

Variable	Description
<code>%h</code>	The value of the user <code>.home</code> system property.
<code>%t</code>	The system temporary directory.
<code>%u</code>	A unique number to resolve conflicts.
<code>%g</code>	The generation number for rotated logs. (A <code>%.g</code> suffix is used if rotation is specified and the pattern doesn't contain <code>%g</code> .)
<code>%%</code>	The <code>%</code> character.

You can also define your own handlers by extending the `Handler` or the `StreamHandler` class. We define such a handler in the example program at the end of this section. That handler displays the records in a window (see Figure 11–2).

The handler extends the `StreamHandler` class and installs a stream whose write methods display the stream output in a text area.

```
class WindowHandler extends StreamHandler
{
    public WindowHandler()
    {
        . . .
        final JTextArea output = new JTextArea();
        setOutputStream(new
            OutputStream()
            {
                public void write(int b) {} // not called
                public void write(byte[] b, int off, int len)
                {
                    output.append(new String(b, off, len));
                }
            });
    }
    . . .
}
```



**Figure 11–2** A log handler that displays records in a window

There is just one problem with this approach—the handler buffers the records and only writes them to the stream when the buffer is full. Therefore, we override the `publish` method to flush the buffer after each record:

```
class WindowHandler extends StreamHandler
{
    . . .
    public void publish(LogRecord record)
    {
        super.publish(record);
        flush();
    }
}
```

If you want to write more exotic stream handlers, extend the `Handler` class and define the `publish`, `flush`, and `close` methods.

**Filters**

By default, records are filtered according to their logging levels. Each logger and handler can have an optional filter to perform added filtering. You define a filter by implementing the `Filter` interface and defining the method

```
boolean isLoggable(LogRecord record)
```

Analyze the log record, using any criteria that you desire, and return `true` for those records that should be included in the log. For example, a particular filter may only be interested in the messages generated by the `entering` and `exiting` methods. The filter should then call `record.getMessage()` and check whether it starts with `ENTRY` or `RETURN`.

To install a filter into a logger or handler, simply call the `setFilter` method. Note that you can have at most one filter at a time.

**Formatters**

The `ConsoleHandler` and `FileHandler` classes emit the log records in text and XML formats. However, you can define your own formats as well. You need to extend the `Formatter` class and override the method

```
String format(LogRecord record)
```

Format the information in the record in any way you like and return the resulting string. In your format method, you may want to call the method

```
String formatMessage(LogRecord record)
```

That method formats the message part of the record, substituting parameters and applying localization.

Many file formats (such as XML) require a head and tail part that surrounds the formatted records. In that case, override the methods

```
String getHead(Handler h)
String getTail(Handler h)
```

Finally, call the `setFormatter` method to install the formatter into the handler.

**A Logging Recipe**

With so many options for logging, it is easy to lose track of the fundamentals. The following recipe summarizes the most common operations.

1. For a simple application, choose a single logger. It is a good idea to give the logger the same name as your main application package, such as `com.mycompany.myprog`. You can always get the logger by calling

```
Logger logger = Logger.getLogger("com.mycompany.myprog");
```

For convenience, you may want to add static fields

```
private static final Logger logger = Logger.getLogger("com.mycompany.myprog");
```

to classes with a lot of logging activity.

2. The default logging configuration logs all messages of level `INFO` or higher to the console. Users can override the default configuration, but as you have seen, the process is a bit involved. Therefore, it is a good idea to install a more reasonable default in your application.

The following code ensures that all messages are logged to an application-specific file. Place the code into the `main` method of your application.

```

if (System.getProperty("java.util.logging.config.class") == null
    && System.getProperty("java.util.logging.config.file") == null)
{
    try
    {
        Logger.getLogger("").setLevel(Level.ALL);
        final int LOG_ROTATION_COUNT = 10;
        Handler handler = new FileHandler("%h/myapp.log", 0, LOG_ROTATION_COUNT);
        Logger.getLogger("").addHandler(handler);
    }
    catch (IOException e)
    {
        logger.log(Level.SEVERE, "Can't create log file handler", e);
    }
}

```

3. Now you are ready to log to your heart's content. Keep in mind that all messages with level INFO, WARNING, and SEVERE show up on the console. Therefore, reserve these levels for messages that are meaningful to the users of your program. The level FINE is a good choice for logging messages that are intended for programmers.

Whenever you are tempted to call `System.out.println`, emit a log message instead:

```
logger.fine("File open dialog canceled");
```

It is also a good idea to log unexpected exceptions. For example:

```

try
{
    . . .
}
catch (SomeException e)
{
    logger.log(Level.FINE, "explanation", e);
}

```

Listing 11–2 puts this recipe to use with an added twist: Logging messages are also displayed in a log window.

#### Listing 11–2 LoggingImageViewer.java

```

1. import java.awt.*;
2. import java.awt.event.*;
3. import java.io.*;
4. import java.util.logging.*;
5. import javax.swing.*;
6.
7. /**
8.  * A modification of the image viewer program that logs various events.
9.  * @version 1.02 2007-05-31
10. * @author Cay Horstmann
11. */

```



**Listing 11-2** LoggingImageViewer.java (continued)

```
12. public class LoggingImageViewer
13. {
14.     public static void main(String[] args)
15.     {
16.         if (System.getProperty("java.util.logging.config.class") == null
17.             && System.getProperty("java.util.logging.config.file") == null)
18.         {
19.             try
20.             {
21.                 Logger.getLogger("com.horstmann.corejava").setLevel(Level.ALL);
22.                 final int LOG_ROTATION_COUNT = 10;
23.                 Handler handler = new FileHandler("%h/LoggingImageViewer.log", 0, LOG_ROTATION_COUNT);
24.                 Logger.getLogger("com.horstmann.corejava").addHandler(handler);
25.             }
26.             catch (IOException e)
27.             {
28.                 Logger.getLogger("com.horstmann.corejava").log(Level.SEVERE,
29.                     "Can't create log file handler", e);
30.             }
31.         }
32.
33.         EventQueue.invokeLater(new Runnable()
34.         {
35.             public void run()
36.             {
37.                 Handler windowHandler = new WindowHandler();
38.                 windowHandler.setLevel(Level.ALL);
39.                 Logger.getLogger("com.horstmann.corejava").addHandler(windowHandler);
40.
41.                 JFrame frame = new ImageViewerFrame();
42.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
43.
44.                 Logger.getLogger("com.horstmann.corejava").fine("Showing frame");
45.                 frame.setVisible(true);
46.             }
47.         });
48.     }
49. }
50.
51. /**
52.  * The frame that shows the image.
53.  */
54. class ImageViewerFrame extends JFrame
55. {
56.     public ImageViewerFrame()
57.     {
58.         logger.entering("ImageViewerFrame", "<i>init");
59.         setTitle("LoggingImageViewer");
60.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
61.     }

```

**Listing 11-2** LoggingImageViewer.java (continued)

```
62.    // set up menu bar
63.    JMenuBar menuBar = new JMenuBar();
64.    setJMenuBar(menuBar);
65.
66.    JMenu menu = new JMenu("File");
67.    menuBar.add(menu);
68.
69.    JMenuItem openItem = new JMenuItem("Open");
70.    menu.add(openItem);
71.    openItem.addActionListener(new FileOpenListener());
72.
73.    JMenuItem exitItem = new JMenuItem("Exit");
74.    menu.add(exitItem);
75.    exitItem.addActionListener(new ActionListener()
76.    {
77.        public void actionPerformed(ActionEvent event)
78.        {
79.            logger.fine("Exiting.");
80.            System.exit(0);
81.        }
82.    });
83.
84.    // use a label to display the images
85.    label = new JLabel();
86.    add(label);
87.    logger.exiting("ImageViewerFrame", "<init>");
88. }
89.
90. private class FileOpenListener implements ActionListener
91. {
92.     public void actionPerformed(ActionEvent event)
93.     {
94.         logger.entering("ImageViewerFrame.FileOpenListener", "actionPerformed", event);
95.
96.         // set up file chooser
97.         JFileChooser chooser = new JFileChooser();
98.         chooser.setCurrentDirectory(new File("."));
99.
100.        // accept all files ending with .gif
101.        chooser.setFileFilter(new javax.swing.filechooser.FileFilter()
102.        {
103.            public boolean accept(File f)
104.            {
105.                return f.getName().toLowerCase().endsWith(".gif") || f.isDirectory();
106.            }
107.
108.            public String getDescription()
109.            {
110.                return "GIF Images";
111.            }
112.        });
113.    }
114. }
```

**Listing 11-2** LoggingImageViewer.java (continued)

```
112.     });
113.
114.     // show file chooser dialog
115.     int r = chooser.showOpenDialog(ImageViewerFrame.this);
116.
117.     // if image file accepted, set it as icon of the label
118.     if (r == JFileChooser.APPROVE_OPTION)
119.     {
120.         String name = chooser.getSelectedFile().getPath();
121.         logger.log(Level.FINE, "Reading file {0}", name);
122.         label.setIcon(new ImageIcon(name));
123.     }
124.     else logger.fine("File open dialog canceled.");
125.     logger.exiting("ImageViewerFrame.FileOpenListener", "actionPerformed");
126. }
127. }
128.
129. private JLabel label;
130. private static Logger logger = Logger.getLogger("com.horstmann.corejava");
131. private static final int DEFAULT_WIDTH = 300;
132. private static final int DEFAULT_HEIGHT = 400;
133. }
134.
135. /**
136.  * A handler for displaying log records in a window.
137.  */
138. class WindowHandler extends StreamHandler
139. {
140.     public WindowHandler()
141.     {
142.         frame = new JFrame();
143.         final JTextArea output = new JTextArea();
144.         output.setEditable(false);
145.         frame.setSize(200, 200);
146.         frame.add(new JScrollPane(output));
147.         frame.setFocusableWindowState(false);
148.         frame.setVisible(true);
149.         setOutputStream(new OutputStream()
150.         {
151.             public void write(int b)
152.             {
153.                 } // not called
154.
155.             public void write(byte[] b, int off, int len)
156.             {
157.                 output.append(new String(b, off, len));
158.             }
159.         });
160.     }
161. }
```

**Listing 11–2** LoggingImageViewer.java (continued)

```

162. public void publish(LogRecord record)
163. {
164.     if (!frame.isVisible()) return;
165.     super.publish(record);
166.     flush();
167. }
168.
169. private JFrame frame;
170. }

```

**API** java.util.logging.Logger 1.4

- Logger getLogger(String loggerName)
- Logger getLogger(String loggerName, String bundleName)  
gets the logger with the given name. If the logger doesn't exist, it is created.
- Parameters:*
  - loggerName The hierarchical logger name, such as com.mycompany.myapp
  - bundleName The name of the resource bundle for looking up localized messages
- void severe(String message)
- void warning(String message)
- void info(String message)
- void config(String message)
- void fine(String message)
- void finer(String message)
- void finest(String message)  
logs a record with the level indicated by the method name and the given message.
- void entering(String className, String methodName)
- void entering(String className, String methodName, Object param)
- void entering(String className, String methodName, Object[] param)
- void exiting(String className, String methodName)
- void exiting(String className, String methodName, Object result)  
logs a record that describes entering or exiting a method with the given parameter(s) or return value.
- void throwing(String className, String methodName, Throwable t)  
logs a record that describes throwing of the given exception object.
- void log(Level level, String message)
- void log(Level level, String message, Object obj)
- void log(Level level, String message, Object[] objs)
- void log(Level level, String message, Throwable t)  
logs a record with the given level and message, optionally including objects or a throwable. To include objects, the message must contain formatting placeholders {0}, {1}, and so on.

- `void logp(Level level, String className, String methodName, String message)`
- `void logp(Level level, String className, String methodName, String message, Object obj)`
- `void logp(Level level, String className, String methodName, String message, Object[] objs)`
- `void logp(Level level, String className, String methodName, String message, Throwable t)`  
logs a record with the given level, precise caller information, and message, optionally including objects or a throwable.
- `void logrb(Level level, String className, String methodName, String bundleName, String message)`
- `void logrb(Level level, String className, String methodName, String bundleName, String message, Object obj)`
- `void logrb(Level level, String className, String methodName, String bundleName, String message, Object[] objs)`
- `void logrb(Level level, String className, String methodName, String bundleName, String message, Throwable t)`  
logs a record with the given level, precise caller information, resource bundle name, and message, optionally including objects or a throwable.
- `Level getLevel()`
- `void setLevel(Level l)`  
gets and sets the level of this logger.
- `Logger getParent()`
- `void setParent(Logger l)`  
gets and sets the parent logger of this logger.
- `Handler[] getHandlers()`  
gets all handlers of this logger.
- `void addHandler(Handler h)`
- `void removeHandler(Handler h)`  
adds or removes a handler for this logger.
- `boolean getUseParentHandlers()`
- `void setUseParentHandlers(boolean b)`  
gets and sets the “use parent handler” property. If this property is true, the logger forwards all logged records to the handlers of its parent.
- `Filter getFilter()`
- `void setFilter(Filter f)`  
gets and sets the filter of this logger.

**API** `java.util.logging.Handler` 1.4

- `abstract void publish(LogRecord record)`  
sends the record to the intended destination.
- `abstract void flush()`  
flushes any buffered data.
- `abstract void close()`  
flushes any buffered data and releases all associated resources.
- `Filter getFilter()`
- `void setFilter(Filter f)`  
gets and sets the filter of this handler.

- `Formatter getFormatter()`
- `void setFormatter(Formatter f)`  
gets and sets the formatter of this handler.
- `Level getLevel()`
- `void setLevel(Level l)`  
gets and sets the level of this handler.

**API** `java.util.logging.ConsoleHandler` 1.4

- `ConsoleHandler()`  
constructs a new console handler.

**API** `java.util.logging.FileHandler` 1.4

- `FileHandler(String pattern)`
- `FileHandler(String pattern, boolean append)`
- `FileHandler(String pattern, int limit, int count)`
- `FileHandler(String pattern, int limit, int count, boolean append)`  
constructs a file handler.

<i>Parameters:</i>	<code>pattern</code>	The pattern for constructing the log file name. See Table 11–3 on page 581 for pattern variables.
	<code>limit</code>	The approximate maximum number of bytes before a new log file is opened.
	<code>count</code>	The number of files in a rotation sequence.
	<code>append</code>	true if a newly constructed file handler object should append to an existing log file.

**API** `java.util.logging.LogRecord` 1.4

- `Level getLevel()`  
gets the logging level of this record.
- `String getLoggerName()`  
gets the name of the logger that is logging this record.
- `ResourceBundle getResourceBundle()`
- `String getResourceBundleName()`  
gets the resource bundle, or its name, to be used for localizing the message, or null if none is provided.
- `String getMessage()`  
gets the “raw” message before localization or formatting.
- `Object[] getParameters()`  
gets the parameter objects, or null if none is provided.
- `Throwable getThrown()`  
gets the thrown object, or null if none is provided.

- `String getSourceClassName()`
- `String getSourceMethodName()`  
gets the location of the code that logged this record. This information may be supplied by the logging code or automatically inferred from the runtime stack. It might be inaccurate, if the logging code supplied the wrong value or if the running code was optimized and the exact location cannot be inferred.
- `Long getMillis()`  
gets the creation time, in milliseconds, since 1970.
- `Long getSequenceNumber()`  
gets the unique sequence number of this record.
- `int getThreadID()`  
gets the unique ID for the thread in which this record was created. These IDs are assigned by the `LogRecord` class and have no relationship to other thread IDs.

**API** `java.util.logging.Filter` 1.4

- `boolean isLoggable(LogRecord record)`  
returns true if the given log record should be logged.

**API** `java.util.logging.Formatter` 1.4

- `abstract String format(LogRecord record)`  
returns the string that results from formatting the given log record.
- `String getHead(Handler h)`
- `String getTail(Handler h)`  
returns the strings that should appear at the head and tail of the document containing the log records. The `Formatter` superclass defines these methods to return the empty string; override them if necessary.
- `String formatMessage(LogRecord record)`  
returns the localized and formatted message part of the log record.

## Debugging Tips

Suppose you wrote your program and made it bulletproof by catching and properly handling all exceptions. Then you run it, and it does not work right. Now what? (If you never have this problem, you can skip the remainder of this chapter.)

Of course, it is best if you have a convenient and powerful debugger. Debuggers are available as a part of professional development environments such as Eclipse and NetBeans. We discuss the debugger later in this chapter. In this section, we offer you a number of tips that may be worth trying before you launch the debugger.

1. You can print or log the value of any variable with code like this:

```
System.out.println("x=" + x);
```

or

```
Logger.global.info("x=" + x);
```

If `x` is a number, it is converted to its string equivalent. If `x` is an object, then Java calls its `toString` method. To get the state of the implicit parameter object, print the state of the `this` object.

```
Logger.global.info("this=" + this);
```

Most of the classes in the Java library are very conscientious about overriding the `toString` method to give you useful information about the class. This is a real boon for debugging. You should make the same effort in your classes.

2. One seemingly little-known but very useful trick is that you can put a separate `main` method in each class. Inside it, you can put a unit test stub that lets you test the class in isolation.

```
public class MyClass
{
    methods and fields
    . . .
    public static void main(String[] args)
    {
        test code
    }
}
```

Make a few objects, call all methods, and check that each of them does the right thing. You can leave all these `main` methods in place and launch the Java virtual machine separately on each of the files to run the tests. When you run an applet, none of these `main` methods are ever called. When you run an application, the Java virtual machine calls only the `main` method of the startup class.

3. If you liked the preceding tip, you should check out JUnit from <http://junit.org>. JUnit is a very popular unit testing framework that makes it easy to organize suites of test cases. Run the tests whenever you make changes to a class, and add another test case whenever you find a bug.
4. A *logging proxy* is an object of a subclass that intercepts method calls, logs them, and then calls the superclass. For example, if you have trouble with the `setBackground` method of a panel, you can create a proxy object as an instance of an anonymous subclass:

```
JPanel panel = new
    JPanel()
    {
        public void setBackground(Color c)
        {
            Logger.global.info("setBackground: c=" + c);
            super.setBackground(c);
        }
    };
```

Whenever the `setBackground` method is called, a log message is generated. To find out who called the method, generate a stack trace.



5. You can get a stack trace from any exception object with the `printStackTrace` method in the `Throwable` class. The following code catches any exception, prints the exception object and the stack trace, and rethrows the exception so it can find its intended handler.

```
try
{
    . . .
}
catch (Throwable t)
{
    t.printStackTrace();
    throw t;
}
```

You don't even need to catch an exception to generate a stack trace. Simply insert the statement

```
Thread.dumpStack();
```

anywhere into your code to get a stack trace.

6. Normally, the stack trace is displayed on `System.err`. You can send it to a file with the void `printStackTrace(PrintWriter s)` method. Or, if you want to log or display the stack trace, here is how you can capture it into a string:

```
StringWriter out = new StringWriter();
new Throwable().printStackTrace(new PrintWriter(out));
String trace = out.toString();
```

(See Chapter 1 of Volume II for the `PrintWriter` and `StringWriter` classes.)

7. It is often handy to trap program errors in a file. However, errors are sent to `System.err`, not `System.out`. Therefore, you cannot simply trap them by running

```
java MyProgram > errors.txt
```

Instead, capture the error stream as

```
java MyProgram 2> errors.txt
```

To capture both `System.err` and `System.out` in the same file, use

```
java MyProgram >& errors.txt
```

This works in `bash` and the Windows shell.

8. Having stack traces of uncaught exceptions show up in `System.err` is not ideal. These messages are confusing to end users if they happen to see them, and they are not available for diagnostic purposes when you need them. A better approach is to log them to a file. As of Java SE 5.0, you can change the handler for uncaught exceptions with the static `Thread.setDefaultUncaughtExceptionHandler` method:

```
Thread.setDefaultUncaughtExceptionHandler(
    new Thread.UncaughtExceptionHandler()
    {
        public void uncaughtException(Thread t, Throwable e)
        {
            save information in log file
        }
    });
```

9. To watch class loading, launch the Java virtual machine with the `-verbose` flag. You get a printout such as the following:

```
[Opened /usr/local/jdk5.0/jre/lib/rt.jar]
[Opened /usr/local/jdk5.0/jre/lib/jsse.jar]
[Opened /usr/local/jdk5.0/jre/lib/jce.jar]
[Opened /usr/local/jdk5.0/jre/lib/charsets.jar]
[Loaded java.lang.Object from shared objects file]
[Loaded java.io.Serializable from shared objects file]
[Loaded java.lang.Comparable from shared objects file]
[Loaded java.lang.CharSequence from shared objects file]
[Loaded java.lang.String from shared objects file]
[Loaded java.lang.reflect.GenericDeclaration from shared objects file]
[Loaded java.lang.reflect.Type from shared objects file]
[Loaded java.lang.reflect.AnnotatedElement from shared objects file]
[Loaded java.lang.Class from shared objects file]
[Loaded java.lang.Cloneable from shared objects file]
...
```

This can occasionally be helpful to diagnose class path problems.

10. If you ever looked at a Swing window and wondered how its designer managed to get all the components to line up so nicely, you can spy on the contents. Press `CTRL+SHIFT+F1`, and you get a printout of all components in the hierarchy:

```
FontDialog[frame0,0,0,300x200,layout=java.awt.BorderLayout,...
javax.swing.JRootPane[,4,23,292x173,layout=javafx.swing.JRootPane$RootLayout,...
javax.swing.JPanel[null.glassPane,0,0,292x173,hidden,layout=java.awt.FlowLayout,...
javax.swing.JLayeredPane[null.layeredPane,0,0,292x173,...
javax.swing.JPanel[null.contentPane,0,0,292x173,layout=java.awt.GridBagLayout,...
javax.swing.JList[,0,0,73x152,alignmentX=null,alignmentY=null,...
javax.swing.CellRendererPane[,0,0,0x0,hidden]
javax.swing.DefaultListCellRenderer$UIResource[,-73,-19,0x0,...
javax.swing.JCheckBox[,157,13,50x25,layout=javafx.swing.OverlayLayout,...
javax.swing.JCheckBox[,156,65,52x25,layout=javafx.swing.OverlayLayout,...
javax.swing.JLabel[,114,119,30x17,alignmentX=0.0,alignmentY=null,...
javax.swing.JTextField[,186,117,105x21,alignmentX=null,alignmentY=null,...
javax.swing.JTextField[,0,152,291x21,alignmentX=null,alignmentY=null,...
```

11. If you design your own custom Swing component and it doesn't seem to be displayed correctly, you'll really love the *Swing graphics debugger*. And even if you don't write your own component classes, it is instructive and fun to see exactly how the contents of a component are drawn. To turn on debugging for a Swing component, use the `setDebugGraphicsOptions` method of the `JComponent` class. The following options are available:

<code>DebugGraphics.FLASH_OPTION</code>	Flashes each line, rectangle, and text in red before drawing it
<code>DebugGraphics.LOG_OPTION</code>	Prints a message for each drawing operation
<code>DebugGraphics.BUFFERED_OPTION</code>	Displays the operations that are performed on the off-screen buffer
<code>DebugGraphics.NONE_OPTION</code>	Turns graphics debugging off

We have found that for the flash option to work, you must disable “double buffering,” the strategy used by Swing to reduce flicker when updating a window. The magic incantation for turning on the flash option is

```
RepaintManager.currentManager(getRootPane()).setDoubleBufferingEnabled(false);
((JComponent) getContentPane()).setDebugGraphicsOptions(DebugGraphics.FLASH_OPTION);
```

Simply place these lines at the end of your frame constructor. When the program runs, you will see the content pane filled in slow motion. Or, for more localized debugging, just call `setDebugGraphicsOptions` for a single component. Control freaks can set the duration, count, and color of the flashes—see the on-line documentation of the `DebugGraphics` class for details.

12. Java SE 5.0 added the `-Xlint` option to the compiler for spotting common code problems. For example, if you compile with the command

```
javac -Xlint:fallthrough
```

then the compiler reports missing break statements in switch statements. (The term “lint” originally described a tool for locating potential problems in C programs, and is now generically applied to tools that flag constructs that are questionable but not illegal.)

The following options are available:

<code>-Xlint</code> or <code>-Xlint:all</code>	Carries out all checks
<code>-Xlint:deprecation</code>	Same as <code>-deprecation</code> , checks for deprecated methods
<code>-Xlint:fallthrough</code>	Checks for missing break statements in switch statements
<code>-Xlint:finally</code>	Warns about finally clauses that cannot complete normally
<code>-Xlint:none</code>	Carries out none of the checks
<code>-Xlint:path</code>	Checks that all directories on the class path and source path exist
<code>-Xlint:serial</code>	Warns about serializable classes without <code>serialVersionUID</code> (see Chapter 1 of Volume II)
<code>-Xlint:unchecked</code>	Warns of unsafe conversions between generic and raw types (see Chapter 12)

13. Java SE 5.0 added support for *monitoring and management* of Java applications, allowing the installation of agents in the virtual machine that track memory consumption, thread usage, class loading, and so on. This feature is particularly important for large and long-running Java programs such as application servers. As a demonstration of these capabilities, the JDK ships with a graphical tool called `jconsole` that displays statistics about the performance of a virtual machine (see Figure 11–3). Find out the ID of the operating system process that runs the virtual machine. In UNIX/Linux, run the `ps` utility; in Windows, use the task manager. Then launch the `jconsole` program:

```
jconsole processID
```

The console gives you a wealth of information about your running program. See <http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html> for more information.



NOTE: Prior to Java SE 6, you need to launch your program with the `-Dcom.sun.management.jmxremote` option:

```
java -Dcom.sun.management.jmxremote MyProgram
jconsole processID
```

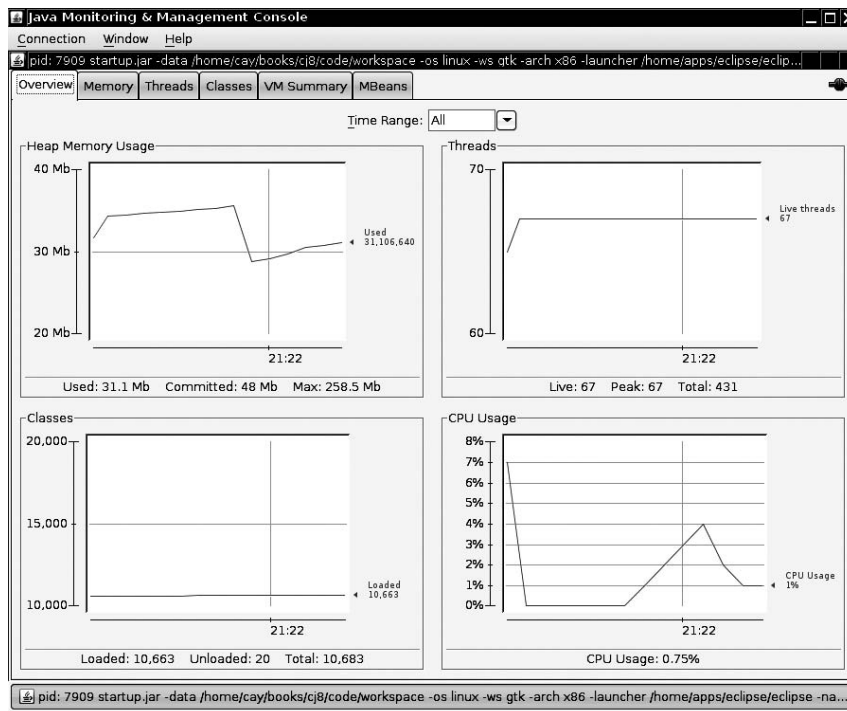


Figure 11-3 The jconsole program

14. You can use the `jmap` utility to get a heap dump that shows you every object on the heap. Use these commands:

```
jmap -dump:format=b,file=dumpFileName processID
jhat dumpFileName
```

Then, point your browser to `localhost:7000`. You will get a web application that lets you drill down into the contents of the heap at the time of the dump.

15. If you launch the Java virtual machine with the `-Xprof` flag, it runs a rudimentary *profiler* that keeps track of the methods in your code that were executed most often. The profiling information is sent to `System.out`. The output also tells you which methods were compiled by the just-in-time compiler.

**X** CAUTION: The `-X` options of the compiler are not officially supported and may not be present in all versions of the JDK. Run `java -X` to get a listing of all nonstandard options.

### Using a Console Window

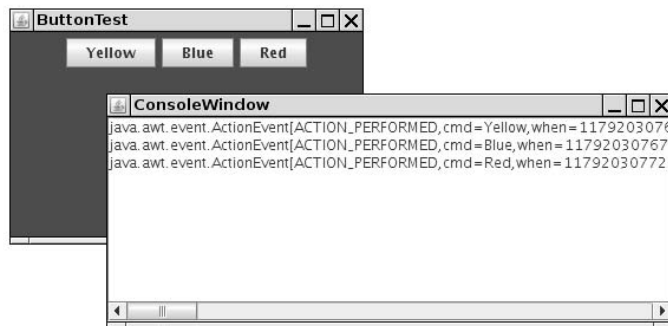
When you debug an applet, you can see error messages in a window: In the configuration panel of the Java Plug-in, check the Show Java Console box (see Chapter 10). The Java Console window has a set of scrollbars, so you can retrieve messages that have scrolled off the window. Windows users will find this a definite advantage over the DOS shell window in which the `System.out` and `System.err` output normally appears.

We give you a similar window class so you can enjoy the same benefit of seeing your debugging messages in a window when debugging a program. Figure 11-4 shows our `ConsoleWindow` class in action.

The class is easy to use. Simply call

```
ConsoleWindow.init()
```

Then print to `System.out` or `System.err` in the normal way.



**Figure 11-4** The console window

Listing 11-3 lists the code for the `ConsoleWindow` class. As you can see, the class is quite simple. Messages are displayed in a `JTextArea` inside a `JScrollPane`. We call the `System.setOut` and `System.setErr` methods to set the output and error streams to a special stream that adds all messages to the text area.

#### Listing 11-3 ConsoleWindow.java

```
1. import javax.swing.*;
2. import java.io.*;
3.
4. /**
5.  * A window that displays the bytes sent to System.out and System.err
6.  * @version 1.01 2004-05-10
7.  * @author Cay Horstmann
8.  */
```

**Listing 11-3** ConsoleWindow.java (continued)

```

9. public class ConsoleWindow
10. {
11.     public static void init()
12.     {
13.         JFrame frame = new JFrame();
14.         frame.setTitle("ConsoleWindow");
15.         final JTextArea output = new JTextArea();
16.         output.setEditable(false);
17.         frame.add(new JScrollPane(output));
18.         frame.setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
19.         frame.setLocation(DEFAULT_LEFT, DEFAULT_TOP);
20.         frame.setFocusableWindowState(false);
21.         frame.setVisible(true);
22.
23.         // define a PrintStream that sends its bytes to the output text area
24.         PrintStream consoleStream = new PrintStream(new
25.             OutputStream()
26.             {
27.                 public void write(int b) {} // never called
28.                 public void write(byte[] b, int off, int len)
29.                 {
30.                     output.append(new String(b, off, len));
31.                 }
32.             });
33.
34.         // set both System.out and System.err to that stream
35.         System.setOut(consoleStream);
36.         System.setErr(consoleStream);
37.     }
38.
39.     public static final int DEFAULT_WIDTH = 300;
40.     public static final int DEFAULT_HEIGHT = 200;
41.     public static final int DEFAULT_LEFT = 200;
42.     public static final int DEFAULT_TOP = 200;
43. }

```

**Tracing AWT Events**

When you write a fancy user interface in Java, you need to know what events AWT sends to what components. Unfortunately, the AWT documentation is somewhat sketchy in this regard. For example, suppose you want to show hints in the status line when the user moves the mouse over different parts of the screen. The AWT generates mouse and focus events that you may be able to trap.

We give you a useful `EventTracer` class to spy on these events. It prints out all event handling methods and their parameters. See Figure 11-5 for a display of the traced events.

To spy on messages, add the component whose events you want to trace to an event tracer:

```

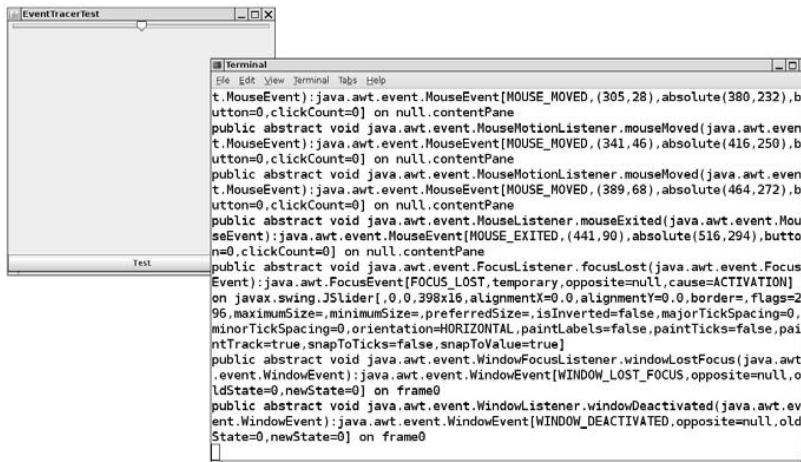
EventTracer tracer = new EventTracer();
tracer.add(frame);

```

That prints a textual description of all events, like this:

```
public abstract void java.awt.event.MouseListener.mouseExited(java.awt.event.MouseEvent):
java.awt.event.MouseEvent[MOUSE_EXITED,(408,14),button=0,clickCount=0] on javax.swing.JBut-
ton[,0,345,400x25,...]
public abstract void java.awt.event.FocusListener.focusLost(java.awt.event.FocusEvent):
java.awt.event.FocusEvent[FOCUS_LOST,temporary,opposite=null] on javax.swing.JButton[,0,345,400x25,...]
```

You may want to capture this output in a file or a console window, as explained in the preceding sections.



**Figure 11-5** The EventTracer class at work

Listing 11-4 is the EventTracer class. The idea behind the class is easy even if the implementation is a bit mysterious. Here are the steps that are carried out behind the scenes:

1. When you add a component to the event tracer in the `add` method, the JavaBeans introspection class analyzes the component for methods of the form `void addXXXListener(XXXListener)`. (See Chapter 8 of Volume II for more information on JavaBeans.) For each matching method, an `EventSetDescriptor` is generated. We pass each descriptor to the `addListener` method.
2. If the component is a container, we enumerate its components and recursively call `add` for each of them.
3. The `addListener` method is called with two parameters: the component on whose events we want to spy and the event set descriptor. The `getListenerType` method of the `EventSetDescriptor` class returns a `Class` object that describes the event listener interface such as `ActionListener` or `ChangeListener`. We create a proxy object for that interface. The proxy handler simply prints the name and event parameter of the invoked event method. The `getAddListenerMethod` method of the `EventSetDescriptor` class returns a `Method` object that we use to add the proxy object as the event listener to the component.

This program is a good example of the power of the reflection mechanism. We don't have to hardwire the fact that the `JButton` class has a method `addActionListener` whereas a `JSlider` has a method `addChangeListener`. The reflection mechanism discovers these facts for us.

Listing 11-5 tests the event tracer. The program displays a frame with a button and a slider and traces the events that these components generate.

**Listing 11-4** EventTracer.java

```
1. import java.awt.*;
2. import java.beans.*;
3. import java.lang.reflect.*;
4.
5. /**
6.  * @version 1.31 2004-05-10
7.  * @author Cay Horstmann
8.  */
9. public class EventTracer
10. {
11.     public EventTracer()
12.     {
13.         // the handler for all event proxies
14.         handler = new InvocationHandler()
15.         {
16.             public Object invoke(Object proxy, Method method, Object[] args)
17.             {
18.                 System.out.println(method + ":" + args[0]);
19.                 return null;
20.             }
21.         };
22.     }
23.
24.     /**
25.     * Adds event tracers for all events to which this component and its children can listen
26.     * @param c a component
27.     */
28.     public void add(Component c)
29.     {
30.         try
31.         {
32.             // get all events to which this component can listen
33.             BeanInfo info = Introspector.getBeanInfo(c.getClass());
34.
35.             EventSetDescriptor[] eventSets = info.getEventSetDescriptors();
36.             for (EventSetDescriptor eventSet : eventSets)
37.                 addListener(c, eventSet);
```



**Listing 11-4** EventTracer.java (continued)

```
38.     }
39.     catch (IntrospectionException e)
40.     {
41.     }
42.     // ok not to add listeners if exception is thrown
43.
44.     if (c instanceof Container)
45.     {
46.         // get all children and call add recursively
47.         for (Component comp : ((Container) c).getComponents())
48.             add(comp);
49.     }
50. }
51.
52. /**
53.  * Add a listener to the given event set
54.  * @param c a component
55.  * @param eventSet a descriptor of a listener interface
56.  */
57. public void addListener(Component c, EventSetDescriptor eventSet)
58. {
59.     // make proxy object for this listener type and route all calls to the handler
60.     Object proxy = Proxy.newProxyInstance(null, new Class[] { eventSet.getListenerType() },
61.         handler);
62.
63.     // add the proxy as a listener to the component
64.     Method addListenerMethod = eventSet.getAddListenerMethod();
65.     try
66.     {
67.         addListenerMethod.invoke(c, proxy);
68.     }
69.     catch (InvocationTargetException e)
70.     {
71.     }
72.     catch (IllegalAccessException e)
73.     {
74.     }
75.     // ok not to add listener if exception is thrown
76. }
77.
78. private InvocationHandler handler;
79. }
```

**Listing 11-5** EventTracerTest.java

```
1. import java.awt.*;
2.
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.13 2007-06-12
7.  * @author Cay Horstmann
8.  */
9. public class EventTracerTest
10. {
11.     public static void main(String[] args)
12.     {
13.         EventQueue.invokeLater(new Runnable()
14.         {
15.             public void run()
16.             {
17.                 JFrame frame = new EventTracerFrame();
18.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.                 frame.setVisible(true);
20.             }
21.         });
22.     }
23. }
24.
25. class EventTracerFrame extends JFrame
26. {
27.     public EventTracerFrame()
28.     {
29.         setTitle("EventTracerTest");
30.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
31.
32.         // add a slider and a button
33.         add(new JSlider(), BorderLayout.NORTH);
34.         add(new JButton("Test"), BorderLayout.SOUTH);
35.
36.         // trap all events of components inside the frame
37.         EventTracer tracer = new EventTracer();
38.         tracer.add(this);
39.     }
40.
41.     public static final int DEFAULT_WIDTH = 400;
42.     public static final int DEFAULT_HEIGHT = 400;
43. }
```

---

### **Letting the AWT Robot Do the Work**

Java SE 1.3 added a `Robot` class that you can use to send keystrokes and mouse clicks to any AWT program. This class is intended for automatic testing of user interfaces.

To get a robot, you need to first get a `GraphicsDevice` object. You get the default screen device through the sequence of calls:

```
GraphicsEnvironment environment = GraphicsEnvironment.getLocalGraphicsEnvironment();
GraphicsDevice screen = environment.getDefaultScreenDevice();
```

Then you construct a robot:

```
Robot robot = new Robot(screen);
```

To send a keystroke, tell the robot to simulate a key press and a key release:

```
robot.keyPress(KeyEvent.VK_TAB);
robot.keyRelease(KeyEvent.VK_TAB);
```

For a mouse click, you first need to move the mouse and then press and release a button:

```
robot.mouseMove(x, y); // x and y are absolute screen pixel coordinates.
robot.mousePress(InputEvent.BUTTON1_MASK);
robot.mouseRelease(InputEvent.BUTTON1_MASK);
```

The idea is that you simulate key and mouse input and afterwards take a screen snapshot to see whether the application did what it was supposed to. You capture the screen with the `createScreenCapture` method:

```
Rectangle rect = new Rectangle(x, y, width, height);
BufferedImage image = robot.createScreenCapture(rect);
```

The rectangle coordinates also refer to absolute screen pixels.

Finally, you usually want to add a small delay between robot instructions so that the application can catch up. Use the `delay` method and give it the number of milliseconds to delay. For example:

```
robot.delay(1000); // delay by 1000 milliseconds
```

The program in Listing 11–6 shows how you can use the robot. A robot tests the button test program that you saw in Chapter 8. First, pressing the space bar activates the leftmost button. Then the robot waits for two seconds so that you can see what it has done. After the delay, the robot simulates the tab key and another space bar press to click on the next button. Finally, we simulate a mouse click on the third button. (You may need to adjust the `x` and `y` coordinates of the program to actually press the button.) The program ends by taking a screen capture and displaying it in another frame (see Figure 11–6).

As you can see from this example, the `Robot` class is not by itself suitable for convenient user interface testing. Instead, it is a basic building block that can be a foundational part of a testing tool. A professional testing tool can capture, store, and replay user interaction scenarios and find out the screen locations of the components so that mouse clicks aren't guesswork. Hopefully, as Java applications are becoming more popular, we will see more sophisticated testing tools.

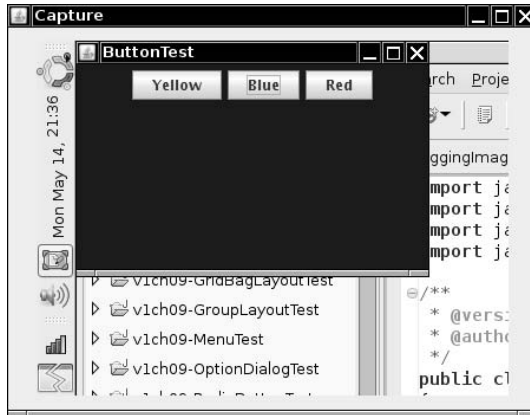


Figure 11-6 Capturing the screen with the AWT robot

**Listing 11-6** RobotTest.java

```

1. import java.awt.*;
2. import java.awt.event.*;
3. import java.awt.image.*;
4. import javax.swing.*;
5.
6. /**
7.  * @version 1.03 2007-06-12
8.  * @author Cay Horstmann
9.  */
10. public class RobotTest
11. {
12.     public static void main(String[] args)
13.     {
14.         EventQueue.invokeLater(new Runnable()
15.         {
16.             public void run()
17.             {
18.                 // make frame with a button panel
19.
20.                 ButtonFrame frame = new ButtonFrame();
21.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22.                 frame.setVisible(true);
23.
24.                 // attach a robot to the screen device
25.
26.                 GraphicsEnvironment environment = GraphicsEnvironment.getLocalGraphicsEnvironment();
27.                 GraphicsDevice screen = environment.getDefaultScreenDevice();

```

**Listing 11-6** RobotTest.java (continued)

```
28.
29.     try
30.     {
31.         Robot robot = new Robot(screen);
32.         runTest(robot);
33.     }
34.     catch (AWTException e)
35.     {
36.         e.printStackTrace();
37.     }
38. }
39. });
40. }
41.
42. /**
43.  * Runs a sample test procedure
44.  * @param robot the robot attached to the screen device
45.  */
46. public static void runTest(Robot robot)
47. {
48.     // simulate a space bar press
49.     robot.keyPress(' ');
50.     robot.keyRelease(' ');
51.
52.     // simulate a tab key followed by a space
53.     robot.delay(2000);
54.     robot.keyPress(KeyEvent.VK_TAB);
55.     robot.keyRelease(KeyEvent.VK_TAB);
56.     robot.keyPress(' ');
57.     robot.keyRelease(' ');
58.
59.     // simulate a mouse click over the rightmost button
60.     robot.delay(2000);
61.     robot.mouseMove(200, 50);
62.     robot.mousePress(InputEvent.BUTTON1_MASK);
63.     robot.mouseRelease(InputEvent.BUTTON1_MASK);
64.
65.     // capture the screen and show the resulting image
66.     robot.delay(2000);
67.     BufferedImage image = robot.createScreenCapture(new Rectangle(0, 0, 400, 300));
68.
69.     ImageFrame frame = new ImageFrame(image);
70.     frame.setVisible(true);
71. }
72. }
73.
74. /**
75.  * A frame to display a captured image
76.  */
```

**Listing 11-6** RobotTest.java (continued)

```

77. class ImageFrame extends JFrame
78. {
79.     /**
80.      * @param image the image to display
81.      */
82.     public ImageFrame(Image image)
83.     {
84.         setTitle("Capture");
85.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
86.
87.         JLabel label = new JLabel(new ImageIcon(image));
88.         add(label);
89.     }
90.
91.     public static final int DEFAULT_WIDTH = 450;
92.     public static final int DEFAULT_HEIGHT = 350;
93. }

```

**API** java.awt.GraphicsEnvironment 1.2

- static GraphicsEnvironment getLocalGraphicsEnvironment()  
returns the local graphics environment.
- GraphicsDevice getDefaultScreenDevice()  
returns the default screen device. Note that computers with multiple monitors have one graphics device per screen—use the getScreenDevices method to obtain an array of all screen devices.

**API** java.awt.Robot 1.3

- Robot(GraphicsDevice device)  
constructs a robot that can interact with the given device.
- void keyPress(int key)
- void keyRelease(int key)  
simulates a key press or release.  
*Parameters:*     key             The key code. See the KeyStroke class for more information on key codes
- void mouseMove(int x, int y)  
simulates a mouse move.  
*Parameters:*     x, y             The mouse position in absolute pixel coordinates
- void mousePress(int eventMask)
- void mouseRelease(int eventMask)  
simulates a mouse button press or release.  
*Parameters:*     eventMask       The event mask describing the mouse buttons. See the InputEvent class for more information on event masks

- `void delay(int milliseconds)`  
delays the robot for the given number of milliseconds.
- `BufferedImage createScreenCapture(Rectangle rect)`  
captures a portion of the screen.

*Parameters:*     `rect`           The rectangle to be captured, in absolute pixel coordinates

### Using a Debugger

Debugging with print statements is not one of life's more joyful experiences. You constantly find yourself adding and removing the statements, then recompiling the program. Using a debugger is better. A debugger runs your program in full motion until it reaches a breakpoint, and then you can look at everything that interests you.

Listing 11–7 show a deliberately corrupted version of the `ButtonTest` program from Chapter 8. When you click on any of the buttons, nothing happens. Look at the source code—button clicks are supposed to set the background color to the color specified by the button name.

#### Listing 11–7   BuggyButtonTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * @version 1.22 2007-05-14
7.  * @author Cay Horstmann
8.  */
9. public class BuggyButtonTest
10. {
11.     public static void main(String[] args)
12.     {
13.         EventQueue.invokeLater(new Runnable()
14.         {
15.             public void run()
16.             {
17.                 BuggyButtonFrame frame = new BuggyButtonFrame();
18.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
19.                 frame.setVisible(true);
20.             }
21.         });
22.     }
23. }
24.
25. class BuggyButtonFrame extends JFrame
26. {
27.     public BuggyButtonFrame()
28.     {
```

**Listing 11-7** BuggyButtonTest.java (continued)

```
29.     setTitle("BuggyButtonTest");
30.     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
31.
32.     // add panel to frame
33.
34.     BuggyButtonPanel panel = new BuggyButtonPanel();
35.     add(panel);
36. }
37.
38. public static final int DEFAULT_WIDTH = 300;
39. public static final int DEFAULT_HEIGHT = 200;
40. }
41.
42. class BuggyButtonPanel extends JPanel
43. {
44.     public BuggyButtonPanel()
45.     {
46.         ActionListener listener = new ButtonListener();
47.
48.         JButton yellowButton = new JButton("Yellow");
49.         add(yellowButton);
50.         yellowButton.addActionListener(listener);
51.
52.         JButton blueButton = new JButton("Blue");
53.         add(blueButton);
54.         blueButton.addActionListener(listener);
55.
56.         JButton redButton = new JButton("Red");
57.         add(redButton);
58.         redButton.addActionListener(listener);
59.     }
60.
61.     private class ButtonListener implements ActionListener
62.     {
63.         public void actionPerformed(ActionEvent event)
64.         {
65.             String arg = event.getActionCommand();
66.             if (arg.equals("yellow")) setBackground(Color.yellow);
67.             else if (arg.equals("blue")) setBackground(Color.blue);
68.             else if (arg.equals("red")) setBackground(Color.red);
69.         }
70.     }
71. }
```

In a program this short, you may be able to find the bug just by reading the source code. Let us pretend that scanning the source code for errors is not practical. We show you how to use the Eclipse debugger to locate the error.



✓ NOTE: If you use a stand-alone debugger such as JSwat (<http://www.bluemarsh.com/java/jswat/>) or the venerable and extremely clunky jdb, you must first compile your program with the `-g` option. For example:

```
javac -g BuggyButtonTest.java
```

In an integrated environment, this is done automatically.

In Eclipse, start the debugger with the menu option `Run -> Debug As -> Java Application`. The program will start running.

Set a breakpoint at the first line of the `actionPerformed` method: Right-click in the left margin, next to the line of code, and chose `Toggle Breakpoint`.

The breakpoint will be hit as soon as Java starts processing code in the `actionPerformed` method. For this, click on the Yellow button. The debugger breaks at the start of the `actionPerformed` method—see Figure 11–7.

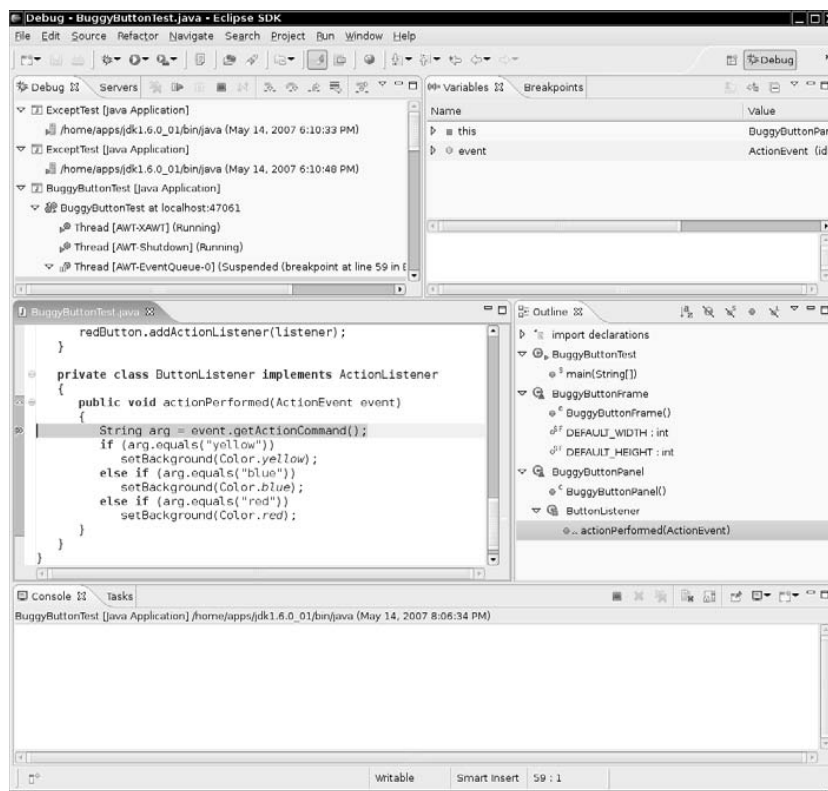
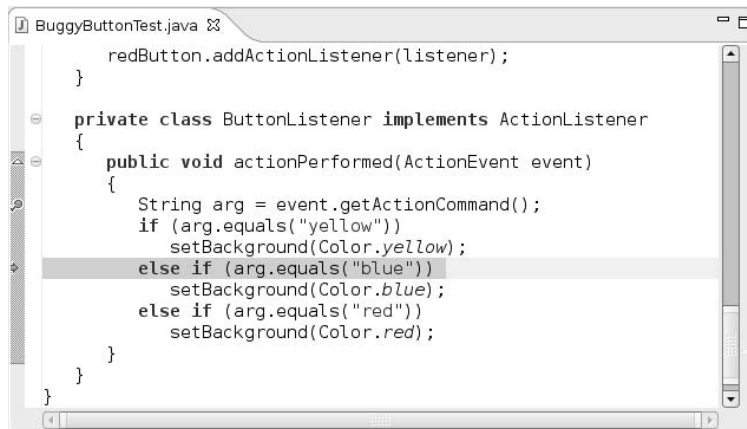


Figure 11–7 Stopping at a breakpoint

There are two basic commands to single-step through a program. The “Step Into” command steps into every method call. The “Step Over” command goes to the next line without stepping inside any further method calls. Eclipse uses menu options Run -> Step Into and Run -> Step Over, with keyboard shortcuts F5 and F6. Issue the “Step Over” command twice and see where you are.

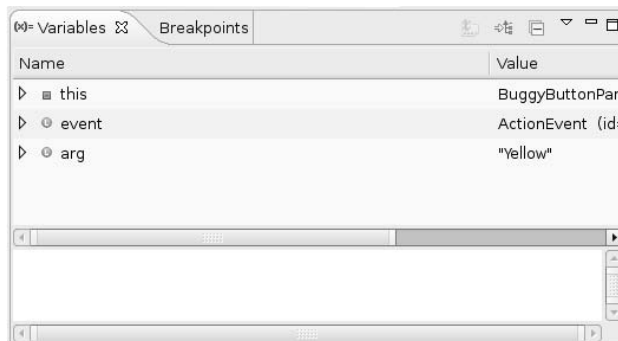


```
BuggyButtonTest.java
    redButton.addActionListener(listener);
}

private class ButtonListener implements ActionListener
{
    public void actionPerformed(ActionEvent event)
    {
        String arg = event.getActionCommand();
        if (arg.equals("yellow"))
            setBackground(Color.yellow);
        else if (arg.equals("blue"))
            setBackground(Color.blue);
        else if (arg.equals("red"))
            setBackground(Color.red);
    }
}
```

That is not what should have happened. The program was supposed to call `setColor(Color.yellow)` and then exit the method.

Inspect the local variables and check the value of the `arg` variable:



Name	Value
▶ this	BuggyButtonPan
▶ event	ActionEvent (id=
▶ arg	"Yellow"

Now you can see what happened. The value of `arg` was "Yellow", with an uppercase Y, but the comparison tested

```
if (arg.equals("yellow"))
```

with a lowercase `y`. Mystery solved.

To quit the debugger, select Run -> Terminate from the menu.

There are more advanced debugging commands in Eclipse, but you can get a long way with the simple techniques that you just saw. Other debuggers, such as the NetBeans debugger, have very similar commands.

This chapter introduced you to exception handling and gave you some useful hints for testing and debugging. The next two chapters cover generic programming and its most important application: the Java collections framework.



# *Chapter*

# 12

## GENERIC PROGRAMMING

- ▼ WHY GENERIC PROGRAMMING?
- ▼ DEFINITION OF A SIMPLE GENERIC CLASS
- ▼ GENERIC METHODS
- ▼ BOUNDS FOR TYPE VARIABLES
- ▼ GENERIC CODE AND THE VIRTUAL MACHINE
- ▼ RESTRICTIONS AND LIMITATIONS
- ▼ INHERITANCE RULES FOR GENERIC TYPES
- ▼ WILDCARD TYPES
- ▼ REFLECTION AND GENERICS

**G**enerics constitute the most significant change in the Java programming language since the 1.0 release. The addition of generics to Java SE 5.0 was the result of one of the first Java Specification Requests, JSR 14, that was formulated in 1999. The expert group spent about five years on specifications and test implementations.

Generics are desirable because they let you write code that is safer and easier to read than code that is littered with `Object` variables and casts. Generics are particularly useful for collection classes, such as the ubiquitous `ArrayList`.

Generics are—at least on the surface—similar to templates in C++. In C++, as in Java, templates were first added to the language to support strongly typed collections. However, over the years, other uses were discovered. After reading this chapter, perhaps you will find novel uses for Java generics in your programs.

### Why Generic Programming?

*Generic programming* means to write code that can be reused for objects of many different types. For example, you don't want to program separate classes to collect `String` and `File` objects. And you don't have to—the single class `ArrayList` collects objects of any class. This is one example of generic programming.

Before Java SE 5.0, generic programming in Java was always achieved with *inheritance*. The `ArrayList` class simply maintained an array of `Object` references:

```
public class ArrayList // before Java SE 5.0
{
    public Object get(int i) { . . . }
    public void add(Object o) { . . . }
    . . .
    private Object[] elementData;
}
```

This approach has two problems. A cast is necessary whenever you retrieve a value:

```
ArrayList files = new ArrayList();
. . .
String filename = (String) names.get(0);
```

Moreover, there is no error checking. You can add values of any class:

```
files.add(new File(" . . ."));
```

This call compiles and runs without error. Elsewhere, casting the result of `get` to a `String` will cause an error.

Generics offer a better solution: *type parameters*. The `ArrayList` class now has a type parameter that indicates the element type:

```
ArrayList<String> files = new ArrayList<String>();
```

This makes your code easier to read. You can tell right away that this particular array list contains `String` objects.

The compiler can make good use of this information too. No cast is required for calling `get`. The compiler knows that the return type is `String`, not `Object`:

```
String filename = files.get(0);
```

The compiler also knows that the `add` method of an `ArrayList<String>` has a parameter of type `String`. That is a lot safer than having an `Object` parameter. Now the compiler can check that you don't insert objects of the wrong type. For example, the statement

```
files.add(new File(" . . .")); // can only add String objects to an ArrayList<String>
```

will not compile. A compiler error is much better than a class cast exception at runtime. This is the appeal of type parameters: they make your programs easier to read and safer.

### **Who Wants to Be a Generic Programmer?**

It is easy to use a generic class such as `ArrayList`. Most Java programmers will simply use types such as `ArrayList<String>` as if they had been built into the language, just like `String[]` arrays. (Of course, array lists are better than arrays because they can expand automatically.)

However, it is not so easy to implement a generic class. The programmers who use your code will want to plug in all sorts of classes for your type parameters. They expect everything to work without onerous restrictions and confusing error messages. Your job as a generic programmer, therefore, is to anticipate all the potential future uses of your class.

How hard can this get? Here is a typical issue that the designers of the standard class library had to grapple with. The `ArrayList` class has a method `addAll` to add all elements of another collection. A programmer may want to add all elements from an `ArrayList<Manager>` to an `ArrayList<Employee>`. But, of course, doing it the other way around should not be legal. How do you allow one call and disallow the other? The Java language designers invented an ingenious new concept, the *wildcard type*, to solve this problem. Wildcard types are rather abstract, but they allow a library builder to make methods as flexible as possible.

Generic programming falls into three skill levels. At a basic level, you just use generic classes—typically, collections such as `ArrayList`—without thinking how and why they work. Most application programmers will want to stay at that level until something goes wrong. You may encounter a confusing error message when mixing different generic classes, or when interfacing with legacy code that knows nothing about type parameters. At that point, you need to learn enough about Java generics to solve problems systematically rather than through random tinkering. Finally, of course, you may want to implement your own generic classes and methods.

Application programmers probably won't write lots of generic code. The folks at Sun have already done the heavy lifting and supplied type parameters for all the collection classes. As a rule of thumb, only code that traditionally involved lots of casts from very general types (such as `Object` or the `Comparable` interface) will benefit from using type parameters.

In this chapter, we tell you everything you need to know to implement your own generic code. However, we expect most readers to use this knowledge primarily for help with troubleshooting, and to satisfy their curiosity about the inner workings of the parameterized collection classes.

### Definition of a Simple Generic Class

A *generic class* is a class with one or more type variables. In this chapter, we use a simple `Pair` class as an example. This class allows us to focus on generics without being distracted by data storage details. Here is the code for the `genericPair` class:

```
public class Pair<T>
{
    public Pair() { first = null; second = null; }
    public Pair(T first, T second) { this.first = first; this.second = second; }

    public T getFirst() { return first; }
    public T getSecond() { return second; }

    public void setFirst(T newValue) { first = newValue; }
    public void setSecond(T newValue) { second = newValue; }

    private T first;
    private T second;
}
```

The `Pair` class introduces a type variable `T`, enclosed in angle brackets `< >`, after the class name. A generic class can have more than one type variable. For example, we could have defined the `Pair` class with separate types for the first and second field:

```
public class Pair<T, U> { . . . }
```

The type variables are used throughout the class definition to specify method return types and the types of fields and local variables. For example:

```
private T first; // uses type variable
```



**NOTE:** It is common practice to use uppercase letters for type variables, and to keep them short. The Java library uses the variable `E` for the element type of a collection, `K` and `V` for key and value types of a table, and `T` (and the neighboring letters `U` and `S`, if necessary) for “any type at all”.

You *instantiate* the generic type by substituting types for the type variables, such as

```
Pair<String>
```

You can think of the result as an ordinary class with constructors

```
Pair<String>()
Pair<String>(String, String)
```

and methods

```
String getFirst()
String getSecond()
void setFirst(String)
void setSecond(String)
```

In other words, the generic class acts as a factory for ordinary classes.

The program in Listing 12-1 puts the `Pair` class to work. The static `minmax` method traverses an array and simultaneously computes the minimum and maximum value. It uses a `Pair` object to return both results. Recall that the `compareTo` method compares two



strings, returning 0 if the strings are identical, a negative integer if the first string comes before the second in dictionary order, and a positive integer otherwise.



**C++ NOTE:** Superficially, generic classes in Java are similar to template classes in C++. The only obvious difference is that Java has no special template keyword. However, as you will see throughout this chapter, there are substantial differences between these two mechanisms.

**Listing 12-1** PairTest1.java

```
1. /**
2.  * @version 1.00 2004-05-10
3.  * @author Cay Horstmann
4.  */
5. public class PairTest1
6. {
7.     public static void main(String[] args)
8.     {
9.         String[] words = { "Mary", "had", "a", "little", "lamb" };
10.        Pair<String> mm = ArrayAlg.minmax(words);
11.        System.out.println("min = " + mm.getFirst());
12.        System.out.println("max = " + mm.getSecond());
13.    }
14. }
15.
16. class ArrayAlg
17. {
18.     /**
19.      * Gets the minimum and maximum of an array of strings.
20.      * @param a an array of strings
21.      * @return a pair with the min and max value, or null if a is null or empty
22.      */
23.     public static Pair<String> minmax(String[] a)
24.     {
25.         if (a == null || a.length == 0) return null;
26.         String min = a[0];
27.         String max = a[0];
28.         for (int i = 1; i < a.length; i++)
29.         {
30.             if (min.compareTo(a[i]) > 0) min = a[i];
31.             if (max.compareTo(a[i]) < 0) max = a[i];
32.         }
33.         return new Pair<String>(min, max);
34.     }
35. }
```

## Generic Methods

In the preceding section, you have seen how to define a generic class. You can also define a single method with type parameters.

```
class ArrayAlg
{
    public static <T> T getMiddle(T[] a)
    {
        return a[a.length / 2];
    }
}
```

This method is defined inside an ordinary class, not inside a generic class. However, it is a generic method, as you can see from the angle brackets and the type variable. Note that the type variables are inserted after the modifiers (`public static`, in our case) and before the return type.

You can define generic methods both inside ordinary classes and inside generic classes.

When you call a generic method, you can place the actual types, enclosed in angle brackets, before the method name:

```
String[] names = { "John", "Q.", "Public" };
String middle = ArrayAlg.<String>getMiddle(names);
```

In this case (and indeed in most cases), you can omit the `<String>` type parameter from the method call. The compiler has enough information to infer the method that you want. It matches the type of `names` (that is, `String[]`) against the generic type `T[]` and deduces that `T` must be `String`. That is, you can simply call

```
String middle = ArrayAlg.getMiddle(names);
```

In almost all cases, type inference for generic methods works smoothly. Occasionally, the compiler gets it wrong, and you'll need to decipher an error report. Consider this example:

```
double middle = ArrayAlg.getMiddle(3.14, 1729, 0);
```

The error message is: “found: java.lang.Number&java.lang.Comparable<? extends java.lang.Number&java.lang.Comparable<?>>, required: double”. You will learn later in this chapter how to decipher the “found” type declaration. In a nutshell, the compiler autoboxed the parameters into a `Double` and two `Integer` objects, and then it tried to find a common supertype of these classes. It actually found two: `Number` and the `Comparable` interface, which is itself a generic type. In this case, the remedy is to write all parameters as `double` values.



**TIP:** Peter von der Ahé recommends this trick if you want to see which type the compiler infers for a generic method call: Purposefully introduce an error and study the resulting error message. For example, consider the call `ArrayAlg.getMiddle("Hello", 0, null)`. Assign the result to a `JButton`, which can't possibly be right. You will get an error report “found: java.lang.Object&java.io.Serializable&java.lang.Comparable<? extends java.lang.Object&java.io.Serializable&java.lang.Comparable<?>>”

In plain English, you can assign the result to `Object`, `Serializable`, or `Comparable`.

**C++** NOTE: In C++, you place the type parameters after the method name. That can lead to nasty parsing ambiguities. For example, `g(f<a,b>(c))` can mean “call `g` with the result of `f<a,b>(c)`”, or “call `g` with the two boolean values `f<a` and `b>(c)`”.

### Bounds for Type Variables

Sometimes, a class or a method needs to place restrictions on type variables. Here is a typical example. We want to compute the smallest element of an array:

```
class ArrayAlg
{
    public static <T> T min(T[] a) // almost correct
    {
        if (a == null || a.length == 0) return null;
        T smallest = a[0];
        for (int i = 1; i < a.length; i++)
            if (smallest.compareTo(a[i]) > 0) smallest = a[i];
        return smallest;
    }
}
```

But there is a problem. Look inside the code of the `min` method. The variable `smallest` has type `T`, which means that it could be an object of an arbitrary class. How do we know that the class to which `T` belongs has a `compareTo` method?

The solution is to restrict `T` to a class that implements the `Comparable` interface—a standard interface with a single method, `compareTo`. You achieve this by giving a *bound* for the type variable `T`:

```
public static <T extends Comparable> T min(T[] a) . . .
```

Actually, the `Comparable` interface is itself a generic type. For now, we will ignore that complexity and the warnings that the compiler generates. “Wildcard Types” on page 632 discusses how to properly use type parameters with the `Comparable` interface.

Now, the generic `min` method can only be called with arrays of classes that implement the `Comparable` interface, such as `String`, `Date`, and so on. Calling `min` with a `Rectangle` array is a compile-time error because the `Rectangle` class does not implement `Comparable`.

**C++** NOTE: In C++, you cannot restrict the types of template parameters. If a programmer instantiates a template with an inappropriate type, an (often obscure) error message is reported inside the template code.

You may wonder why you use the `extends` keyword rather than the `implements` keyword in this situation—after all, `Comparable` is an interface. The notation

```
<T extends BoundingType>
```

expresses that `T` should be a *subtype* of the bounding type. Both `T` and the bounding type can be either a class or an interface. The `extends` keyword was chosen because it is a reasonable approximation of the subtype concept, and the Java designers did not want to add a new keyword (such as `sub`) to the language.

A type variable or wildcard can have multiple bounds. For example:

```
T extends Comparable & Serializable
```

The bounding types are separated by ampersands (&) because commas are used to separate type variables.

As with Java inheritance, you can have as many interface supertypes as you like, but at most one of the bounds can be a class. If you have a class as a bound, it must be the first one in the bounds list.

In the next sample program (Listing 12-2), we rewrite the `minmax` method to be generic. The method computes the minimum and maximum of a generic array, returning a `Pair<T>`.

**Listing 12-2** PairTest2.java

```

1. import java.util.*;
2.
3. /**
4.  * @version 1.00 2004-05-10
5.  * @author Cay Horstmann
6.  */
7. public class PairTest2
8. {
9.     public static void main(String[] args)
10.    {
11.        GregorianCalendar[] birthdays =
12.            {
13.                new GregorianCalendar(1906, Calendar.DECEMBER, 9), // G. Hopper
14.                new GregorianCalendar(1815, Calendar.DECEMBER, 10), // A. Lovelace
15.                new GregorianCalendar(1903, Calendar.DECEMBER, 3), // J. von Neumann
16.                new GregorianCalendar(1910, Calendar.JUNE, 22), // K. Zuse
17.            };
18.        Pair<GregorianCalendar> mm = ArrayAlg.minmax(birthdays);
19.        System.out.println("min = " + mm.getFirst().getTime());
20.        System.out.println("max = " + mm.getSecond().getTime());
21.    }
22. }
23.
24. class ArrayAlg
25. {
26.     /**
27.      * Gets the minimum and maximum of an array of objects of type T.
28.      * @param a an array of objects of type T
29.      * @return a pair with the min and max value, or null if a is
30.      *         null or empty
31.      */
32.     public static <T extends Comparable> Pair<T> minmax(T[] a)
33.     {
34.         if (a == null || a.length == 0) return null;

```

**Listing 12-2** PairTest2.java (continued)

```

35.     T min = a[0];
36.     T max = a[0];
37.     for (int i = 1; i < a.length; i++)
38.     {
39.         if (min.compareTo(a[i]) > 0) min = a[i];
40.         if (max.compareTo(a[i]) < 0) max = a[i];
41.     }
42.     return new Pair<T>(min, max);
43. }
44. }
```

### Generic Code and the Virtual Machine

The virtual machine does not have objects of generic types—all objects belong to ordinary classes. An earlier version of the generics implementation was even able to compile a program that uses generics into class files that executed on 1.0 virtual machines! This backward compatibility was only abandoned fairly late in the development for Java generics. If you use the Sun compiler to compile code that uses Java generics, the resulting class files will *not* execute on pre-5.0 virtual machines.



**NOTE:** If you want to have the benefits of generics while retaining bytecode compatibility with older virtual machines, check out <http://sourceforge.net/projects/retroweaver>. The Retroweaver program rewrites class files so that they are compatible with older virtual machines.

Whenever you define a generic type, a corresponding *raw* type is automatically provided. The name of the raw type is simply the name of the generic type, with the type parameters removed. The type variables are *erased* and replaced by their bounding types (or `Object` for variables without bounds.)

For example, the raw type for `Pair<T>` looks like this:

```

public class Pair
{
    public Pair(Object first, Object second)
    {
        this.first = first;
        this.second = second;
    }

    public Object getFirst() { return first; }
    public Object getSecond() { return second; }

    public void setFirst(Object newValue) { first = newValue; }
    public void setSecond(Object newValue) { second = newValue; }

    private Object first;
    private Object second;
}
```

Because `T` is an unbounded type variable, it is simply replaced by `Object`.

The result is an ordinary class, just as you might have implemented it before generics were added to the Java programming language.

Your programs may contain different kinds of `Pair`, such as `Pair<String>` or `Pair<GregorianCalendar>`, but erasure turns them all into raw `Pair` types.



**C++ NOTE:** In this regard, Java generics are very different from C++ templates. C++ produces different types for each template instantiation, a phenomenon called “template code bloat.” Java does not suffer from this problem.

The raw type replaces type variables with the first bound, or `Object` if no bounds are given. For example, the type variable in the class `Pair<T>` has no explicit bounds, hence the raw type replaces `T` with `Object`. Suppose we declare a slightly different type:

```
public class Interval<T extends Comparable & Serializable> implements Serializable
{
    public Interval(T first, T second)
    {
        if (first.compareTo(second) <= 0) { lower = first; upper = second; }
        else { lower = second; upper = first; }
    }
    . . .
    private T lower;
    private T upper;
}
```

The raw type `Interval` looks like this:

```
public class Interval implements Serializable
{
    public Interval(Comparable first, Comparable second) { . . . }
    . . .
    private Comparable lower;
    private Comparable upper;
}
```



**NOTE:** You may wonder what happens if you switch the bounds: `class Interval<Serializable & Comparable>`. In that case, the raw type replaces `T` with `Serializable`, and the compiler inserts casts to `Comparable` when necessary. For efficiency, you should therefore put tagging interfaces (that is, interfaces without methods) at the end of the bounds list.

### Translating Generic Expressions

When you program a call to a generic method, the compiler inserts casts when the return type has been erased. For example, consider the sequence of statements

```
Pair<Employee> buddies = . . . ;
Employee buddy = buddies.getFirst();
```

The erasure of `getFirst` has return type `Object`. The compiler automatically inserts the cast to `Employee`. That is, the compiler translates the method call into two virtual machine instructions:

- A call to the raw method `Pair.getFirst`
- A cast of the returned `Object` to the `Employee` type

Casts are also inserted when you access a generic field. Suppose the first and second fields of the `Pair` class were public. (Not a good programming style, perhaps, but it is legal Java.) Then the expression

```
Employee buddy = buddies.first;
```

also has a cast inserted in the resulting byte codes.

### Translating Generic Methods

Type erasure also happens for generic methods. Programmers usually think of a generic method such as

```
public static <T extends Comparable> T min(T[] a)
```

as a whole family of methods, but after erasure, only a single method is left:

```
public static Comparable min(Comparable[] a)
```

Note that the type parameter `T` has been erased, leaving only its bounding type `Comparable`.

Erasure of method brings up a couple of complexities. Consider this example:

```
class DateInterval extends Pair<Date>
{
    public void setSecond(Date second)
    {
        if (second.compareTo(getFirst()) >= 0)
            super.setSecond(second);
    }
    . . .
}
```

A date interval is a pair of `Date` objects, and we'll want to override the methods to ensure that the second value is never smaller than the first. This class is erased to

```
class DateInterval extends Pair // after erasure
{
    public void setSecond(Date second) { . . . }
    . . .
}
```

Perhaps surprisingly, there is another `setSecond` method, inherited from `Pair`, namely,

```
public void setSecond(Object second)
```

This is clearly a different method because it has a parameter of a different type—`Object` instead of `Date`. But it *shouldn't* be different. Consider this sequence of statements:

```
DateInterval interval = new DateInterval(. . .);
Pair<Date> pair = interval; // OK--assignment to superclass
pair.setSecond(aDate);
```

Our expectation is that the call to `setSecond` is polymorphic and that the appropriate method is called. Because `pair` refers to a `DateInterval` object, that should be `DateInterval.setSecond`. The problem is that the type erasure interferes with polymorphism. To fix this problem, the compiler generates a *bridge method* in the `DateInterval` class:

```
public void setSecond(Object second) { setSecond((Date) second); }
```

To see why this works, let us carefully follow the execution of the statement

```
pair.setSecond(aDate)
```

The variable `pair` has declared type `Pair<Date>`, and that type only has a single method called `setSecond`, namely `setSecond(Object)`. The virtual machine calls that method on the object to which `pair` refers. That object is of type `DateInterval`. Therefore, the method `DateInterval.setSecond(Object)` is called. That method is the synthesized bridge method. It calls `DateInterval.setSecond(Date)`, which is what we want.

Bridge methods can get even stranger. Suppose the `DateInterval` method also overrides the `getSecond` method:

```
class DateInterval extends Pair<Date>
{
    public Date getSecond() { return (Date) super.getSecond().clone(); }
    . . .
}
```

In the erased type, there are two `getSecond` methods:

```
Date getSecond() // defined in DateInterval
Object getSecond() // defined in Pair
```

You could not write Java code like that—it would be illegal to have two methods with the same parameter types—here, no parameters. However, in the virtual machine, the parameter types *and the return type* specify a method. Therefore, the compiler can produce bytecodes for two methods that differ only in their return type, and the virtual machine will handle this situation correctly.



**NOTE:** Bridge methods are not limited to generic types. We already noted in Chapter 5 that, starting with Java SE 5.0, it is legal for a method to specify a more restrictive return type when overriding another method. For example:

```
public class Employee implements Cloneable
{
    public Employee clone() throws CloneNotSupportedException { ... }
}
```

The `Object.clone` and `Employee.clone` methods are said to have *covariant return types*.

Actually, the `Employee` class has *two* `clone` methods:

```
Employee clone() // defined above
Object clone() // synthesized bridge method, overrides Object.clone
```

The synthesized bridge method calls the newly defined method.

In summary, you need to remember these facts about translation of Java generics:

- There are no generics in the virtual machines, only ordinary classes and methods.
- All type parameters are replaced by their bounds.
- Bridge methods are synthesized to preserve polymorphism.
- Casts are inserted as necessary to preserve type safety.



### Calling Legacy Code

Lots of Java code was written before Java SE 5.0. If generic classes could not interoperate with that code, they would probably not be widely used. Fortunately, it is straightforward to use generic classes together with their raw equivalents in legacy APIs.

Let us look at a concrete example. To set the labels of a `JSlider`, you use the method

```
void setLabelTable(Dictionary table)
```

In Chapter 9, we used the following code to populate the label table:

```
Dictionary<Integer, Component> labelTable = new Hashtable<Integer, Component>();
labelTable.put(0, new JLabel(new ImageIcon("nine.gif")));
labelTable.put(20, new JLabel(new ImageIcon("ten.gif")));
. . .
slider.setLabelTable(labelTable); // WARNING
```

In Java SE 5.0, the `Dictionary` and `Hashtable` classes were turned into a generic class. Therefore, we are able to form `Dictionary<Integer, Component>` instead of using a raw `Dictionary`. However, when you pass the `Dictionary<Integer, Component>` object to `setLabelTable`, the compiler issues a warning.

```
Dictionary<Integer, Component> labelTable = . . . ;
slider.setLabelTable(labelTable); // WARNING
```

After all, the compiler has no assurance about what the `setLabelTable` might do to the `Dictionary` object. That method might replace all the keys with strings. That breaks the guarantee that the keys have type `Integer`, and future operations may cause bad cast exceptions.

There isn't much you can do with this warning, except ponder it and ask what the `JSlider` is likely going to do with this `Dictionary` object. In our case, it is pretty clear that the `JSlider` only reads the information, so we can ignore the warning.

Now consider the opposite case, in which you get an object of a raw type from a legacy class. You can assign it to a parameterized type variable, but of course you will get a warning. For example:

```
Dictionary<Integer, Components> labelTable = slider.getLabelTable(); // WARNING
```

That's ok—review the warning and make sure that the label table really contains `Integer` and `Component` objects. Of course, there never is an absolute guarantee. A malicious coder might have installed a different `Dictionary` in the slider. But again, the situation is no worse than it was before Java SE 5.0. In the worst case, your program will throw an exception.

After you are done pondering the warning, you can use an *annotation* to make it disappear. The annotation must be placed before the method whose code generates the warning, like this:

```
@SuppressWarnings("unchecked")
public void configureSlider() { . . . }
```

Unfortunately, this annotation turns off checking for all code inside the method. It is a good idea to isolate potentially unsafe code into separate methods so that they can be reviewed more easily.



NOTE: The `Hashtable` class is a concrete subclass of the abstract `Dictionary` class. Both `Dictionary` and `Hashtable` have been declared as “obsolete” ever since they were superseded by the `Map` interface and the `HashMap` class of Java SE 1.2. Apparently though, they are still alive and kicking. After all, the `JSlider` class was only added in Java SE 1.3. Didn't its programmers know about the `Map` class by then? Does this make you hopeful that they are going to adopt generics in the near future? Well, that's the way it goes with legacy code.

### Restrictions and Limitations

In the following sections, we discuss a number of restrictions that you need to consider when working with Java generics. Most of these restrictions are a consequence of type erasure.

#### **Type Parameters Cannot Be Instantiated with Primitive Types**

You cannot substitute a primitive type for a type parameter. Thus, there is no `Pair<double>`, only `Pair<Double>`. The reason is, of course, type erasure. After erasure, the `Pair` class has fields of type `Object`, and you can't use them to store `double` values.

This is an annoyance, to be sure, but it is consistent with the separate status of primitive types in the Java language. It is not a fatal flaw—there are only eight primitive types, and you can always handle them with separate classes and methods when wrapper types are not an acceptable substitute.

#### **Runtime Type Inquiry Only Works with Raw Types**

Objects in the virtual machine always have a specific nongeneric type. Therefore, all type inquiries yield only the raw type. For example,

```
if (a instanceof Pair<String>) // same as a instanceof Pair
```

really only tests whether `a` is a `Pair` of any type. The same is true for the test

```
if (a instanceof Pair<T>) // T is ignored
```

or the cast

```
Pair<String> p = (Pair<String>) a; // WARNING--can only test that a is a Pair
```

To remind you of the risk, you will get a compiler warning whenever you use `instanceof` or cast expressions that involve generic types.

In the same spirit, the `getClass` method always returns the raw type. For example:

```
Pair<String> stringPair = . . . ;
Pair<Employee> employeePair = . . . ;
if (stringPair.getClass() == employeePair.getClass()) // they are equal
```

The comparison yields `true` because both calls to `getClass` return `Pair.class`.

#### **You Cannot Throw or Catch Instances of a Generic Class**

You can neither throw nor catch objects of a generic class. In fact, it is not even legal for a generic class to extend `Throwable`. For example, the following definition will not compile:

```
public class Problem<T> extends Exception { /* . . . */ } // ERROR--can't extend Throwable
```

You cannot use a type variable in a catch clause. For example, the following method will not compile:

```

public static <T extends Throwable> void doWork(Class<T> t)
{
    try
    {
        do work
    }
    catch (T e) // ERROR--can't catch type variable
    {
        Logger.global.info(...)
    }
}

```

However, it is ok to use type variables in exception specifications. The following method is legal:

```

public static <T extends Throwable> void doWork(T t) throws T // OK
{
    try
    {
        do work
    }
    catch (Throwable realCause)
    {
        t.initCause(realCause);
        throw t;
    }
}

```

### **Arrays of Parameterized Types Are Not Legal**

You cannot declare arrays of parameterized types, such as

```
Pair<String>[] table = new Pair<String>[10]; // ERROR
```

What's wrong with that? After erasure, the type of `table` is `Pair[]`. You can convert it to `Object[]`:

```
Object[] objarray = table;
```

An array remembers its component type and throws an `ArrayStoreException` if you try to store an element of the wrong type:

```
objarray[0] = "Hello"; // ERROR--component type is Pair
```

But erasure renders this mechanism ineffective for generic types. The assignment

```
objarray[0] = new Pair<Employee>();
```

would pass the array store check but still result in a type error. For this reason, arrays of parameterized types are outlawed.



**TIP:** If you need to collect parameterized type objects, simply use an `ArrayList`: `ArrayList<Pair<String>>` is safe and effective.

**You Cannot Instantiate Type Variables**

You cannot use type variables in expression such as `new T(...)`, `new T[...]`, or `T.class`. For example, the following `Pair<T>` constructor is illegal:

```
public Pair() { first = new T(); second = new T(); } // ERROR
```

Type erasure would change `T` to `Object`, and surely you don't want to call `new Object()`.

As a workaround, you can construct generic objects through reflection, by calling the `Class.newInstance` method.

Unfortunately, the details are a bit complex. You cannot call

```
first = T.class.newInstance(); // ERROR
```

The expression `T.class` is not legal. Instead, you must design the API so that you are handed a `Class` object, like this:

```
public static <T> Pair<T> makePair(Class<T> cl)
{
    try { return new Pair<T>(cl.newInstance(), cl.newInstance()); }
    catch (Exception ex) { return null; }
}
```

This method could be called as follows:

```
Pair<String> p = Pair.makePair(String.class);
```

Note that the `Class` class is itself generic. For example, `String.class` is an instance (indeed, the sole instance) of `Class<String>`. Therefore, the `makePair` method can infer the type of the pair that it is making.

You cannot construct a generic array:

```
public static <T extends Comparable> T[] minmax(T[] a) { T[] mm = new T[2]; . . . } // ERROR
```

Type erasure would cause this method to always construct an `arrayObject[2]`.

If the array is only used as a private instance field of a class, you can declare the array as `Object[]` and use casts when retrieving elements. For example, the `ArrayList` class could be implemented as follows:

```
public class ArrayList<E>
{
    private Object[] elements;
    @SuppressWarnings("unchecked") public E get(int n) { return (E) elements[n]; }
    public void set(int n, E e) { elements[n] = e; } // no cast needed
    . . .
}
```

The actual implementation is not quite as clean:

```
public class ArrayList<E>
{
    private E[] elements;
    public ArrayList() { elements = (E[]) new Object[10]; }
    . . .
}
```

Here, the cast `E[]` is an outright lie, but type erasure makes it undetectable.

This technique does not work for our `minmax` method since we are returning a `T[]` array, and a runtime error results if we lie about its type. Suppose we implement

```
public static <T extends Comparable> T[] minmax(T[] a)
{
    Object[] mm = new Object[2];
    . . . ;
    return (T[]) mm; // compiles with warning
}
```

The call

```
String[] ss = minmax("Tom", "Dick", "Harry");
```

compiles without any warning. A `ClassCastException` occurs when the `Object[]` reference is assigned to the `String[]` variable.

In this situation, you can use reflection and call `Array.newInstance`:

```
public static <T extends Comparable> T[] minmax(T[] a)
{
    T[] mm = (T[]) Array.newInstance(a.getClass().getComponentType(), 2);
    . . .
}
```

The `toArray` method of the `ArrayList` class is not so lucky. It needs to produce a `T[]` array, but it doesn't have the component type. Therefore, there are two variants:

```
Object[] toArray()
T[] toArray(T[] result)
```

The second method receives an array parameter. If the array is large enough, it is used. Otherwise, a new array of sufficient size is created, using the component type of `result`.

### **Type Variables Are Not Valid in Static Contexts of Generic Classes**

You cannot reference type variables in static fields or methods. For example, the following clever idea won't work:

```
public class Singleton<T>
{
    public static T getInstance() // ERROR
    {
        if (singleInstance == null) construct new instance of T
            return singleInstance;
    }
    private static T singleInstance; // ERROR
}
```

If this could be done, then a program could declare a `Singleton<Random>` to share a random number generator and a `Singleton<JFileChooser>` to share a file chooser dialog. But it can't work. After type erasure there is only one `Singleton` class, and only one `singleInstance` field. For that reason, static fields and methods with type variables are simply outlawed.

### **Beware of Clashes After Erasure**

It is illegal to create conditions that cause clashes when generic types are erased. Here is an example. Suppose we add an `equals` method to the `Pair` class, like this:

```
public class Pair<T>
{
    public boolean equals(T value) { return first.equals(value) && second.equals(value); }
    . . .
}
```

Consider a `Pair<String>`. Conceptually, it has two `equals` methods:

```
boolean equals(String) // defined in Pair<T>
boolean equals(Object) // inherited from Object
```

But the intuition leads us astray. The erasure of the method

```
boolean equals(T)
```

is

```
boolean equals(Object)
```

which clashes with the `Object.equals` method.

The remedy is, of course, to rename the offending method.

The generics specification cites another rule: “To support translation by erasure, we impose the restriction that a class or type variable may not at the same time be a subtype of two interface types which are different parameterizations of the same interface.” For example, the following is illegal:

```
class Calendar implements Comparable<Calendar> { . . . }
class GregorianCalendar extends Calendar implements Comparable<GregorianCalendar>
{ . . . } // ERROR
```

`GregorianCalendar` would then implement both `Comparable<Calendar>` and `Comparable<GregorianCalendar>`, which are different parameterizations of the same interface.

It is not obvious what this restriction has to do with type erasure. After all, the non-generic version

```
class Calendar implements Comparable { . . . }
class GregorianCalendar extends Calendar implements Comparable { . . . }
```

is legal. The reason is far more subtle. There would be a conflict with the synthesized bridge methods. A class that implements `Comparable<X>` gets a bridge method

```
public int compareTo(Object other) { return compareTo((X) other); }
```

You cannot have two such methods for different types `X`.

### Inheritance Rules for Generic Types

When you work with generic classes, you need to learn a few rules about inheritance and subtypes. Let’s start with a situation that many programmers find unintuitive. Consider a class and a subclass, such as `Employee` and `Manager`. Is `Pair<Manager>` a subclass of `Pair<Employee>`? Perhaps surprisingly, the answer is “no.” For example, the following code will not compile:

```
Manager[] topHonchos = . . . ;
Pair<Employee> result = ArrayAlg.minmax(topHonchos); // ERROR
```

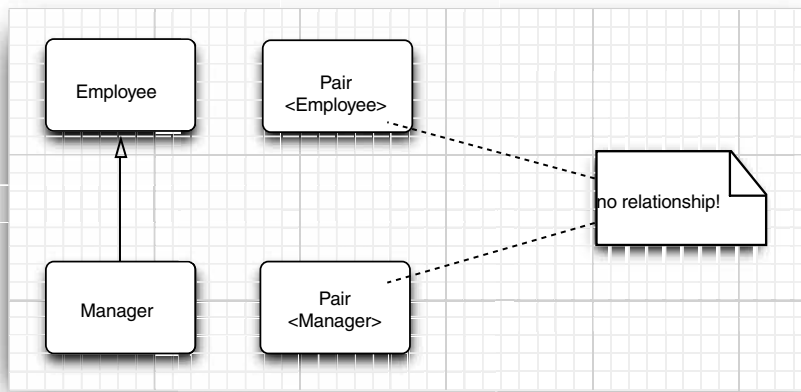
The `minmax` method returns a `Pair<Manager>`, not a `Pair<Employee>`, and it is illegal to assign one to the other.

In general, there is *no* relationship between `Pair<S>` and `Pair<T>`, no matter how `S` and `T` are related (see Figure 12–1).

This seems like a cruel restriction, but it is necessary for type safety. Suppose we were allowed to convert a `Pair<Manager>` to a `Pair<Employee>`. Consider this code:

```
Pair<Manager> managerBuddies = new Pair<Manager>(ceo, cfo);
Pair<Employee> employeeBuddies = managerBuddies; // illegal, but suppose it wasn't
employeeBuddies.setFirst(lowlyEmployee);
```

Clearly, the last statement is legal. But `employeeBuddies` and `managerBuddies` refer to the *same object*. We now managed to pair up the CFO with a lowly employee, which should not be possible for a `Pair<Manager>`.



**Figure 12-1** No inheritance relationship between pair classes



**NOTE:** You just saw an important difference between generic types and Java arrays. You can assign a `Manager[]` array to a variable of type `Employee[]`:

```
Manager[] managerBuddies = { ceo, cfo };
Employee[] employeeBuddies = managerBuddies; // OK
```

However, arrays come with special protection. If you try to store a lowly employee into `employeeBuddies[0]`, the virtual machine throws an `ArrayStoreException`.

You can always convert a parameterized type to a raw type. For example, `Pair<Employee>` is a subtype of the raw type `Pair`. This conversion is necessary for interfacing with legacy code.

Can you convert to the raw type and then cause a type error? Unfortunately, you can. Consider this example:

```
Pair<Manager> managerBuddies = new Pair<Manager>(ceo, cfo);
Pair rawBuddies = managerBuddies; // OK
rawBuddies.setFirst(new File(" . . ")); // only a compile-time warning
```

This sounds scary. However, keep in mind that you are no worse off than you were with older versions of Java. The security of the virtual machine is not at stake. When the

foreign object is retrieved with `getFirst` and assigned to a `Manager` variable, a `ClassCastException` is thrown, just as in the good old days. You merely lose the added safety that generic programming normally provides.

Finally, generic classes can extend or implement other generic classes. In this regard, they are no different from ordinary classes. For example, the class `ArrayList<T>` implements the interface `List<T>`. That means, an `ArrayList<Manager>` can be converted to a `List<Manager>`. However, as you just saw, an `ArrayList<Manager>` is *not* an `ArrayList<Employee>` or `List<Employee>`. Figure 12-2 shows these relationships.

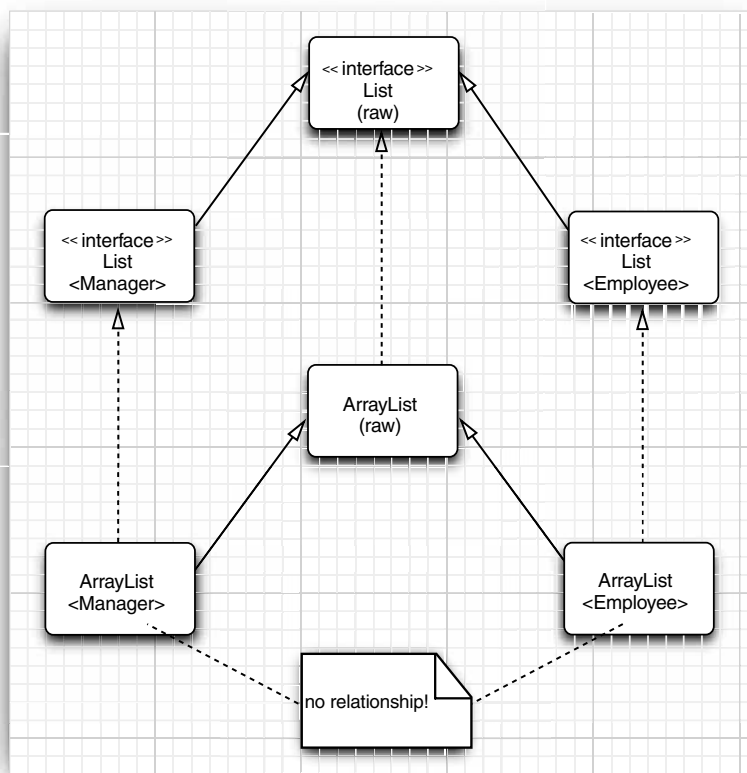


Figure 12-2 Subtype relationships among generic list types

### Wildcard Types

It was known for some time among researchers of type systems that a rigid system of generic types is quite unpleasant to use. The Java designers invented an ingenious (but nevertheless safe) “escape hatch”: the *wildcard type*. For example, the wildcard type

```
Pair<? extends Employee>
```



denotes any generic `Pair` type whose type parameter is a subclass of `Employee`, such as `Pair<Manager>`, but not `Pair<String>`.

Let's say you want to write a method that prints out pairs of employees, like this:

```
public static void printBuddies(Pair<Employee> p)
{
    Employee first = p.getFirst();
    Employee second = p.getSecond();
    System.out.println(first.getName() + " and " + second.getName() + " are buddies.");
}
```

As you saw in the preceding section, you cannot pass a `Pair<Manager>` to that method, which is rather limiting. But the solution is simple—use a wildcard type:

```
public static void printBuddies(Pair<? extends Employee> p)
```

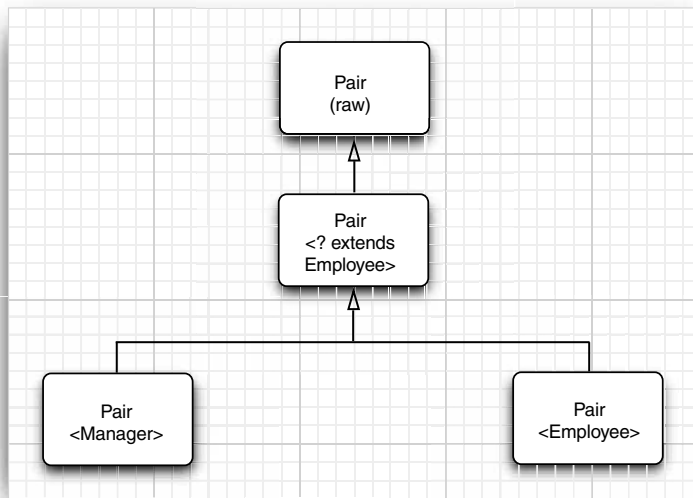
The type `Pair<Manager>` is a subtype of `Pair<? extends Employee>` (see Figure 12–3).

Can we use wildcards to corrupt a `Pair<Manager>` through a `Pair<? extends Employee>` reference?

```
Pair<Manager> managerBuddies = new Pair<Manager>(ceo, cfo);
Pair<? extends Employee> wildcardBuddies = managerBuddies; // OK
wildcardBuddies.setFirst(lowlyEmployee); // compile-time error
```

No corruption is possible. The call to `setFirst` is a type error. To see why, let us have a closer look at the type `Pair<? extends Employee>`. Its methods look like this:

```
? extends Employee getFirst()
void setFirst(? extends Employee)
```



**Figure 12–3** Subtype relationships with wildcards

This makes it impossible to call the `setFirst` method. The compiler only knows that it needs some subtype of `Employee`, but it doesn't know which type. It refuses to pass any specific type—after all, `?` might not match it.

We don't have this problem with `getFirst`: It is perfectly legal to assign the return value of `getFirst` to an `Employee` reference.

This is the key idea behind bounded wildcards. We now have a way of distinguishing between the safe accessor methods and the unsafe mutator methods.

### **Supertype Bounds for Wildcards**

Wildcard bounds are similar to type variable bounds, but they have an added capability—you can specify a *supertype bound*, like this:

```
? super Manager
```

This wildcard is restricted to all supertypes of `Manager`. (It was a stroke of good luck that the existing `super` keyword describes the relationship so accurately.)

Why would you want to do this? A wildcard with a supertype bound gives you the opposite behavior of the wildcards described in the section “Wildcard Types” on page 632. You can supply parameters to methods, but you can't use the return values. For example, `Pair<? super Manager>` has methods

```
void setFirst(? super Manager)
? super Manager getFirst()
```

The compiler doesn't know the exact type of the `setFirst` method but can call it with any object of type `Manager`, `Employee`, or `Object`, but not a subtype such as `Executive`. However, if you call `getFirst`, there is no guarantee about the type of the returned object. You can only assign it to an `Object`.

Here is a typical example. We have an array of managers and want to put the manager with the lowest and highest bonus into a `Pair` object. What kind of `Pair`? A `Pair<Employee>` should be fair game or, for that matter, a `Pair<Object>` (see Figure 12-4). The following method will accept any appropriate `Pair`:

```
public static void minmaxBonus(Manager[] a, Pair<? super Manager> result)
{
    if (a == null || a.length == 0) return;
    Manager min = a[0];
    Manager max = a[0];
    for (int i = 1; i < a.length; i++)
    {
        if (min.getBonus() > a[i].getBonus()) min = a[i];
        if (max.getBonus() < a[i].getBonus()) max = a[i];
    }
    result.setFirst(min);
    result.setSecond(max);
}
```

Intuitively speaking, wildcards with supertype bounds let you write to a generic object, wildcards with subtype bounds let you read from a generic objects.

Here is another use for supertype bounds. The `Comparable` interface is itself a generic type. It is declared as follows:

```
public interface Comparable<T>
{
    public int compareTo(T other);
}
```

Here, the type variable indicates the type of the other parameter. For example, the `String` class implements `Comparable<String>`, and its `compareTo` method is declared as

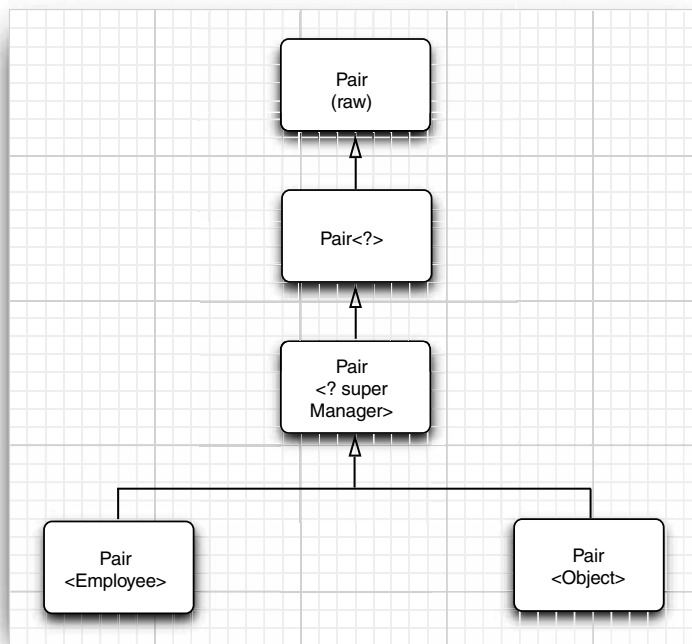
```
public int compareTo(String other)
```

This is nice—the explicit parameter has the correct type. Before Java SE 5.0, `other` was an `Object`, and a cast was necessary in the implementation of the method.

Because `Comparable` is a generic type, perhaps we should have done a better job with the `min` method of the `ArrayAlg` class? We could have declared it as

```
public static <T extends Comparable<T>> T min(T[] a)
```

This looks more thorough than just using `T extends Comparable`, and it would work fine for many classes. For example, if you compute the minimum of a `String` array, then `T` is the type `String`, and `String` is a subtype of `Comparable<String>`. But we run into a problem when processing an array of `GregorianCalendar` objects. As it happens, `GregorianCalendar` is a subclass of `Calendar`, and `Calendar` implements `Comparable<Calendar>`. Thus, `GregorianCalendar` implements `Comparable<Calendar>` but not `Comparable<GregorianCalendar>`.



**Figure 12-4** A wildcard with a supertype bound

In a situation such as this one, supertypes come to the rescue:

```
public static <T extends Comparable<? super T>> T min(T[] a) . . .
```

Now the `compareTo` method has the form

```
int compareTo(? super T)
```

Maybe it is declared to take an object of type `T`, or—for example, when `T` is `GregorianCalendar`—a supertype of `T`. At any rate, it is safe to pass an object of type `T` to the `compareTo` method.

To the uninitiated, a declaration such as `<T extends Comparable<? super T>>` is bound to look intimidating. This is unfortunate, because the intent of this declaration is to help application programmers by removing unnecessary restrictions on the call parameters. Application programmers with no interest in generics will probably learn quickly to gloss over these declarations and just take for granted that library programmers will do the right thing. If you are a library programmer, you'll need to get used to wildcards, or your users will curse you and throw random casts at their code until it compiles.

### **Unbounded Wildcards**

You can even use wildcards with no bounds at all, for example, `Pair<?>`. At first glance, this looks identical to the raw `Pair` type. Actually, the types are very different. The type `Pair<?>` has methods such as

```
? getFirst()
void setFirst(?)
```

The return value of `getFirst` can only be assigned to an `Object`. The `setFirst` method can never be called, *not even with an* `Object`. That's the essential difference between `Pair<?>` and `Pair`: you can call the `setObject` method of the raw `Pair` class with *any* `Object`.

Why would you ever want such a wimpy type? It is useful for very simple operations. For example, the following method tests whether a pair contains a given object. It never needs the actual type.

```
public static boolean hasNulls(Pair<?> p)
{
    return p.getFirst() == null || p.getSecond() == null;
}
```

You could have avoided the wildcard type by turning `contains` into a generic method:

```
public static <T> boolean hasNulls(Pair<T> p)
```

However, the version with the wildcard type seems easier to read.

### **Wildcard Capture**

Let us write a method that swaps the elements of a pair:

```
public static void swap(Pair<?> p)
```

A wildcard is not a type variable, so we can't write code that uses `?` as a type. In other words, the following would be illegal:

```
? t = p.getFirst(); // ERROR
p.setFirst(p.getSecond());
p.setSecond(t);
```

That's a problem because we need to temporarily hold the first element when we do the swapping. Fortunately, there is an interesting solution to this problem. We can write a helper method, `swapHelper`, like this:

```

public static <T> void swapHelper(Pair<T> p)
{
    T t = p.getFirst();
    p.setFirst(p.getSecond());
    p.setSecond(t);
}

```

Note that `swapHelper` is a generic method, whereas `swap` is not—it has a fixed parameter of type `Pair<?>`.

Now we can call `swapHelper` from `swap`:

```

public static void swap(Pair<?> p) { swapHelper(p); }

```

In this case, the parameter `T` of the `swapHelper` method *captures the wildcard*. It isn't known what type the wildcard denotes, but it is a definite type, and the definition of `<T>swapHelper` makes perfect sense when `T` denotes that type.

Of course, in this case, we were not compelled to use a wildcard. We could have directly implemented `<T> void swap(Pair<T> p)` as a generic method without wildcards. However, consider this example in which a wildcard type occurs naturally in the middle of a computation:

```

public static void maxminBonus(Manager[] a, Pair<? super Manager> result)
{
    minmaxBonus(a, result);
    PairAlg.swapHelper(result); // OK--swapHelper captures wildcard type
}

```

Here, the wildcard capture mechanism cannot be avoided.

Wildcard capture is only legal in very limited circumstances. The compiler must be able to guarantee that the wildcard represents a single, definite type. For example, the `T` in `ArrayList<Pair<T>>` can never capture the wildcard in `ArrayList<Pair<?>>`. The array list might hold two `Pair<?>`, each of which has a different type for `?`.

The test program in Listing 12–3 gathers up the various methods that we discussed in the preceding sections, so that you can see them in context.

### Listing 12–3 PairTest3.java

```

1. import java.util.*;
2.
3. /**
4.  * @version 1.00 2004-05-10
5.  * @author Cay Horstmann
6.  */
7. public class PairTest3
8. {
9.     public static void main(String[] args)
10.    {
11.        Manager ceo = new Manager("Gus Greedy", 800000, 2003, 12, 15);
12.        Manager cfo = new Manager("Sid Sneaky", 600000, 2003, 12, 15);
13.        Pair<Manager> buddies = new Pair<Manager>(ceo, cfo);
14.        printBuddies(buddies);

```

**Listing 12-3** PairTest3.java (continued)

```

15.
16.     ceo.setBonus(1000000);
17.     cfo.setBonus(500000);
18.     Manager[] managers = { ceo, cfo };
19.
20.     Pair<Employee> result = new Pair<Employee>();
21.     minmaxBonus(managers, result);
22.     System.out.println("first: " + result.getFirst().getName()
23.         + ", second: " + result.getSecond().getName());
24.     maxminBonus(managers, result);
25.     System.out.println("first: " + result.getFirst().getName()
26.         + ", second: " + result.getSecond().getName());
27. }
28.
29. public static void printBuddies(Pair<? extends Employee> p)
30. {
31.     Employee first = p.getFirst();
32.     Employee second = p.getSecond();
33.     System.out.println(first.getName() + " and " + second.getName() + " are buddies.");
34. }
35.
36. public static void minmaxBonus(Manager[] a, Pair<? super Manager> result)
37. {
38.     if (a == null || a.length == 0) return;
39.     Manager min = a[0];
40.     Manager max = a[0];
41.     for (int i = 1; i < a.length; i++)
42.     {
43.         if (min.getBonus() > a[i].getBonus()) min = a[i];
44.         if (max.getBonus() < a[i].getBonus()) max = a[i];
45.     }
46.     result.setFirst(min);
47.     result.setSecond(max);
48. }
49.
50. public static void maxminBonus(Manager[] a, Pair<? super Manager> result)
51. {
52.     minmaxBonus(a, result);
53.     PairAlg.swapHelper(result); // OK--swapHelper captures wildcard type
54. }
55. }
56.
57. class PairAlg
58. {
59.     public static boolean hasNulls(Pair<?> p)
60.     {
61.         return p.getFirst() == null || p.getSecond() == null;
62.     }
63. }

```

**Listing 12-3** PairTest3.java (continued)

```
64. public static void swap(Pair<?> p) { swapHelper(p); }
65.
66. public static <T> void swapHelper(Pair<T> p)
67. {
68.     T t = p.getFirst();
69.     p.setFirst(p.getSecond());
70.     p.setSecond(t);
71. }
72. }
73.
74. class Employee
75. {
76.     public Employee(String n, double s, int year, int month, int day)
77.     {
78.         name = n;
79.         salary = s;
80.         GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
81.         hireDay = calendar.getTime();
82.     }
83.
84.     public String getName()
85.     {
86.         return name;
87.     }
88.
89.     public double getSalary()
90.     {
91.         return salary;
92.     }
93.
94.     public Date getHireDay()
95.     {
96.         return hireDay;
97.     }
98.
99.     public void raiseSalary(double byPercent)
100.    {
101.        double raise = salary * byPercent / 100;
102.        salary += raise;
103.    }
104.
105.    private String name;
106.    private double salary;
107.    private Date hireDay;
108. }
109.
110. class Manager extends Employee
111. {
```

**Listing 12-3** PairTest3.java (continued)

```

112.  /**
113.     @param n the employee's name
114.     @param s the salary
115.     @param year the hire year
116.     @param month the hire month
117.     @param day the hire day
118.  */
119.  public Manager(String n, double s, int year, int month, int day)
120.  {
121.      super(n, s, year, month, day);
122.      bonus = 0;
123.  }
124.
125.  public double getSalary()
126.  {
127.      double baseSalary = super.getSalary();
128.      return baseSalary + bonus;
129.  }
130.
131.  public void setBonus(double b)
132.  {
133.      bonus = b;
134.  }
135.
136.  public double getBonus()
137.  {
138.      return bonus;
139.  }
140.
141.  private double bonus;
142. }

```

### Reflection and Generics

The `Class` class is now generic. For example, `String.class` is actually an object (in fact, the sole object) of the class `Class<String>`.

The type parameter is useful because it allows the methods of `Class<T>` to be more specific about their return types. The following methods of `Class<T>` take advantage of the type parameter:

```

T newInstance()
T cast(Object obj)
T[] getEnumConstants()
Class<? super T> getSuperclass()
Constructor<T> getConstructor(Class... parameterTypes)
Constructor<T> getDeclaredConstructor(Class... parameterTypes)

```

The `newInstance` method returns an instance of the class, obtained from the default constructor. Its return type can now be declared to be `T`, the same type as the class that is being described by `Class<T>`. That saves a cast.



The `cast` method returns the given object, now declared as type `T` if its type is indeed a subtype of `T`. Otherwise, it throws a `BadCastException`.

The `getEnumConstants` method returns `null` if this class is not an enum class or an array of the enumeration values, which are known to be of type `T`.

Finally, the `getConstructor` and `getDeclaredConstructor` methods return a `Constructor<T>` object. The `Constructor` class has also been made generic so that its `newInstance` method has the correct return type.

**API** `java.lang.Class<T>` 1.0

- `T newInstance()` 5.0  
returns a new instance constructed with the default constructor.
- `T cast(Object obj)` 5.0  
returns `obj` if it is `null` or can be converted to the type `T`, or throws a `BadCastException` otherwise.
- `T[] getEnumConstants()` 5.0  
returns an array of all values if `T` is an enumerated type, `null` otherwise.
- `Class<? super T> getSuperclass()` 5.0  
returns the superclass of this class, or `null` if `T` is not a class or the class `Object`.
- `Constructor<T> getConstructor(Class... parameterTypes)` 5.0
- `Constructor<T> getDeclaredConstructor(Class... parameterTypes)` 5.0  
gets the public constructor, or the constructor with the given parameter types.

**API** `java.lang.reflect.Constructor<T>` 1.1

- `T newInstance(Object... parameters)` 5.0  
returns a new instance constructed with the given parameters.

### Using `Class<T>` Parameters for Type Matching

It is sometimes useful to match the type variable of a `Class<T>` parameter in a generic method. Here is the canonical example:

```
public static <T> Pair<T> makePair(Class<T> c) throws InstantiationException,
    IllegalAccessException
{
    return new Pair<T>(c.newInstance(), c.newInstance());
}
```

If you call

```
makePair(Employee.class)
```

then `Employee.class` is an object of type `Class<Employee>`. The type parameter `T` of the `makePair` method matches `Employee`, and the compiler can infer that the method returns a `Pair<Employee>`.

### Generic Type Information in the Virtual Machine

One of the notable features of Java generics is the erasure of generic types in the virtual machine. Perhaps surprisingly, the erased classes still retain some faint memory of their generic origin. For example, the raw `Pair` class knows that it originated from the generic

class `Pair<T>`, even though an object of type `Pair` can't tell whether it was constructed as a `Pair<String>` or `Pair<Employee>`.

Similarly, consider a method

```
public static Comparable min(Comparable[] a)
```

that is the erasure of a generic method

```
public static <T extends Comparable<? super T>> T min(T[] a)
```

You can use the reflection API enhancements of Java SE 5.0 to determine that

- The generic method has a type parameter called `T`;
- The type parameter has a subtype bound that is itself a generic type;
- The bounding type has a wildcard parameter;
- The wildcard parameter has a supertype bound; and
- The generic method has a generic array parameter.

In other words, you get to reconstruct everything about generic classes and methods that their implementors declared. However, you won't know how the type parameters were resolved for specific objects or method calls.



**NOTE:** The type information that is contained in class files to enable reflection of generics is incompatible with older virtual machines.

In order to express generic type declarations, Java SE 5.0 introduced a new interface `Type` in the `java.lang.reflect` package. The interface has the following subtypes:

- The `Class` class, describing concrete types
- The `TypeVariable` interface, describing type variables (such as `T extends Comparable<? super T>`)
- The `WildcardType` interface, describing wildcards (such as `? super T`)
- The `ParameterizedType` interface, describing generic class or interface types (such as `Comparable<? super T>`)
- The `GenericArrayType` interface, describing generic arrays (such as `T[]`)

Figure 12-5 shows the inheritance hierarchy. Note that the last four subtypes are interfaces—the virtual machine instantiates suitable classes that implement these interfaces.

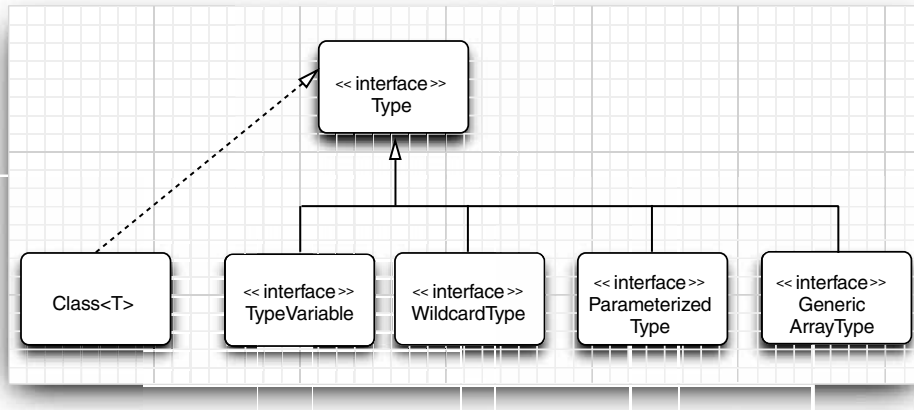
Listing 12-4 uses the generic reflection API to print out what it discovers about a given class. If you run it with the `Pair` class, you get this report:

```
class Pair<T> extends java.lang.Object
  public T getFirst()
  public T getSecond()
  public void setFirst(T)
  public void setSecond(T)
```

If you run it with `ArrayList` in the `PairTest2` directory, the report displays the following method:

```
public static <T extends java.lang.Comparable> Pair<T> minmax(T[])
```

The API notes at the end of this section describe the methods used in the example program.

**Figure 12-5** The Type class and its descendants**Listing 12-4** GenericReflectionTest.java

```
1. import java.lang.reflect.*;
2. import java.util.*;
3.
4. /**
5.  * @version 1.10 2004-05-15
6.  * @author Cay Horstmann
7.  */
8. public class GenericReflectionTest
9. {
10.     public static void main(String[] args)
11.     {
12.         // read class name from command-line args or user input
13.         String name;
14.         if (args.length > 0) name = args[0];
15.         else
16.         {
17.             Scanner in = new Scanner(System.in);
18.             System.out.println("Enter class name (e.g. java.util.Collections): ");
19.             name = in.next();
20.         }
21.
22.         try
23.         {
24.             // print generic info for class and public methods
25.             Class c1 = Class.forName(name);
26.             printClass(c1);
27.             for (Method m : c1.getDeclaredMethods())
28.                 printMethod(m);
```

**Listing 12-4** GenericReflectionTest.java (continued)

```
29.     }
30.     catch (ClassNotFoundException e)
31.     {
32.         e.printStackTrace();
33.     }
34. }
35.
36. public static void printClass(Class cl)
37. {
38.     System.out.print(cl);
39.     printTypes(cl.getTypeParameters(), "<", " ", ">", true);
40.     Type sc = cl.getGenericSuperclass();
41.     if (sc != null)
42.     {
43.         System.out.print(" extends ");
44.         printType(sc, false);
45.     }
46.     printTypes(cl.getGenericInterfaces(), " implements ", " ", "", false);
47.     System.out.println();
48. }
49.
50. public static void printMethod(Method m)
51. {
52.     String name = m.getName();
53.     System.out.print(Modifier.toString(m.getModifiers()));
54.     System.out.print(" ");
55.     printTypes(m.getTypeParameters(), "<", " ", "> ", true);
56.
57.     printType(m.getGenericReturnType(), false);
58.     System.out.print(" ");
59.     System.out.print(name);
60.     System.out.print("(");
61.     printTypes(m.getGenericParameterTypes(), "", " ", " ", false);
62.     System.out.println(")");
63. }
64.
65. public static void printTypes(Type[] types, String pre, String sep, String suf,
66.     boolean isDefinition)
67. {
68.     if (pre.equals(" extends ") && Arrays.equals(types, new Type[] { Object.class })) return;
69.     if (types.length > 0) System.out.print(pre);
70.     for (int i = 0; i < types.length; i++)
71.     {
72.         if (i > 0) System.out.print(sep);
73.         printType(types[i], isDefinition);
74.     }
75.     if (types.length > 0) System.out.print(suf);
76. }
77.
```

**Listing 12-4** GenericReflectionTest.java (continued)

```
78. public static void printType(Type type, boolean isDefinition)
79. {
80.     if (type instanceof Class)
81.     {
82.         Class t = (Class) type;
83.         System.out.print(t.getName());
84.     }
85.     else if (type instanceof TypeVariable)
86.     {
87.         TypeVariable t = (TypeVariable) type;
88.         System.out.print(t.getName());
89.         if (isDefinition)
90.             printTypes(t.getBounds(), " extends ", " & ", "", false);
91.     }
92.     else if (type instanceof WildcardType)
93.     {
94.         WildcardType t = (WildcardType) type;
95.         System.out.print("?");
96.         printTypes(t.getUpperBounds(), " extends ", " & ", "", false);
97.         printTypes(t.getLowerBounds(), " super ", " & ", "", false);
98.     }
99.     else if (type instanceof ParameterizedType)
100.    {
101.        ParameterizedType t = (ParameterizedType) type;
102.        Type owner = t.getOwnerType();
103.        if (owner != null)
104.        {
105.            printType(owner, false);
106.            System.out.print(".");
107.        }
108.        printType(t.getRawType(), false);
109.        printTypes(t.getActualTypeArguments(), "<", " ", ">", false);
110.    }
111.    else if (type instanceof GenericArrayType)
112.    {
113.        GenericArrayType t = (GenericArrayType) type;
114.        System.out.print("");
115.        printType(t.getGenericComponentType(), isDefinition);
116.        System.out.print("[ ]");
117.    }
118. }
119. }
120. }
```

**API** `java.lang.Class<T>` 1.0

- `TypeVariable[] getTypeParameters()` 5.0  
gets the generic type variables if this type was declared as a generic type, or an array of length 0 otherwise.
- `Type getGenericSuperclass()` 5.0  
gets the generic type of the superclass that was declared for this type, or `null` if this type is `Object` or not a class type.
- `Type[] getGenericInterfaces()` 5.0  
gets the generic types of the interfaces that were declared for this type, in declaration order, or an array of length 0 if this type doesn't implement interfaces.

**API** `java.lang.reflect.Method` 1.1

- `TypeVariable[] getTypeParameters()` 5.0  
gets the generic type variables if this method was declared as a generic method, or an array of length 0 otherwise.
- `Type getGenericReturnType()` 5.0  
gets the generic return type with which this method was declared.
- `Type[] getGenericParameterTypes()` 5.0  
gets the generic parameter types with which this method was declared. If the method has no parameters, an array of length 0 is returned.

**API** `java.lang.reflect.TypeVariable` 5.0

- `String getName()`  
gets the name of this type variable.
- `Type[] getBounds()`  
gets the subclass bounds of this type variable, or an array of length 0 if the variable is unbounded.

**API** `java.lang.reflect.WildcardType` 5.0

- `Type[] getLowerBounds()`  
gets the subclass (extends) bounds of this type variable, or an array of length 0 has no subclass bounds
- `Type[] getUpperBounds()`  
gets the superclass (super) bounds of this type variable, or an array of length 0 has no superclass bounds.

**API** `java.lang.reflect.ParameterizedType` 5.0

- `Type getRawType()`  
gets the raw type of this parameterized type.

- `Type[] getActualTypeArguments()`  
gets the type parameters with which this parameterized type was declared.
- `Type getOwnerType()`  
gets the outer class type if this is an inner type, or null if this is a top-level type.

**API** `java.lang.reflect.GenericArrayType` 5.0

- `Type getGenericComponentType()`  
gets the generic component type with which this array type was declared.

You now know how to use generic classes and how to program your own generic classes and methods if the need arises. Just as importantly, you know how to decipher the generic type declarations that you may encounter in the API documentation and in error messages. For an exhaustive discussion of everything there is to know about Java generics, turn to Angelika Langer's excellent list of frequently (and not so frequently) asked questions at <http://angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html>.

In the next chapter, you will see how the Java collections framework puts generics to work.





---

*Chapter*

13

COLLECTIONS

- ▼ COLLECTION INTERFACES
- ▼ CONCRETE COLLECTIONS
- ▼ THE COLLECTIONS FRAMEWORK
- ▼ ALGORITHMS
- ▼ LEGACY COLLECTIONS

The data structures that you choose can make a big difference when it comes to implementing methods in a natural style, as well as for performance. Do you need to search quickly through thousands (or even millions) of sorted items? Do you need to rapidly insert and remove elements in the middle of an ordered sequence? Do you need to establish associations between keys and values?

This chapter shows how the Java library can help you accomplish the traditional data structuring needed for serious programming. In college computer science programs, a course called *Data Structures* usually takes a semester to complete, so there are many, many books devoted to this important topic. Our coverage differs from that of a college course; we skip the theory and just tell you how to use the collection classes in the standard library.

### Collection Interfaces

The initial release of Java supplied only a small set of classes for the most useful data structures: `Vector`, `Stack`, `Hashtable`, `BitSet`, and the `Enumeration` interface that provides an abstract mechanism for visiting elements in an arbitrary container. That was certainly a wise choice—it takes time and skill to come up with a comprehensive collection class library.

With the advent of Java SE 1.2, the designers felt that the time had come to roll out a full-fledged set of data structures. They faced a number of conflicting design decisions. They wanted the library to be small and easy to learn. They did not want the complexity of the “Standard Template Library” (or STL) of C++, but they wanted the benefit of “generic algorithms” that STL pioneered. They wanted the legacy classes to fit into the new framework. As all designers of collection libraries do, they had to make some hard choices, and they came up with a number of idiosyncratic design decisions along the way. In this section, we will explore the basic design of the Java collections framework, show you how to put it to work, and explain the reasoning behind some of the more controversial features.

### Separating Collection Interfaces and Implementation

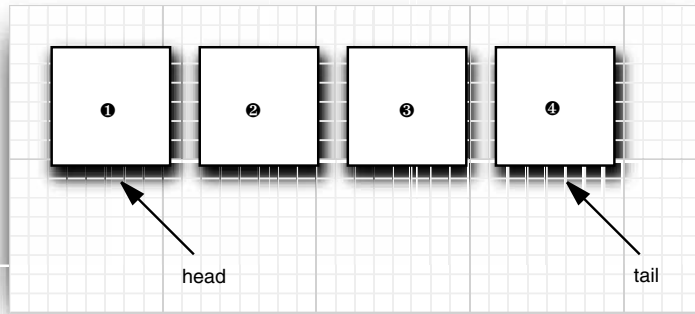
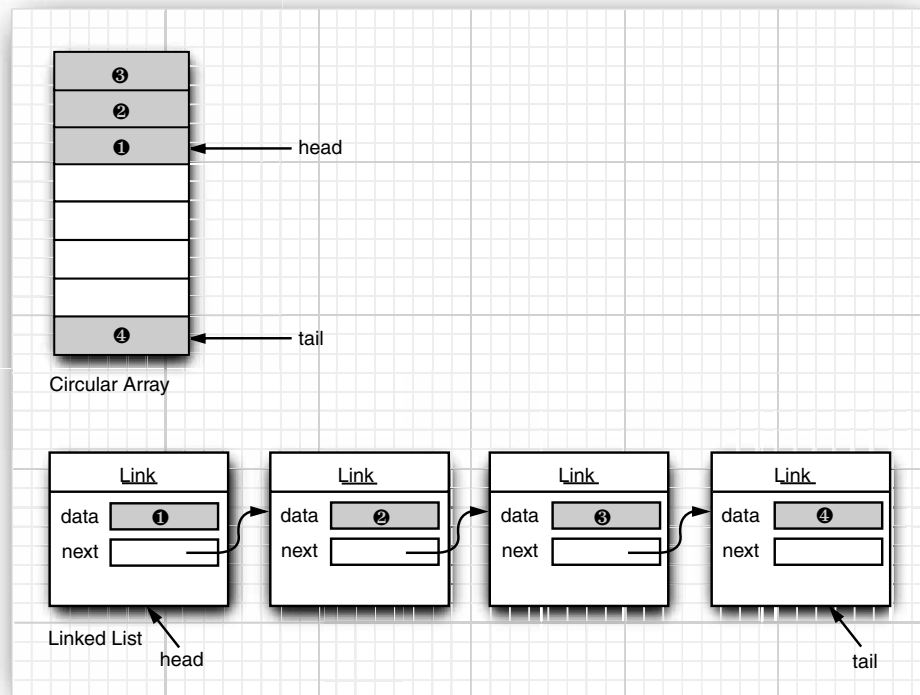
As is common for modern data structure libraries, the Java collection library separates *interfaces* and *implementations*. Let us look at that separation with a familiar data structure, the *queue*.

A *queue interface* specifies that you can add elements at the tail end of the queue, remove them at the head, and find out how many elements are in the queue. You use a queue when you need to collect objects and retrieve them in a “first in, first out” fashion (see Figure 13–1).


A minimal form of a queue interface might look like this:

```
interface Queue<E> // a simplified form of the interface in the standard library
{
    void add(E element);
    E remove();
    int size();
}
```

The interface tells you nothing about how the queue is implemented. Of the two common implementations of a queue, one uses a “circular array” and one uses a linked list (see Figure 13–2).

**Figure 13-1** A queue**Figure 13-2** Queue implementations

---

 NOTE: As of Java SE 5.0, the collection classes are generic classes with type parameters. For more information on generic classes, please turn to Chapter 12.

---


Each implementation can be expressed by a class that implements the `Queue` interface.

```
class CircularArrayQueue<E> implements Queue<E> // not an actual library class
{
    CircularArrayQueue(int capacity) { . . . }
    public void add(E element) { . . . }
    public E remove() { . . . }
    public int size() { . . . }
    private E[] elements;
    private int head;
    private int tail;
}

class LinkedListQueue<E> implements Queue<E> // not an actual library class
{
    LinkedListQueue() { . . . }
    public void add(E element) { . . . }
    public E remove() { . . . }
    public int size() { . . . }

    private Link head;
    private Link tail;
}
```

---

 NOTE: The Java library doesn't actually have classes named `CircularArrayQueue` and `LinkedListQueue`. We use these classes as examples to explain the conceptual distinction between collection interfaces and implementations. If you need a circular array queue, use the `ArrayDeque` class that was introduced in Java SE 6. For a linked list queue, simply use the `LinkedList` class—it implements the `Queue` interface.

---

When you use a queue in your program, you don't need to know which implementation is actually used once the collection has been constructed. Therefore, it makes sense to use the concrete class *only* when you construct the collection object. Use the *interface type* to hold the collection reference.

```
Queue<Customer> expressLane = new CircularArrayQueue<Customer>(100);
expressLane.add(new Customer("Harry"));
```

With this approach if you change your mind, you can easily use a different implementation. You only need to change your program in one place—the constructor call. If you decide that a `LinkedListQueue` is a better choice after all, your code becomes

```
Queue<Customer> expressLane = new LinkedListQueue<Customer>();
expressLane.add(new Customer("Harry"));
```

Why would you choose one implementation over another? The interface says nothing about the efficiency of the implementation. A circular array is somewhat more efficient than a linked list, so it is generally preferable. However, as usual, there is a price to pay.

The circular array is a *bounded* collection—it has a finite capacity. If you don't have an upper limit on the number of objects that your program will collect, you may be better off with a linked list implementation after all.

When you study the API documentation, you will find another set of classes whose name begins with *Abstract*, such as *AbstractQueue*. These classes are intended for library implementors. In the (perhaps unlikely) event that you want to implement your own queue class, you will find it easier to extend *AbstractQueue* than to implement all the methods of the *Queue* interface.

### **Collection and Iterator Interfaces in the Java Library**

The fundamental interface for collection classes in the Java library is the *Collection* interface. The interface has two fundamental methods:

```
public interface Collection<E>
{
    boolean add(E element);
    Iterator<E> iterator();
    . . .
}
```

There are several methods in addition to these two; we discuss them later.

The *add* method adds an element to the collection. The *add* method returns *true* if adding the element actually changes the collection, and *false* if the collection is unchanged. For example, if you try to add an object to a set and the object is already present, then the *add* request has no effect because sets reject duplicates.

The *iterator* method returns an object that implements the *Iterator* interface. You can use the *iterator* object to visit the elements in the collection one by one.

#### **Iterators**

The *Iterator* interface has three methods:

```
public interface Iterator<E>
{
    E next();
    boolean hasNext();
    void remove();
}
```

By repeatedly calling the *next* method, you can visit the elements from the collection one by one. However, if you reach the end of the collection, the *next* method throws a *NoSuchElementException*. Therefore, you need to call the *hasNext* method before calling *next*. That method returns *true* if the *iterator* object still has more elements to visit. If you want to inspect all elements in a collection, you request an *iterator* and then keep calling the *next* method while *hasNext* returns *true*. For example:

```
Collection<String> c = . . .;
Iterator<String> iter = c.iterator();
while (iter.hasNext())
{
    String element = iter.next();
    do something with element
}
```

As of Java SE 5.0, there is an elegant shortcut for this loop. You write the same loop more concisely with the “for each” loop:

```
for (String element : c)
{
    do something with element
}
```

The compiler simply translates the “for each” loop into a loop with an iterator.

The “for each” loop works with any object that implements the `Iterable` interface, an interface with a single method:

```
public interface Iterable<E>
{
    Iterator<E> iterator();
}
```

The `Collection` interface extends the `Iterable` interface. Therefore, you can use the “for each” loop with any collection in the standard library.

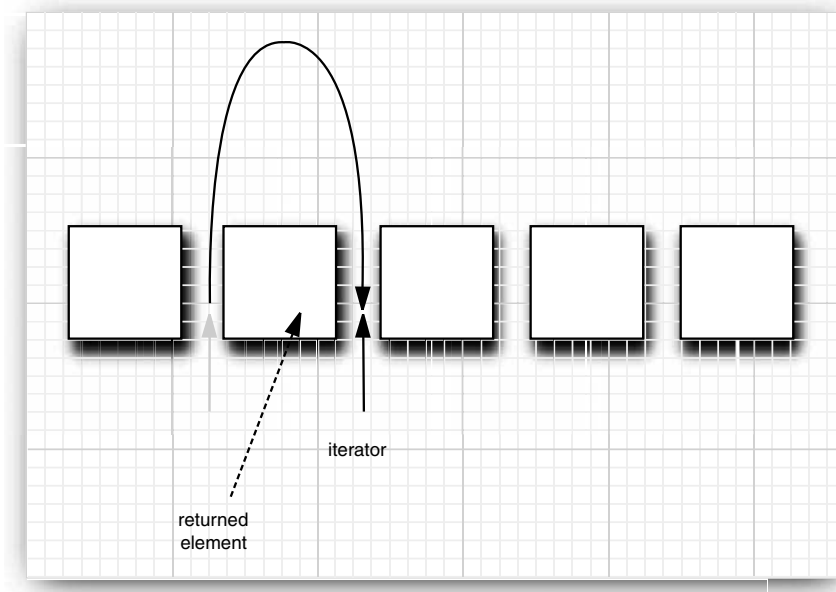
The order in which the elements are visited depends on the collection type. If you iterate over an `ArrayList`, the iterator starts at index 0 and increments the index in each step. However, if you visit the elements in a `HashSet`, you will encounter them in essentially random order. You can be assured that you will encounter all elements of the collection during the course of the iteration, but you cannot make any assumptions about their ordering. This is usually not a problem because the ordering does not matter for computations such as computing totals or counting matches.



**NOTE:** Old-timers will notice that the `next` and `hasNext` methods of the `Iterator` interface serve the same purpose as the `nextElement` and `hasMoreElements` methods of an `Enumeration`. The designers of the Java collection library could have chosen to make use of the `Enumeration` interface. But they disliked the cumbersome method names and instead introduced a new interface with shorter method names.

There is an important conceptual difference between iterators in the Java collection library and iterators in other libraries. In traditional collection libraries such as the Standard Template Library of C++, iterators are modeled after array indexes. Given such an iterator, you can look up the element that is stored at that position, much like you can look up an array element `a[i]` if you have an array index `i`. Independently of the lookup, you can advance the iterator to the next position. This is the same operation as advancing an array index by calling `i++`, without performing a lookup. However, the Java iterators do not work like that. The lookup and position change are tightly coupled. The only way to look up an element is to call `next`, and that lookup advances the position.

Instead, you should think of Java iterators as being *between elements*. When you call `next`, the iterator *jumps over* the next element, and it returns a reference to the element that it just passed (see Figure 13–3).



**Figure 13-3 Advancing an iterator**



**NOTE:** Here is another useful analogy. You can think of `Iterator.next` as the equivalent of `InputStream.read`. Reading a byte from a stream automatically “consumes” the byte. The next call to read consumes and returns the next byte from the input. Similarly, repeated calls to `next` let you read all elements in a collection.

### Removing Elements

The `remove` method of the `Iterator` interface removes the element that was returned by the last call to `next`. In many situations, that makes sense—you need to see the element before you can decide that it is the one that should be removed. But if you want to remove an element in a particular position, you still need to skip past the element. For example, here is how you remove the first element in a collection of strings:

```
Iterator<String> it = c.iterator();
it.next(); // skip over the first element
it.remove(); // now remove it
```

More important, there is a dependency between calls to the `next` and `remove` methods. It is illegal to call `remove` if it wasn't preceded by a call to `next`. If you try, an `IllegalStateException` is thrown.

If you want to remove two adjacent elements, you cannot simply call

```
it.remove();
it.remove(); // Error!
```

Instead, you must first call `next` to jump over the element to be removed.

```
it.remove();
it.next();
it.remove(); // Ok
```

### Generic Utility Methods

Because the `Collection` and `Iterator` interfaces are generic, you can write utility methods that operate on any kind of collection. For example, here is a generic method that tests whether an arbitrary collection contains a given element:

```
public static <E> boolean contains(Collection<E> c, Object obj)
{
    for (E element : c)
        if (element.equals(obj))
            return true;
    return false;
}
```

The designers of the Java library decided that some of these utility methods are so useful that the library should make them available. That way, library users don't have to keep reinventing the wheel. The `contains` method is one such method.

In fact, the `Collection` interface declares quite a few useful methods that all implementing classes must supply. Among them are

```
int size()
boolean isEmpty()
boolean contains(Object obj)
boolean containsAll(Collection<?> c)
boolean equals(Object other)
boolean addAll(Collection<? extends E> from)
boolean remove(Object obj)
boolean removeAll(Collection<?> c)
void clear()
boolean retainAll(Collection<?> c)
Object[] toArray()
<T> T[] toArray(T[] arrayToFill)
```

Many of these methods are self-explanatory; you will find full documentation in the API notes at the end of this section.

Of course, it is a bother if every class that implements the `Collection` interface has to supply so many routine methods. To make life easier for implementors, the library supplies a class `AbstractCollection` that leaves the fundamental methods `size` and `iterator` abstract but implements the routine methods in terms of them. For example:

```
public abstract class AbstractCollection<E>
    implements Collection<E>
{
    . . .
    public abstract Iterator<E> iterator();

    public boolean contains(Object obj)
    {
        for (E element : c) // calls iterator()

```



```

        if (element.equals(obj))
            return = true;
        return false;
    }
    . . .
}

```

A concrete collection class can now extend the `AbstractCollection` class. It is now up to the concrete collection class to supply an `iterator` method, but the `contains` method has been taken care of by the `AbstractCollection` superclass. However, if the subclass has a more efficient way of implementing `contains`, it is free to do so.

This is a good design for a class framework. The users of the collection classes have a richer set of methods available in the generic interface, but the implementors of the actual data structures do not have the burden of implementing all the routine methods.

#### API `java.util.Collection<E>` 1.2

- `Iterator<E> iterator()`  
returns an iterator that can be used to visit the elements in the collection.
- `int size()`  
returns the number of elements currently stored in the collection.
- `boolean isEmpty()`  
returns `true` if this collection contains no elements.
- `boolean contains(Object obj)`  
returns `true` if this collection contains an object equal to `obj`.
- `boolean containsAll(Collection<?> other)`  
returns `true` if this collection contains all elements in the other collection.
- `boolean add(Object element)`  
adds an element to the collection. Returns `true` if the collection changed as a result of this call.
- `boolean addAll(Collection<? extends E> other)`  
adds all elements from the other collection to this collection. Returns `true` if the collection changed as a result of this call.
- `boolean remove(Object obj)`  
removes an object equal to `obj` from this collection. Returns `true` if a matching object was removed.
- `boolean removeAll(Collection<?> other)`  
removes from this collection all elements from the other collection. Returns `true` if the collection changed as a result of this call.
- `void clear()`  
removes all elements from this collection.
- `boolean retainAll(Collection<?> other)`  
removes all elements from this collection that do not equal one of the elements in the other collection. Returns `true` if the collection changed as a result of this call.
- `Object[] toArray()`  
returns an array of the objects in the collection.

- `<T> T[] toArray(T[] arrayToFill)`  
returns an array of the objects in the collection. If `arrayToFill` has sufficient length, it is filled with the elements of this collection. If there is space, a `null` element is appended. Otherwise, a new array with the same component type as `arrayToFill` and the same length as the size of this collection is allocated and filled.

**API** `java.util.Iterator<E>` 1.2

- `boolean hasNext()`  
returns `true` if there is another element to visit.
- `E next()`  
returns the next object to visit. Throws a `NoSuchElementException` if the end of the collection has been reached.
- `void remove()`  
removes the last visited object. This method must immediately follow an element visit. If the collection has been modified since the last element visit, then the method throws an `IllegalStateException`.

**Concrete Collections**

Rather than getting into more details about all the interfaces, we thought it would be helpful to first discuss the concrete data structures that the Java library supplies. Once we have thoroughly described the classes you might want to use, we will return to abstract considerations and see how the collections framework organizes these classes. Table 13–1 shows the collections in the Java library and briefly describes the purpose of each collection class. (For simplicity, we omit the thread-safe collections that will be discussed in Chapter 14.) All classes in Table 13–1 implement the `Collection` interface, with the exception of the classes with names ending in `Map`. Those classes implement the `Map` interface instead. We will discuss the `Map` interface in the section “Maps” on page 680.

**Table 13–1 Concrete Collections in the Java Library**

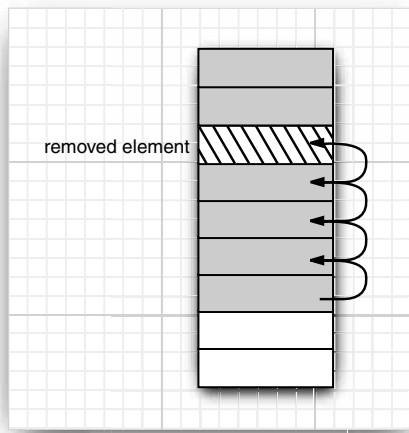
Collection Type	Description	See Page
<code>ArrayList</code>	An indexed sequence that grows and shrinks dynamically	668
<code>LinkedList</code>	An ordered sequence that allows efficient insertions and removal at any location	659
<code>ArrayDeque</code>	A double-ended queue that is implemented as a circular array	678
<code>HashSet</code>	An unordered collection that rejects duplicates	668
<code>TreeSet</code>	A sorted set	672
<code>EnumSet</code>	A set of enumerated type values	687
<code>LinkedHashSet</code>	A set that remembers the order in which elements were inserted	686

**Table 13–1 Concrete Collections in the Java Library (continued)**

Collection Type	Description	See Page
PriorityQueue	A collection that allows efficient removal of the smallest element	679
HashMap	A data structure that stores key/value associations	680
TreeMap	A map in which the keys are sorted	680
EnumMap	A map in which the keys belong to an enumerated type	687
LinkedHashMap	A map that remembers the order in which entries were added	686
WeakHashMap	A map with values that can be reclaimed by the garbage collector if they are not used elsewhere	685
IdentityHashMap	A map with keys that are compared by ==, not equals	688

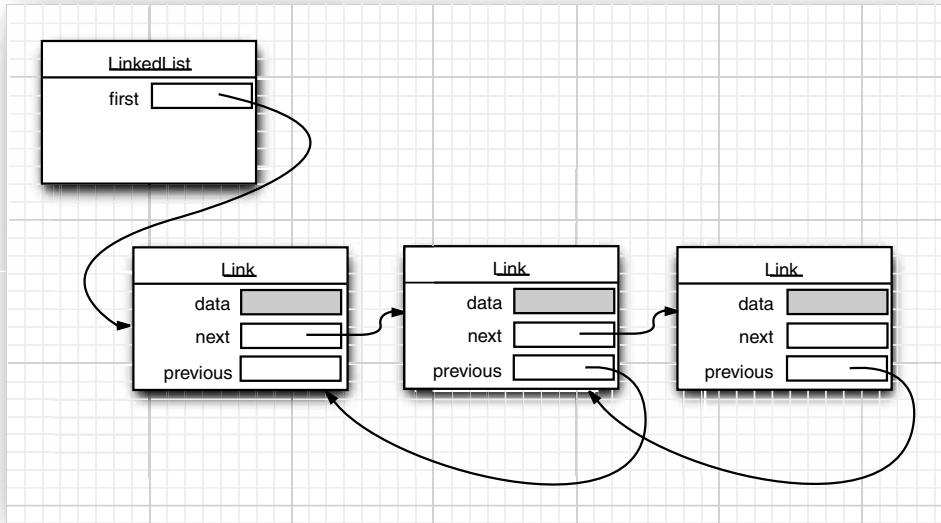
**Linked Lists**

We already used arrays and their dynamic cousin, the `ArrayList` class, for many examples in this book. However, arrays and array lists suffer from a major drawback. Removing an element from the middle of an array is expensive since all array elements beyond the removed one must be moved toward the beginning of the array (see Figure 13–4). The same is true for inserting elements in the middle.

**Figure 13–4 Removing an element from an array**

Another well-known data structure, the *linked list*, solves this problem. Whereas an array stores object references in consecutive memory locations, a linked list stores each

object in a separate *link*. Each link also stores a reference to the next link in the sequence. In the Java programming language, all linked lists are actually *doubly linked*; that is, each link also stores a reference to its predecessor (see Figure 13–5).



**Figure 13–5** A doubly linked list

Removing an element from the middle of a linked list is an inexpensive operation—only the links around the element to be removed need to be updated (see Figure 13–6).

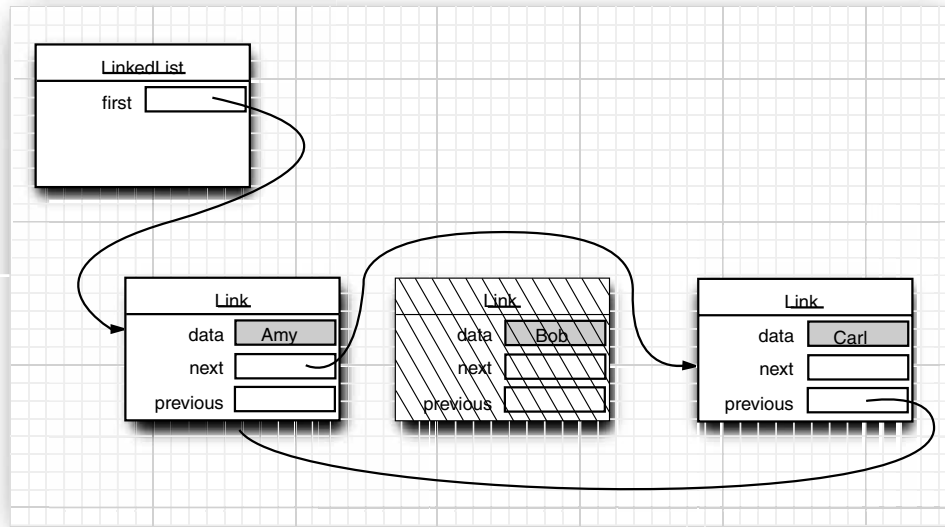
Perhaps you once took a data structures course in which you learned how to implement linked lists. You may have bad memories of tangling up the links when removing or adding elements in the linked list. If so, you will be pleased to learn that the Java collections library supplies a class `LinkedList` ready for you to use.

The following code example adds three elements and then removes the second one:

```

List<String> staff = new LinkedList<String>(); // LinkedList implements List
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
Iterator iter = staff.iterator();
String first = iter.next(); // visit first element
String second = iter.next(); // visit second element
iter.remove(); // remove last visited element
  
```

There is, however, an important difference between linked lists and generic collections. A linked list is an *ordered collection* in which the position of the objects matters. The `LinkedList.add` method adds the object to the end of the list. But you often want to add objects somewhere in the middle of a list. This position-dependent `add` method is the



**Figure 13-6 Removing an element from a linked list**

responsibility of an iterator, since iterators describe positions in collections. Using iterators to add elements makes sense only for collections that have a natural ordering. For example, the *set* data type that we discuss in the next section does not impose any ordering on its elements. Therefore, there is no `add` method in the `Iterator` interface. Instead, the collections library supplies a subinterface `ListIterator` that contains an `add` method:

```

interface ListIterator<E> extends Iterator<E>
{
    void add(E element);
    . . .
}
  
```

Unlike `Collection.add`, this method does not return a `boolean`—it is assumed that the `add` operation always modifies the list.

In addition, the `ListIterator` interface has two methods that you can use for traversing a list backwards.

```

E previous()
boolean hasPrevious()
  
```

Like the `next` method, the `previous` method returns the object that it skipped over.

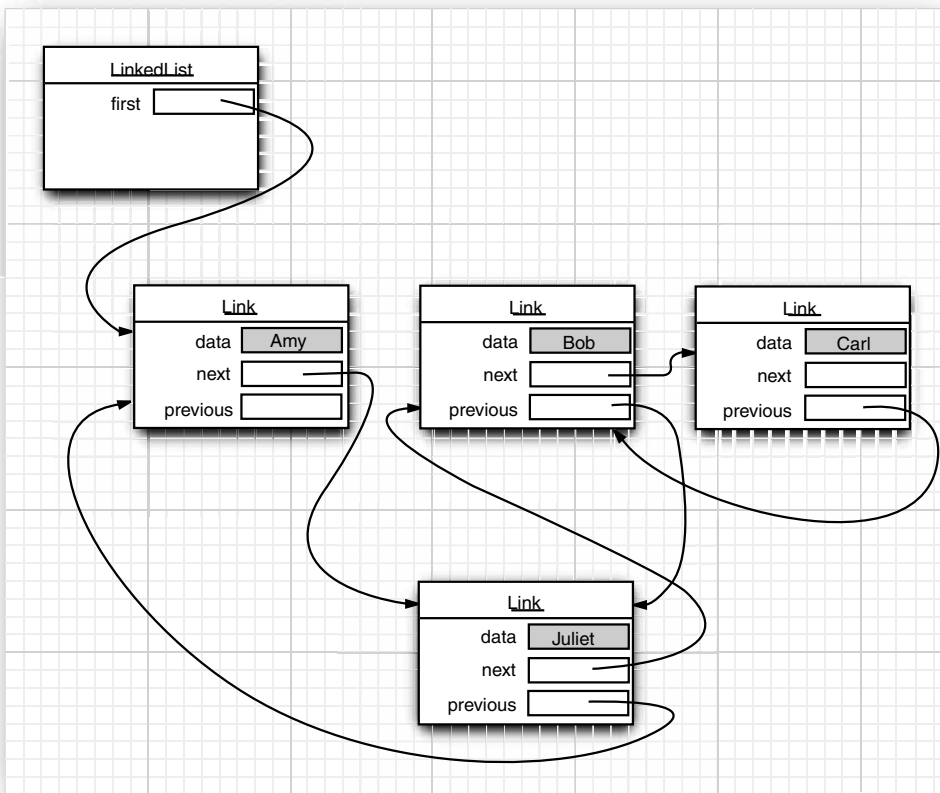
The `listIterator` method of the `LinkedList` class returns an iterator object that implements the `ListIterator` interface.

```

ListIterator<String> iter = staff.listIterator();
  
```

The `add` method adds the new element *before* the iterator position. For example, the following code skips past the first element in the linked list and adds "Juliet" before the second element (see Figure 13-7):

```
List<String> staff = new LinkedList<String>();
staff.add("Amy");
staff.add("Bob");
staff.add("Carl");
ListIterator<String> iter = staff.listIterator();
iter.next(); // skip past first element
iter.add("Juliet");
```



**Figure 13-7** Adding an element to a linked list

If you call the `add` method multiple times, the elements are simply added in the order in which you supplied them. They are all added in turn before the current iterator position.

When you use the `add` operation with an iterator that was freshly returned from the `listIterator` method and that points to the beginning of the linked list, the newly added element becomes the new head of the list. When the iterator has passed the last element of the list (that is, when `hasNext` returns `false`), the added element becomes the new tail of the list. If the linked list has  $n$  elements, there are  $n + 1$  spots for adding a new element. These spots correspond to the  $n + 1$  possible positions of the iterator. For example, if a linked list contains three elements, A, B, and C, then there are four possible positions (marked as `|`) for inserting a new element:

```
|ABC
A|BC
AB|C
ABC|
```



**NOTE:** You have to be careful with the “cursor” analogy. The `remove` operation does not quite work like the `BACKSPACE` key. Immediately after a call to `next`, the `remove` method indeed removes the element to the left of the iterator, just like the `BACKSPACE` key would. However, if you just called `previous`, the element to the right is removed. And you can’t call `remove` twice in a row.

Unlike the `add` method, which depends only on the iterator position, the `remove` method depends on the iterator state.

Finally, a `set` method replaces the last element returned by a call to `next` or `previous` with a new element. For example, the following code replaces the first element of a list with a new value:

```
ListIterator<String> iter = list.listIterator();
String oldValue = iter.next(); // returns first element
iter.set(newValue); // sets first element to newValue
```

As you might imagine, if an iterator traverses a collection while another iterator is modifying it, confusing situations can occur. For example, suppose an iterator points before an element that another iterator has just removed. The iterator is now invalid and should no longer be used. The linked list iterators have been designed to detect such modifications. If an iterator finds that its collection has been modified by another iterator or by a method of the collection itself, then it throws a `ConcurrentModificationException`. For example, consider the following code:

```
List<String> list = . . . ;
ListIterator<String> iter1 = list.listIterator();
ListIterator<String> iter2 = list.listIterator();
iter1.next();
iter1.remove();
iter2.next(); // throws ConcurrentModificationException
```

The call to `iter2.next` throws a `ConcurrentModificationException` since `iter2` detects that the list was modified externally.

To avoid concurrent modification exceptions, follow this simple rule: You can attach as many iterators to a collection as you like, provided that all of them are only readers. Alternatively, you can attach a single iterator that can both read and write.

Concurrent modification detection is achieved in a simple way. The collection keeps track of the number of mutating operations (such as adding and removing elements). Each iterator keeps a separate count of the number of mutating operations that *it* was responsible for. At the beginning of each iterator method, the iterator simply checks whether its own mutation count equals that of the collection. If not, it throws a `ConcurrentModificationException`.



NOTE: There is, however, a curious exception to the detection of concurrent modifications. The linked list only keeps track of *structural* modifications to the list, such as adding and removing links. The `set` method does *not* count as a structural modification. You can attach multiple iterators to a linked list, all of which call `set` to change the contents of existing links. This capability is required for a number of algorithms in the `Collections` class that we discuss later in this chapter.

Now you have seen the fundamental methods of the `LinkedList` class. You use a `ListIterator` to traverse the elements of the linked list in either direction and to add and remove elements.

As you saw in the preceding section, many other useful methods for operating on linked lists are declared in the `Collection` interface. These are, for the most part, implemented in the `AbstractCollection` superclass of the `LinkedList` class. For example, the `toString` method invokes `toString` on all elements and produces one long string of the format `[A, B, C]`. This is handy for debugging. Use the `contains` method to check whether an element is present in a linked list. For example, the call `staff.contains("Harry")` returns `true` if the linked list already contains a string that is equal to the string "Harry".

The library also supplies a number of methods that are, from a theoretical perspective, somewhat dubious. Linked lists do not support fast random access. If you want to see the  $n$ th element of a linked list, you have to start at the beginning and skip past the first  $n - 1$  elements first. There is no shortcut. For that reason, programmers don't usually use linked lists in programming situations in which elements need to be accessed by an integer index.

Nevertheless, the `LinkedList` class supplies a `get` method that lets you access a particular element:

```
LinkedList<String> list = . . . ;
String obj = list.get(n);
```

Of course, this method is not very efficient. If you find yourself using it, you are probably using the wrong data structure for your problem.

You should *never* use this illusory random access method to step through a linked list. The code

```
for (int i = 0; i < list.size(); i++)
    do something with list.get(i);
```

is staggeringly inefficient. Each time you look up another element, the search starts again from the beginning of the list. The `LinkedList` object makes no effort to cache the position information.





**NOTE:** The `get` method has one slight optimization: If the index is at least `size() / 2`, then the search for the element starts at the end of the list.

The list iterator interface also has a method to tell you the index of the current position. In fact, because Java iterators conceptually point between elements, it has two of them: The `nextIndex` method returns the integer index of the element that would be returned by the next call to `next`; the `previousIndex` method returns the index of the element that would be returned by the next call to `previous`. Of course, that is simply one less than `nextIndex`. These methods are efficient—the iterators keep a count of the current position. Finally, if you have an integer index `n`, then `list.listIterator(n)` returns an iterator that points just before the element with index `n`. That is, calling `next` yields the same element as `list.get(n)`; obtaining that iterator is inefficient.

If you have a linked list with only a handful of elements, then you don't have to be overly paranoid about the cost of the `get` and `set` methods. But then why use a linked list in the first place? The only reason to use a linked list is to minimize the cost of insertion and removal in the middle of the list. If you have only a few elements, you can just use an `ArrayList`.

We recommend that you simply stay away from all methods that use an integer index to denote a position in a linked list. If you want random access into a collection, use an array or `ArrayList`, not a linked list.

The program in Listing 13–1 puts linked lists to work. It simply creates two lists, merges them, then removes every second element from the second list, and finally tests the `removeAll` method. We recommend that you trace the program flow and pay special attention to the iterators. You may find it helpful to draw diagrams of the iterator positions, like this:

```
|ACE |BDFG
A|CE |BDFG
AB|CE B|DFG
. . .
```

Note that the call

```
System.out.println(a);
```

prints all elements in the linked list `a` by invoking the `toString` method in `AbstractCollection`.

#### Listing 13–1 LinkedListTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates operations on linked lists.
5.  * @version 1.10 2004-08-02
6.  * @author Cay Horstmann
7.  */
8. public class LinkedListTest
9. {
```

**Listing 13-1** LinkedListTest.java (continued)

```
10. public static void main(String[] args)
11. {
12.     List<String> a = new LinkedList<String>();
13.     a.add("Amy");
14.     a.add("Carl");
15.     a.add("Erica");
16.
17.     List<String> b = new LinkedList<String>();
18.     b.add("Bob");
19.     b.add("Doug");
20.     b.add("Frances");
21.     b.add("Gloria");
22.
23.     // merge the words from b into a
24.
25.     ListIterator<String> aIter = a.listIterator();
26.     Iterator<String> bIter = b.iterator();
27.
28.     while (bIter.hasNext())
29.     {
30.         if (aIter.hasNext()) aIter.next();
31.         aIter.add(bIter.next());
32.     }
33.
34.     System.out.println(a);
35.
36.     // remove every second word from b
37.
38.     bIter = b.iterator();
39.     while (bIter.hasNext())
40.     {
41.         bIter.next(); // skip one element
42.         if (bIter.hasNext())
43.         {
44.             bIter.next(); // skip next element
45.             bIter.remove(); // remove that element
46.         }
47.     }
48.
49.     System.out.println(b);
50.
51.     // bulk operation: remove all words in b from a
52.
53.     a.removeAll(b);
54.
55.     System.out.println(a);
56. }
57. }
```

---

**API** `java.util.List<E>` 1.2

- `ListIterator<E> listIterator()`  
returns a list iterator for visiting the elements of the list.
- `ListIterator<E> listIterator(int index)`  
returns a list iterator for visiting the elements of the list whose first call to `next` will return the element with the given index.
- `void add(int i, E element)`  
adds an element at the specified position.
- `void addAll(int i, Collection<? extends E> elements)`  
adds all elements from a collection to the specified position.
- `E remove(int i)`  
removes and returns the element at the specified position.
- `E get(int i)`  
gets the element at the specified position.
- `E set(int i, E element)`  
replaces the element at the specified position with a new element and returns the old element.
- `int indexOf(Object element)`  
returns the position of the first occurrence of an element equal to the specified element, or `-1` if no matching element is found.
- `int lastIndexOf(Object element)`  
returns the position of the last occurrence of an element equal to the specified element, or `-1` if no matching element is found.

**API** `java.util.ListIterator<E>` 1.2

- `void add(E newElement)`  
adds an element before the current position.
- `void set(E newElement)`  
replaces the last element visited by `next` or `previous` with a new element. Throws an `IllegalStateException` if the list structure was modified since the last call to `next` or `previous`.
- `boolean hasPrevious()`  
returns `true` if there is another element to visit when iterating backwards through the list.
- `E previous()`  
returns the previous object. Throws a `NoSuchElementException` if the beginning of the list has been reached.
- `int nextIndex()`  
returns the index of the element that would be returned by the next call to `next`.
- `int previousIndex()`  
returns the index of the element that would be returned by the next call to `previous`.

**API** `java.util.LinkedList<E>` 1.2

- `LinkedList()`  
constructs an empty linked list.
- `LinkedList(Collection<? extends E> elements)`  
constructs a linked list and adds all elements from a collection.
- `void addFirst(E element)`
- `void addLast(E element)`  
adds an element to the beginning or the end of the list.
- `E getFirst()`
- `E getLast()`  
returns the element at the beginning or the end of the list.
- `E removeFirst()`
- `E removeLast()`  
removes and returns the element at the beginning or the end of the list.

**Array Lists**

In the preceding section, you saw the `List` interface and the `LinkedList` class that implements it. The `List` interface describes an ordered collection in which the position of elements matters. There are two protocols for visiting the elements: through an iterator and by random access with methods `get` and `set`. The latter is not appropriate for linked lists, but of course `get` and `set` make a lot of sense for arrays. The collections library supplies the familiar `ArrayList` class that also implements the `List` interface. An `ArrayList` encapsulates a dynamically reallocated array of objects.



**NOTE:** If you are a veteran Java programmer, you may have used the `Vector` class whenever you needed a dynamic array. Why use an `ArrayList` instead of a `Vector`? For one simple reason: All methods of the `Vector` class are *synchronized*. It is safe to access a `Vector` object from two threads. But if you access a vector from only a single thread—by far the more common case—your code wastes quite a bit of time with synchronization. In contrast, the `ArrayList` methods are not synchronized. We recommend that you use an `ArrayList` instead of a `Vector` whenever you don't need synchronization.

**Hash Sets**

Linked lists and arrays let you specify the order in which you want to arrange the elements. However, if you are looking for a particular element and you don't remember its position, then you need to visit all elements until you find a match. That can be time consuming if the collection contains many elements. If you don't care about the ordering of the elements, then there are data structures that let you find elements much faster. The drawback is that those data structures give you no control over the order in which the elements appear. The data structures organize the elements in an order that is convenient for their own purposes.

A well-known data structure for finding objects quickly is the *hash table*. A hash table computes an integer, called the *hash code*, for each object. A hash code is an integer that is somehow derived from the instance fields of an object, preferably such that objects

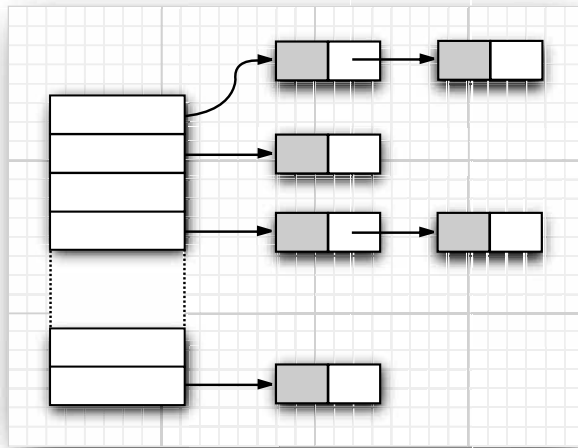
with different data yield different codes. Table 13–2 lists a few examples of hash codes that result from the `hashCode` method of the `String` class.

**Table 13–2 Hash Codes Resulting from the `hashCode` Function**

String	Hash Code
"Lee"	76268
"lee"	107020
"eel"	100300

If you define your own classes, you are responsible for implementing your own `hashCode` method—see Chapter 5 for more information. Your implementation needs to be compatible with the `equals` method: If `a.equals(b)`, then `a` and `b` must have the same hash code. What's important for now is that hash codes can be computed quickly and that the computation depends only on the state of the object that needs to be hashed, and not on the other objects in the hash table.

In Java, hash tables are implemented as arrays of linked lists. Each list is called a *bucket* (see Figure 13–8). To find the place of an object in the table, compute its hash code and reduce it modulo the total number of buckets. The resulting number is the index of the bucket that holds the element. For example, if an object has hash code 76268 and there are 128 buckets, then the object is placed in bucket 108 (because the remainder  $76268 \% 128$  is 108). Perhaps you are lucky and there is no other element in that bucket. Then, you simply insert the element into that bucket. Of course, it is inevitable that you sometimes hit a bucket that is already filled. This is called a *hash*



**Figure 13–8 A hash table**

*collision*. Then, you compare the new object with all objects in that bucket to see if it is already present. Provided that the hash codes are reasonably randomly distributed and the number of buckets is large enough, only a few comparisons should be necessary.

If you want more control over the performance of the hash table, you can specify the initial bucket count. The bucket count gives the number of buckets that are used to collect objects with identical hash values. If too many elements are inserted into a hash table, the number of collisions increases and retrieval performance suffers.

If you know approximately how many elements will eventually be in the table, then you can set the bucket count. Typically, you set it to somewhere between 75% and 150% of the expected element count. Some researchers believe that it is a good idea to make the bucket count a prime number to prevent a clustering of keys. The evidence for this isn't conclusive, however. The standard library uses bucket counts that are a power of 2, with a default of 16. (Any value you supply for the table size is automatically rounded to the next power of 2.)

Of course, you do not always know how many elements you need to store, or your initial guess may be too low. If the hash table gets too full, it needs to be *rehashed*. To rehash the table, a table with more buckets is created, all elements are inserted into the new table, and the original table is discarded. The *load factor* determines when a hash table is rehashed. For example, if the load factor is 0.75 (which is the default) and the table is more than 75% full, then it is automatically rehashed, with twice as many buckets. For most applications, it is reasonable to leave the load factor at 0.75.

Hash tables can be used to implement several important data structures. The simplest among them is the *set* type. A set is a collection of elements without duplicates. The `add` method of a set first tries to find the object to be added, and adds it only if it is not yet present.

The Java collections library supplies a `HashSet` class that implements a set based on a hash table. You add elements with the `add` method. The `contains` method is redefined to make a fast lookup to find if an element is already present in the set. It checks only the elements in one bucket and not all elements in the collection.

The hash set iterator visits all buckets in turn. Because the hashing scatters the elements around in the table, they are visited in seemingly random order. You would only use a `HashSet` if you don't care about the ordering of the elements in the collection.

The sample program at the end of this section (Listing 13–2) reads words from `System.in`, adds them to a set, and finally prints out all words in the set. For example, you can feed the program the text from *Alice in Wonderland* (which you can obtain from <http://www.gutenberg.net>) by launching it from a command shell as

```
java SetTest < alice30.txt
```

The program reads all words from the input and adds them to the hash set. It then iterates through the unique words in the set and finally prints out a count. (*Alice in Wonderland* has 5,909 unique words, including the copyright notice at the beginning.) The words appear in random order.



**CAUTION:** Be careful when you mutate set elements. If the hash code of an element were to change, then the element would no longer be in the correct position in the data structure.

**Listing 13-2** SetTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program uses a set to print all unique words in System.in.
5.  * @version 1.10 2003-08-02
6.  * @author Cay Horstmann
7.  */
8. public class SetTest
9. {
10.     public static void main(String[] args)
11.     {
12.         Set<String> words = new HashSet<String>(); // HashSet implements Set
13.         long totalTime = 0;
14.
15.         Scanner in = new Scanner(System.in);
16.         while (in.hasNext())
17.         {
18.             String word = in.next();
19.             long callTime = System.currentTimeMillis();
20.             words.add(word);
21.             callTime = System.currentTimeMillis() - callTime;
22.             totalTime += callTime;
23.         }
24.
25.         Iterator<String> iter = words.iterator();
26.         for (int i = 1; i <= 20; i++)
27.             System.out.println(iter.next());
28.         System.out.println(". . .");
29.         System.out.println(words.size() + " distinct words. " + totalTime + " milliseconds.");
30.     }
31. }
```

**API** java.util.HashSet<E> 1.2

- HashSet()  
constructs an empty hash set.
- HashSet(Collection<? extends E> elements)  
constructs a hash set and adds all elements from a collection.
- HashSet(int initialCapacity)  
constructs an empty hash set with the specified capacity (number of buckets).
- HashSet(int initialCapacity, float loadFactor)  
constructs an empty hash set with the specified capacity and load factor (a number between 0.0 and 1.0 that determines at what percentage of fullness the hash table will be rehashed into a larger one).

**API** `java.lang.Object` 1.0

- `int hashCode()`  
returns a hash code for this object. A hash code can be any integer, positive or negative. The definitions of `equals` and `hashCode` must be compatible: If `x.equals(y)` is true, then `x.hashCode()` must be the same value as `y.hashCode()`.

**Tree Sets**

The `TreeSet` class is similar to the hash set, with one added improvement. A tree set is a *sorted collection*. You insert elements into the collection in any order. When you iterate through the collection, the values are automatically presented in sorted order. For example, suppose you insert three strings and then visit all elements that you added.

```
SortedSet<String> sorter = new TreeSet<String>(); // TreeSet implements SortedSet
sorter.add("Bob");
sorter.add("Amy");
sorter.add("Carl");
for (String s : sorter) System.println(s);
```

Then, the values are printed in sorted order: Amy Bob Carl. As the name of the class suggests, the sorting is accomplished by a tree data structure. (The current implementation uses a *red-black tree*. For a detailed description of red-black trees, see, for example, *Introduction to Algorithms* by Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein [The MIT Press, 2001].) Every time an element is added to a tree, it is placed into its proper sorting position. Therefore, the iterator always visits the elements in sorted order.

Adding an element to a tree is slower than adding it to a hash table, but it is still much faster than adding it into the right place in an array or linked list. If the tree contains  $n$  elements, then an average of  $\log_2 n$  comparisons are required to find the correct position for the new element. For example, if the tree already contains 1,000 elements, then adding a new element requires about 10 comparisons.

Thus, adding elements into a `TreeSet` is somewhat slower than adding into a `HashSet`—see Table 13-3 for a comparison—but the `TreeSet` automatically sorts the elements.

**Table 13-3** Adding Elements into Hash and Tree Sets

Document	Total Number of Words	Number of Distinct Words	HashSet	TreeSet
<i>Alice in Wonderland</i>	28195	5909	5 sec	7 sec
<i>The Count of Monte Cristo</i>	466300	37545	75 sec	98 sec

**API** `java.util.TreeSet<E>` 1.2

- `TreeSet()`  
constructs an empty tree set.
- `TreeSet(Collection<? extends E> elements)`  
constructs a tree set and adds all elements from a collection.



### Object Comparison

How does the `TreeSet` know how you want the elements sorted? By default, the tree set assumes that you insert elements that implement the `Comparable` interface. That interface defines a single method:

```
public interface Comparable<T>
{
    int compareTo(T other);
}
```

The call `a.compareTo(b)` must return 0 if `a` and `b` are equal, a negative integer if `a` comes before `b` in the sort order, and a positive integer if `a` comes after `b`. The exact value does not matter; only its sign ( $>0$ , 0, or  $<0$ ) matters. Several standard Java platform classes implement the `Comparable` interface. One example is the `String` class. Its `compareTo` method compares strings in dictionary order (sometimes called *lexicographic order*).

If you insert your own objects, you must define a sort order yourself by implementing the `Comparable` interface. There is no default implementation of `compareTo` in the `Object` class.

For example, here is how you can sort `Item` objects by part number:

```
class Item implements Comparable<Item>
{
    public int compareTo(Item other)
    {
        return partNumber - other.partNumber;
    }
    . . .
}
```

If you compare two *positive* integers, such as part numbers in our example, then you can simply return their difference—it will be negative if the first item should come before the second item, zero if the part numbers are identical, and positive otherwise.



**CAUTION:** This trick only works if the integers are from a small enough range. If  $x$  is a large positive integer and  $y$  is a large negative integer, then the difference  $x - y$  can overflow.

However, using the `Comparable` interface for defining the sort order has obvious limitations. A given class can implement the interface only once. But what can you do if you need to sort a bunch of items by part number in one collection and by description in another? Furthermore, what can you do if you need to sort objects of a class whose creator didn't bother to implement the `Comparable` interface?

In those situations, you tell the tree set to use a different comparison method, by passing a `Comparator` object into the `TreeSet` constructor. The `Comparator` interface declares a `compare` method with two explicit parameters:

```
public interface Comparator<T>
{
    int compare(T a, T b);
}
```

Just like the `compareTo` method, the `compare` method returns a negative integer if `a` comes before `b`, zero if they are identical, or a positive integer otherwise.

To sort items by their description, simply define a class that implements the `Comparator` interface:

```
class ItemComparator implements Comparator<Item>
{
    public int compare(Item a, Item b)
    {
        String descrA = a.getDescription();
        String descrB = b.getDescription();
        return descrA.compareTo(descrB);
    }
}
```

You then pass an object of this class to the tree set constructor:

```
ItemComparator comp = new ItemComparator();
SortedSet<Item> sortByDescription = new TreeSet<Item>(comp);
```

If you construct a tree with a comparator, it uses this object whenever it needs to compare two elements.

Note that this item comparator has no data. It is just a holder for the comparison method. Such an object is sometimes called a *function object*.

Function objects are commonly defined “on the fly,” as instances of anonymous inner classes:

```
SortedSet<Item> sortByDescription = new TreeSet<Item>(new
    Comparator<Item>()
    {
        public int compare(Item a, Item b)
        {
            String descrA = a.getDescription();
            String descrB = b.getDescription();
            return descrA.compareTo(descrB);
        }
    });
```



**NOTE:** Actually, the `Comparator<T>` interface is declared to have two methods: `compare` and `equals`. Of course, every class has an `equals` method; thus, there seems little benefit in adding the method to the interface declaration. The API documentation explains that you need not override the `equals` method but that doing so may yield improved performance in some cases. For example, the `addAll` method of the `TreeSet` class can work more effectively if you add elements from another set that uses the same comparator.

If you look back at Table 13–3, you may well wonder if you should always use a tree set instead of a hash set. After all, adding elements does not seem to take much longer, and the elements are automatically sorted. The answer depends on the data that you are collecting. If you don’t need the data sorted, there is no reason to pay for the sorting overhead. More important, with some data it is much more difficult to come up with a sort order than a hash function. A hash function only needs to do a reasonably good job of scrambling the objects, whereas a comparison function must tell objects apart with complete precision.

To make this distinction more concrete, consider the task of collecting a set of rectangles. If you use a `TreeSet`, you need to supply a `Comparator<Rectangle>`. How do you compare two rectangles? By area? That doesn't work. You can have two different rectangles with different coordinates but the same area. The sort order for a tree must be a *total ordering*. Any two elements must be comparable, and the comparison can only be zero if the elements are equal. There is such a sort order for rectangles (the lexicographic ordering on its coordinates), but it is unnatural and cumbersome to compute. In contrast, a hash function is already defined for the `Rectangle` class. It simply hashes the coordinates.



NOTE: As of Java SE 6, the `TreeSet` class implements the `NavigableSet` interface. That interface adds several convenient methods for locating elements, and for backward traversal. See the API notes for details.

The program in Listing 13-3 builds two tree sets of `Item` objects. The first one is sorted by part number, the default sort order of `Item` objects. The second set is sorted by description, by means of a custom comparator.

**Listing 13-3** `TreeSetTest.java`

```
1. /**
2.  @version 1.10 2004-08-02
3.  @author Cay Horstmann
4. */
5.
6. import java.util.*;
7.
8. /**
9.  This program sorts a set of items by comparing
10. their descriptions.
11. */
12. public class TreeSetTest
13. {
14.     public static void main(String[] args)
15.     {
16.         SortedSet<Item> parts = new TreeSet<Item>();
17.         parts.add(new Item("Toaster", 1234));
18.         parts.add(new Item("Widget", 4562));
19.         parts.add(new Item("Modem", 9912));
20.         System.out.println(parts);
21.
22.         SortedSet<Item> sortByDescription = new TreeSet<Item>(new
23.             Comparator<Item>()
24.             {
25.                 public int compare(Item a, Item b)
26.                 {
27.                     String descrA = a.getDescription();
28.                     String descrB = b.getDescription();
29.                     return descrA.compareTo(descrB);
```

**Listing 13-3** TreeSetTest.java (continued)

```
30.     }
31.     });
32.
33.     sortByDescription.addAll(parts);
34.     System.out.println(sortByDescription);
35. }
36. }
37.
38. /**
39.  * An item with a description and a part number.
40.  */
41. class Item implements Comparable<Item>
42. {
43.     /**
44.      * Constructs an item.
45.      * @param aDescription the item's description
46.      * @param aPartNumber the item's part number
47.      */
48.     public Item(String aDescription, int aPartNumber)
49.     {
50.         description = aDescription;
51.         partNumber = aPartNumber;
52.     }
53.
54.     /**
55.      * Gets the description of this item.
56.      * @return the description
57.      */
58.     public String getDescription()
59.     {
60.         return description;
61.     }
62.
63.     public String toString()
64.     {
65.         return "[description=" + description
66.             + ", partNumber=" + partNumber + "];"
67.     }
68.
69.     public boolean equals(Object otherObject)
70.     {
71.         if (this == otherObject) return true;
72.         if (otherObject == null) return false;
73.         if (getClass() != otherObject.getClass()) return false;
74.         Item other = (Item) otherObject;
75.         return description.equals(other.description)
76.             && partNumber == other.partNumber;
77.     }
78.
```

**Listing 13-3** TreeSetTest.java (continued)

```
79. public int hashCode()
80. {
81.     return 13 * description.hashCode() + 17 * partNumber;
82. }
83.
84. public int compareTo(Item other)
85. {
86.     return partNumber - other.partNumber;
87. }
88.
89. private String description;
90. private int partNumber;
91. }
```

**API** java.lang.Comparable<T> 1.2

- `int compareTo(T other)`  
compares this object with another object and returns a negative value if this comes before `other`, zero if they are considered identical in the sort order, and a positive value if this comes after `other`.

**API** java.util.Comparator<T> 1.2

- `int compare(T a, T b)`  
compares two objects and returns a negative value if `a` comes before `b`, zero if they are considered identical in the sort order, and a positive value if `a` comes after `b`.

**API** java.util.SortedSet<E> 1.2

- `Comparator<? super E> comparator()`  
returns the comparator used for sorting the elements, or `null` if the elements are compared with the `compareTo` method of the `Comparable` interface.
- `E first()`
- `E last()`  
returns the smallest or largest element in the sorted set.

**API** java.util.NavigableSet<E> 6

- `E higher(E value)`
- `E lower(E value)`  
returns the least element  $>$  `value` or the largest element  $<$  `value`, or `null` if there is no such element.
- `E ceiling(E value)`
- `E floor(E value)`  
returns the least element  $\geq$  `value` or the largest element  $\leq$  `value`, or `null` if there is no such element.

- `E pollFirst()`
- `E pollLast()`  
removes and returns the smallest or largest element in this set, or `null` if the set is empty.
- `Iterator<E> descendingIterator()`  
returns an iterator that traverses this set in descending direction.

**API** `java.util.TreeSet<E>` 1.2

- `TreeSet()`  
constructs a tree set for storing `Comparable` objects.
- `TreeSet(Comparator<? super E> c)`  
constructs a tree set and uses the specified comparator for sorting its elements.
- `TreeSet(SortedSet<? extends E> elements)`  
constructs a tree set, adds all elements from a sorted set, and uses the same element comparator as the given sorted set.

**Queues and Deques**

As we already discussed, a queue lets you efficiently add elements at the tail and remove elements from the head. A double ended queue or *deque* lets you efficiently add or remove elements at the head and tail. Adding elements in the middle is not supported. Java SE 6 introduced a `Deque` interface. It is implemented by the `ArrayDeque` and `LinkedList` classes, both of which provide deques whose size grows as needed. In Chapter 14, you will see bounded queues and deques.

**API** `java.util.Queue<E>` 5.0

- `boolean add(E element)`
- `boolean offer(E element)`  
adds the given element to the tail of this deque and returns `true`, provided the queue is not full. If the queue is full, the first method throws an `IllegalStateException`, whereas the second method returns `false`
- `E remove()`
- `E poll()`  
removes and returns the element at the head of this queue, provided the queue is not empty. If the queue is empty, the first method throws a `NoSuchElementException`, whereas the second method returns `null`.
- `E element()`
- `E peek()`  
returns the element at the head of this queue without removing it, provided the queue is not empty. If the queue is empty, the first method throws a `NoSuchElementException`, whereas the second method returns `null`.

**API** `java.util.Deque<E>` 6

- `void addFirst(E element)`
  - `void addLast(E element)`
  - `boolean offerFirst(E element)`
  - `boolean offerLast(E element)`
- adds the given element to the head or tail of this deque. If the queue is full, the first two methods throw an `IllegalStateException`, whereas the last two methods return `false`.
- `E removeFirst()`
  - `E removeLast()`
  - `E pollFirst()`
  - `E pollLast()`
- removes and returns the element at the head of this queue, provided the queue is not empty. If the queue is empty, the first two methods throw a `NoSuchElementException`, whereas the last two methods return `null`.
- `E getFirst()`
  - `E getLast()`
  - `E peekFirst()`
  - `E peekLast()`
- returns the element at the head of this queue without removing it, provided the queue is not empty. If the queue is empty, the first two methods throw a `NoSuchElementException`, whereas the last two methods return `null`.

**API** `java.util.ArrayDeque<E>` 6

- `ArrayDeque()`
  - `ArrayDeque(int initialCapacity)`
- constructs unbounded deques with an initial capacity of 16 or the given initial capacity.

**Priority Queues**

A priority queue retrieves elements in sorted order after they were inserted in arbitrary order. That is, whenever you call the `remove` method, you get the smallest element currently in the priority queue. However, the priority queue does not sort all its elements. If you iterate over the elements, they are not necessarily sorted. The priority queue makes use of an elegant and efficient data structure, called a *heap*. A heap is a self-organizing binary tree in which the `add` and `remove` operations cause the smallest element to gravitate to the root, without wasting time on sorting all elements.

Just like a `TreeSet`, a priority queue can either hold elements of a class that implements the `Comparable` interface or a `Comparator` object you supply in the constructor.

A typical use for a priority queue is job scheduling. Each job has a priority. Jobs are added in random order. Whenever a new job can be started, the highest-priority job is removed from the queue. (Since it is traditional for priority 1 to be the “highest” priority, the `remove` operation yields the minimum element.)

Listing 13–4 shows a priority queue in action. Unlike iteration in a `TreeSet`, the iteration here does not visit the elements in sorted order. However, removal always yields the smallest remaining element.

**Listing 13–4** PriorityQueueTest.java

```

1. import java.util.*;
2.
3. /**
4.  * This program demonstrates the use of a priority queue.
5.  * @version 1.00 2004-08-03
6.  * @author Cay Horstmann
7.  */
8. public class PriorityQueueTest
9. {
10.     public static void main(String[] args)
11.     {
12.         PriorityQueue<GregorianCalendar> pq = new PriorityQueue<GregorianCalendar>();
13.         pq.add(new GregorianCalendar(1906, Calendar.DECEMBER, 9)); // G. Hopper
14.         pq.add(new GregorianCalendar(1815, Calendar.DECEMBER, 10)); // A. Lovelace
15.         pq.add(new GregorianCalendar(1903, Calendar.DECEMBER, 3)); // J. von Neumann
16.         pq.add(new GregorianCalendar(1910, Calendar.JUNE, 22)); // K. Zuse
17.
18.         System.out.println("Iterating over elements...");
19.         for (GregorianCalendar date : pq)
20.             System.out.println(date.get(Calendar.YEAR));
21.         System.out.println("Removing elements...");
22.         while (!pq.isEmpty())
23.             System.out.println(pq.remove().get(Calendar.YEAR));
24.     }
25. }

```

**API** `java.util.PriorityQueue 5.0`

- `PriorityQueue()`
- `PriorityQueue(int initialCapacity)`  
constructs a priority queue for storing `Comparable` objects.
- `PriorityQueue(int initialCapacity, Comparator<? super E> c)`  
constructs a priority queue and uses the specified comparator for sorting its elements.

### Maps

A set is a collection that lets you quickly find an existing element. However, to look up an element, you need to have an exact copy of the element to find. That isn't a very common lookup—usually, you have some key information, and you want to look up the associated element. The *map* data structure serves that purpose. A map stores key/value pairs. You can find a value if you provide the key. For example, you may store a table of employee records, where the keys are the employee IDs and the values are `Employee` objects.



The Java library supplies two general-purpose implementations for maps: `HashMap` and `TreeMap`. Both classes implement the `Map` interface.

A hash map hashes the keys, and a tree map uses a total ordering on the keys to organize them in a search tree. The hash or comparison function is applied *only to the keys*. The values associated with the keys are not hashed or compared.

Should you choose a hash map or a tree map? As with sets, hashing is a bit faster, and it is the preferred choice if you don't need to visit the keys in sorted order.

Here is how you set up a hash map for storing employees:

```
Map<String, Employee> staff = new HashMap<String, Employee>(); // HashMap implements Map
Employee harry = new Employee("Harry Hacker");
staff.put("987-98-9996", harry);
. . .
```

Whenever you add an object to a map, you must supply a key as well. In our case, the key is a string, and the corresponding value is an `Employee` object.

To retrieve an object, you must use (and, therefore, remember) the key.

```
String s = "987-98-9996";
e = staff.get(s); // gets harry
```

If no information is stored in the map with the particular key specified, then `get` returns `null`.

Keys must be unique. You cannot store two values with the same key. If you call the `put` method twice with the same key, then the second value replaces the first one. In fact, `put` returns the previous value stored with the key parameter.

The `remove` method removes an element with a given key from the map. The `size` method returns the number of entries in the map.

The collections framework does not consider a map itself as a collection. (Other frameworks for data structures consider a map as a collection of *pairs*, or as a collection of values that is indexed by the keys.) However, you can obtain *views* of the map, objects that implement the `Collection` interface, or one of its subinterfaces.

There are three views: the set of keys, the collection of values (which is not a set), and the set of key/value pairs. The keys and key/value pairs form a set because there can be only one copy of a key in a map. The methods

```
Set<K> keySet()
Collection<V> values()
Set<Map.Entry<K, V>> entrySet()
```

return these three views. (The elements of the entry set are objects of the static inner class `Map.Entry`.)

Note that the `keySet` is *not* a `HashSet` or `TreeSet`, but an object of some other class that implements the `Set` interface. The `Set` interface extends the `Collection` interface. Therefore, you can use a `keySet` as you would use any collection.

For example, you can enumerate all keys of a map:

```
Set<String> keys = map.keySet();
for (String key : keys)
```

```

{
    do something with key
}

```



TIP: If you want to look at both keys and values, then you can avoid value lookups by enumerating the *entries*. Use the following code skeleton:

```

for (Map.Entry<String, Employee> entry : staff.entrySet())
{
    String key = entry.getKey();
    Employee value = entry.getValue();
    do something with key, value
}

```

If you invoke the `remove` method of the iterator, you actually remove the *key and its associated value* from the map. However, you cannot *add* an element to the key set view. It makes no sense to add a key without also adding a value. If you try to invoke the `add` method, it throws an `UnsupportedOperationException`. The entry set view has the same restriction, even though it would make conceptual sense to add a new key/value pair.

Listing 13–5 illustrates a map at work. We first add key/value pairs to a map. Then, we remove one key from the map, which removes its associated value as well. Next, we change the value that is associated with a key and call the `get` method to look up a value. Finally, we iterate through the entry set.

#### Listing 13–5 MapTest.java

```

1. import java.util.*;
2.
3. /**
4.  * This program demonstrates the use of a map with key type String and value type Employee.
5.  * @version 1.10 2004-08-02
6.  * @author Cay Horstmann
7.  */
8. public class MapTest
9. {
10.     public static void main(String[] args)
11.     {
12.         Map<String, Employee> staff = new HashMap<String, Employee>();
13.         staff.put("144-25-5464", new Employee("Amy Lee"));
14.         staff.put("567-24-2546", new Employee("Harry Hacker"));
15.         staff.put("157-62-7935", new Employee("Gary Cooper"));
16.         staff.put("456-62-5527", new Employee("Francesca Cruz"));
17.
18.         // print all entries
19.
20.         System.out.println(staff);
21.
22.         // remove an entry
23.

```

**Listing 13-5** MapTest.java (continued)

```
24.     staff.remove("567-24-2546");
25.
26.     // replace an entry
27.
28.     staff.put("456-62-5527", new Employee("Francesca Miller"));
29.
30.     // look up a value
31.
32.     System.out.println(staff.get("157-62-7935"));
33.
34.     // iterate through all entries
35.
36.     for (Map.Entry<String, Employee> entry : staff.entrySet())
37.     {
38.         String key = entry.getKey();
39.         Employee value = entry.getValue();
40.         System.out.println("key=" + key + ", value=" + value);
41.     }
42. }
43. }
44.
45. /**
46.  * A minimalist employee class for testing purposes.
47.  */
48. class Employee
49. {
50.     /**
51.     * Constructs an employee with $0 salary.
52.     * @param n the employee name
53.     */
54.     public Employee(String n)
55.     {
56.         name = n;
57.         salary = 0;
58.     }
59.
60.     public String toString()
61.     {
62.         return "[name=" + name + ", salary=" + salary + "];"
63.     }
64.
65.     private String name;
66.     private double salary;
67. }
```

**API** `java.util.Map<K, V>` 1.2

- `V get(K key)`  
gets the value associated with the key; returns the object associated with the key, or `null` if the key is not found in the map. The key may be `null`.
- `V put(K key, V value)`  
puts the association of a key and a value into the map. If the key is already present, the new object replaces the old one previously associated with the key. This method returns the old value of the key, or `null` if the key was not previously present. The key may be `null`, but the value must not be `null`.
- `void putAll(Map<? extends K, ? extends V> entries)`  
adds all entries from the specified map to this map.
- `boolean containsKey(Object key)`  
returns `true` if the key is present in the map.
- `boolean containsValue(Object value)`  
returns `true` if the value is present in the map.
- `Set<Map.Entry<K, V>> entrySet()`  
returns a set view of `Map.Entry` objects, the key/value pairs in the map. You can remove elements from this set and they are removed from the map, but you cannot add any elements.
- `Set<K> keySet()`  
returns a set view of all keys in the map. You can remove elements from this set and the keys and associated values are removed from the map, but you cannot add any elements.
- `Collection<V> values()`  
returns a collection view of all values in the map. You can remove elements from this set and the removed value and its key are removed from the map, but you cannot add any elements.

**API** `java.util.Map.Entry<K, V>` 1.2

- `K getKey()`
- `V getValue()`  
returns the key or value of this entry.
- `V setValue(V newValue)`  
changes the value *in the associated map* to the new value and returns the old value.

**API** `java.util.HashMap<K, V>` 1.2

- `HashMap()`
- `HashMap(int initialCapacity)`
- `HashMap(int initialCapacity, float loadFactor)`  
constructs an empty hash map with the specified capacity and load factor (a number between 0.0 and 1.0 that determines at what percentage of fullness the hash table will be rehashed into a larger one). The default load factor is 0.75.

**API** `java.util.TreeMap<K,V>` 1.2

- `TreeMap(Comparator<? super K> c)`  
constructs a tree map and uses the specified comparator for sorting its keys.
- `TreeMap(Map<? extends K, ? extends V> entries)`  
constructs a tree map and adds all entries from a map.
- `TreeMap(SortedMap<? extends K, ? extends V> entries)`  
constructs a tree map, adds all entries from a sorted map, and uses the same element comparator as the given sorted map.

**API** `java.util.SortedMap<K, V>` 1.2

- `Comparator<? super K> comparator()`  
returns the comparator used for sorting the keys, or `null` if the keys are compared with the `compareTo` method of the `Comparable` interface.
- `K firstKey()`
- `K lastKey()`  
returns the smallest or largest key in the map.

**Specialized Set and Map Classes**

The collection class library has several map classes for specialized needs that we briefly discuss in this section.

**Weak Hash Maps**

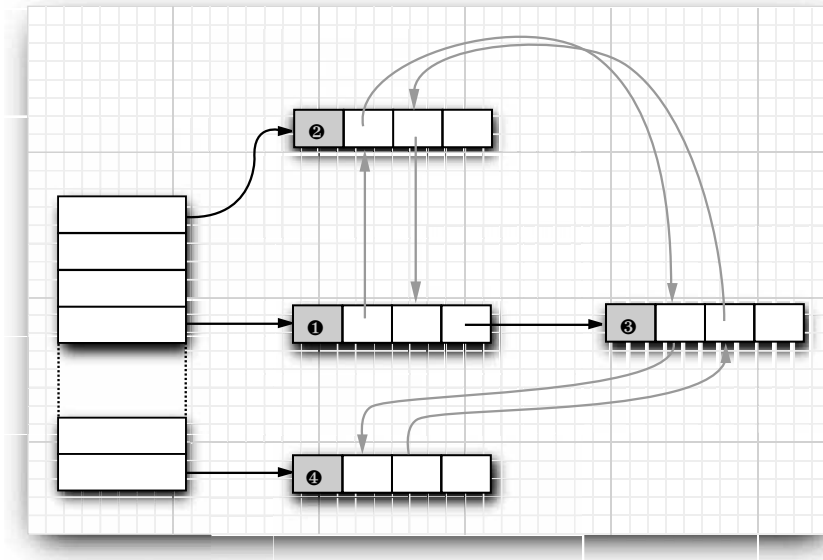
The `WeakHashMap` class was designed to solve an interesting problem. What happens with a value whose key is no longer used anywhere in your program? Suppose the last reference to a key has gone away. Then, there is no longer any way to refer to the value object. But because no part of the program has the key any more, the key/value pair cannot be removed from the map. Why can't the garbage collector remove it? Isn't it the job of the garbage collector to remove unused objects?

Unfortunately, it isn't quite so simple. The garbage collector traces *live* objects. As long as the map object is live, then *all* buckets in it are live and they won't be reclaimed. Thus, your program should take care to remove unused values from long-lived maps. Or, you can use a `WeakHashMap` instead. This data structure cooperates with the garbage collector to remove key/value pairs when the only reference to the key is the one from the hash table entry.

Here are the inner workings of this mechanism. The `WeakHashMap` uses *weak references* to hold keys. A `WeakReference` object holds a reference to another object, in our case, a hash table key. Objects of this type are treated in a special way by the garbage collector. Normally, if the garbage collector finds that a particular object has no references to it, it simply reclaims the object. However, if the object is reachable *only* by a `WeakReference`, the garbage collector still reclaims the object, but it places the weak reference that led to it into a queue. The operations of the `WeakHashMap` periodically check that queue for newly arrived weak references. The arrival of a weak reference in the queue signifies that the key was no longer used by anyone and that it has been collected. The `WeakHashMap` then removes the associated entry.

**Linked Hash Sets and Maps**

Java SE 1.4 added classes `LinkedHashSet` and `LinkedHashMap` that remember in which order you inserted items. That way, you avoid the seemingly random order of items in a hash table. As entries are inserted into the table, they are joined in a doubly linked list (see Figure 13–9).



**Figure 13–9** A linked hash table

For example, consider the following map insertions from Listing 13–5:

```
Map staff = new LinkedHashMap();
staff.put("144-25-5464", new Employee("Amy Lee"));
staff.put("567-24-2546", new Employee("Harry Hacker"));
staff.put("157-62-7935", new Employee("Gary Cooper"));
staff.put("456-62-5527", new Employee("Francesca Cruz"));
```

Then, `staff.keySet().iterator()` enumerates the keys in this order:

```
144-25-5464
567-24-2546
157-62-7935
456-62-5527
```

and `staff.values().iterator()` enumerates the values in this order:

```
Amy Lee
Harry Hacker
Gary Cooper
Francesca Cruz
```

A linked hash map can alternatively use *access order*, not insertion order, to iterate through the map entries. Every time you call `get` or `put`, the affected entry is removed from its current position and placed at the *end* of the linked list of entries. (Only the position in the linked list of entries is affected, not the hash table bucket. An entry always stays in the bucket that corresponds to the hash code of the key.) To construct such a hash map, call

```
LinkedHashMap<K, V>(initialCapacity, loadFactor, true)
```

Access order is useful for implementing a “least recently used” discipline for a cache. For example, you may want to keep frequently accessed entries in memory and read less frequently accessed objects from a database. When you don’t find an entry in the table, and the table is already pretty full, then you can get an iterator into the table and remove the first few elements that it enumerates. Those entries were the least recently used ones.

You can even automate that process. Form a subclass of `LinkedHashMap` and override the method

```
protected boolean removeEldestEntry(Map.Entry<K, V> eldest)
```

Adding a new entry then causes the `eldest` entry to be removed whenever your method returns `true`. For example, the following cache is kept at a size of at most 100 elements:

```
Map<K, V> cache = new
    LinkedHashMap<K, V>(128, 0.75F, true)
    {
        protected boolean removeEldestEntry(Map.Entry<K, V> eldest)
        {
            return size() > 100;
        }
    };
```

Alternatively, you can consider the `eldest` entry to decide whether to remove it. For example, you may want to check a time stamp stored with the entry.

### Enumeration Sets and Maps

The `EnumSet` is an efficient set implementation with elements that belong to an enumerated type. Because an enumerated type has a finite number of instances, the `EnumSet` is internally implemented simply as a sequence of bits. A bit is turned on if the corresponding value is present in the set.

The `EnumSet` class has no public constructors. You use a static factory method to construct the set:

```
enum Weekday { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY };
EnumSet<Weekday> always = EnumSet.allOf(Weekday.class);
EnumSet<Weekday> never = EnumSet.noneOf(Weekday.class);
EnumSet<Weekday> workday = EnumSet.range(Weekday.MONDAY, Weekday.FRIDAY);
EnumSet<Weekday> mwf = EnumSet.of(Weekday.MONDAY, Weekday.WEDNESDAY, Weekday.FRIDAY);
```

You can use the usual methods of the `Set` interface to modify an `EnumSet`.

An `EnumMap` is a map with keys that belong to an enumerated type. It is simply and efficiently implemented as an array of values. You need to specify the key type in the constructor:

```
EnumMap<Weekday, Employee> personInCharge = new EnumMap<Weekday, Employee>(Weekday.class);
```



NOTE: In the API documentation for `EnumSet`, you will see odd-looking type parameters of the form `E extends Enum<E>`. This simply means “E is an enumerated type.” All enumerated types extend the generic `Enum` class. For example, `Weekday` extends `Enum<Weekday>`.

### Identity Hash Maps

Java SE 1.4 added another class `IdentityHashMap` for another quite specialized purpose, where the hash values for the keys should not be computed by the `hashCode` method but by the `System.identityHashCode` method. That’s the method that `Object.hashCode` uses to compute a hash code from the object’s memory address. Also, for comparison of objects, the `IdentityHashMap` uses `==`, not `equals`.

In other words, different key objects are considered distinct even if they have equal contents. This class is useful for implementing object traversal algorithms (such as object serialization), in which you want to keep track of which objects have already been traversed.

**API** `java.util.WeakHashMap<K, V>` 1.2

- `WeakHashMap()`
- `WeakHashMap(int initialCapacity)`
- `WeakHashMap(int initialCapacity, float loadFactor)`  
constructs an empty hash map with the specified capacity and load factor.

**API** `java.util.LinkedHashSet<E>` 1.4

- `LinkedHashSet()`
- `LinkedHashSet(int initialCapacity)`
- `LinkedHashSet(int initialCapacity, float loadFactor)`  
constructs an empty linked hash set with the specified capacity and load factor.

**API** `java.util.LinkedHashMap<K, V>` 1.4

- `LinkedHashMap()`
- `LinkedHashMap(int initialCapacity)`
- `LinkedHashMap(int initialCapacity, float loadFactor)`
- `LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)`  
constructs an empty linked hash map with the specified capacity, load factor, and ordering. The `accessOrder` parameter is `true` for access order, `false` for insertion order.
- protected boolean `removeEldestEntry(Map.Entry<K, V> eldest)`  
should be overridden to return `true` if you want the `eldest` entry to be removed. The `eldest` parameter is the entry whose removal is being contemplated. This method is called after an entry has been added to the map. The default implementation returns `false`—old elements are not removed by default. However, you can redefine this method to selectively return `true`; for example, if the `eldest` entry fits a certain condition or the map exceeds a certain size.



**API** `java.util.EnumSet<E extends Enum<E>> 5.0`

- `static <E extends Enum<E>> EnumSet<E> allOf(Class<E> enumType)`  
returns a set that contains all values of the given enumerated type.
- `static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> enumType)`  
returns an empty set, capable of holding values of the given enumerated type.
- `static <E extends Enum<E>> EnumSet<E> range(E from, E to)`  
returns a set that contains all values between from and to (inclusive).
- `static <E extends Enum<E>> EnumSet<E> of(E value)`
- `static <E extends Enum<E>> EnumSet<E> of(E value, E... values)`  
returns a set that contains the given values.

**API** `java.util.EnumMap<K extends Enum<K>, V> 5.0`

- `EnumMap(Class<K> keyType)`  
constructs an empty map whose keys have the given type.

**API** `java.util.IdentityHashMap<K, V> 1.4`

- `IdentityHashMap()`
- `IdentityHashMap(int expectedMaxSize)`  
constructs an empty identity hash map whose capacity is the smallest power of 2 exceeding  $1.5 * \text{expectedMaxSize}$ . (The default for `expectedMaxSize` is 21.)

**API** `java.lang.System 1.0`

- `static int identityHashCode(Object obj) 1.1`  
returns the same hash code (derived from the object's memory address) that `Object.hashCode` computes, even if the class to which `obj` belongs has redefined the `hashCode` method.

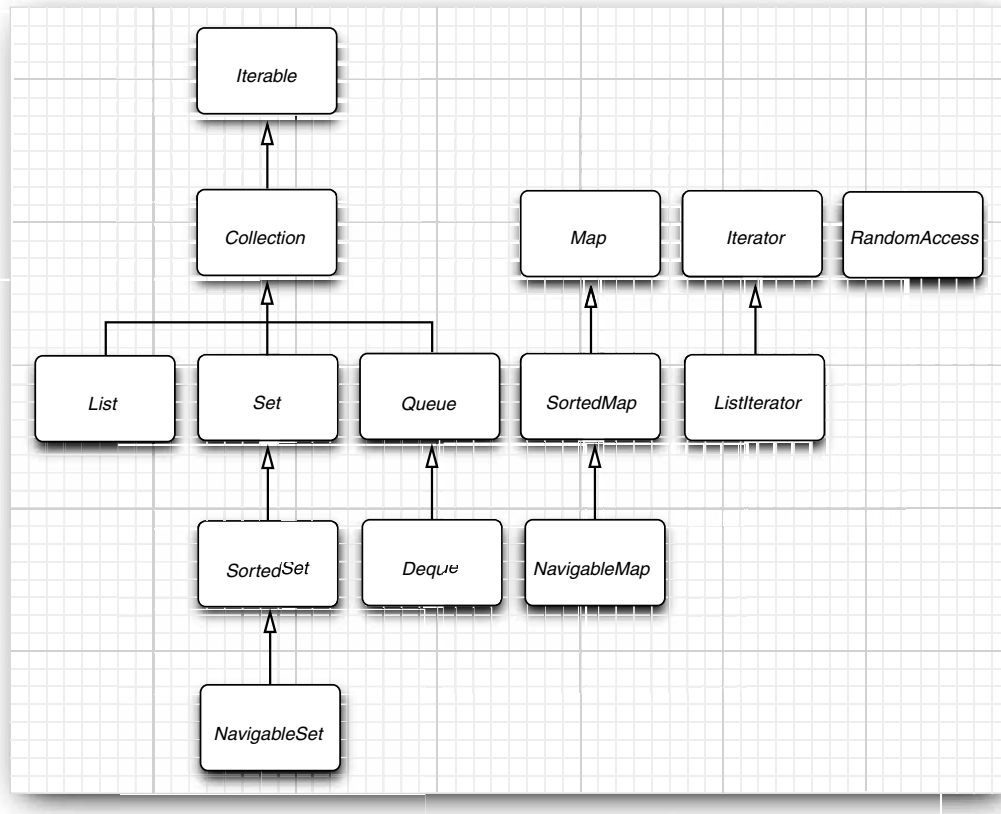
**The Collections Framework**

A *framework* is a set of classes that form the basis for building advanced functionality. A framework contains superclasses with useful functionality, policies, and mechanisms. The user of a framework forms subclasses to extend the functionality without having to reinvent the basic mechanisms. For example, Swing is a framework for user interfaces.

The Java collections library forms a framework for collection classes. It defines a number of interfaces and abstract classes for implementors of collections (see Figure 13–10), and it prescribes certain mechanisms, such as the iteration protocol. You can use the collection classes without having to know much about the framework—we did just that in the preceding sections. However, if you want to implement generic algorithms that work for multiple collection types or if you want to add a new collection type, it is helpful to understand the framework.

There are two fundamental interfaces for collections: `Collection` and `Map`. You insert elements into a collection with a method:

```
boolean add(E element)
```



**Figure 13–10 The interfaces of the collections framework**

However, maps hold key/value pairs, and you use the `put` method to insert them.

```
V put(K key, V value)
```

To read elements from a collection, you visit them with an iterator. However, you can read values from a map with the `get` method:

```
V get(K key)
```

A `List` is an *ordered collection*. Elements are added into a particular position in the container. An object can be placed into its position in two ways: by an integer index and by a list iterator. The `List` interface defines methods for random access:

```
void add(int index, E element)
E get(int index)
void remove(int index)
```

As already discussed, the `List` interface provides these random access methods whether or not they are efficient for a particular implementation. To avoid carrying out costly

random access operations, Java SE 1.4 introduced a tagging interface, `RandomAccess`. That interface has no methods, but you can use it to test whether a particular collection supports efficient random access:

```
if (c instanceof RandomAccess)
{
    use random access algorithm
}
else
{
    use sequential access algorithm
}
```

The `ArrayList` and `Vector` classes implement the `RandomAccess` interface.



**NOTE:** From a theoretical point of view, it would have made sense to have a separate `Array` interface that extends the `List` interface and declares the random access methods. If there were a separate `Array` interface, then those algorithms that require random access would use `Array` parameters and you could not accidentally apply them to collections with slow random access. However, the designers of the collections framework chose not to define a separate interface, because they wanted to keep the number of interfaces in the library small. Also, they did not want to take a paternalistic attitude toward programmers. You are free to pass a linked list to algorithms that use random access—you just need to be aware of the performance costs.

The `ListIterator` interface defines a method for adding an element before the iterator position:

```
void add(E element)
```

To get and remove elements at a particular position, you simply use the `next` and `remove` methods of the `Iterator` interface.

The `Set` interface is identical to the `Collection` interface, but the behavior of the methods is more tightly defined. The `add` method of a set should reject duplicates. The `equals` method of a set should be defined so that two sets are identical if they have the same elements, but not necessarily in the same order. The `hashCode` method should be defined such that two sets with the same elements yield the same hash code.

Why make a separate interface if the method signatures are the same? Conceptually, not all collections are sets. Making a `Set` interface enables programmers to write methods that accept only sets.

The `SortedSet` and `SortedMap` interfaces expose the comparator object used for sorting, and they define methods to obtain views of subsets of the collections. We discuss these views in the next section.

Finally, Java SE 6 introduced interfaces `NavigableSet` and `NavigableMap` that contain additional methods for searching and traversal in sorted sets and maps. (Ideally, these methods should have simply been included in the `SortedSet` and `SortedMap` interface.) The `TreeSet` and `TreeMap` classes implement these interfaces.

Now, let us turn from the interfaces to the classes that implement them. We already discussed that the collection interfaces have quite a few methods that can be trivially implemented from more fundamental methods. Abstract classes supply many of these routine implementations:

AbstractCollection  
AbstractList  
AbstractSequentialList  
AbstractSet  
AbstractQueue  
AbstractMap

If you implement your own collection class, then you probably want to extend one of these classes so that you can pick up the implementations of the routine operations.

The Java library supplies concrete classes:

LinkedList  
ArrayList  
ArrayDeque  
HashSet  
TreeSet  
PriorityQueue  
HashMap  
TreeMap

Figure 13–11 shows the relationships between these classes.

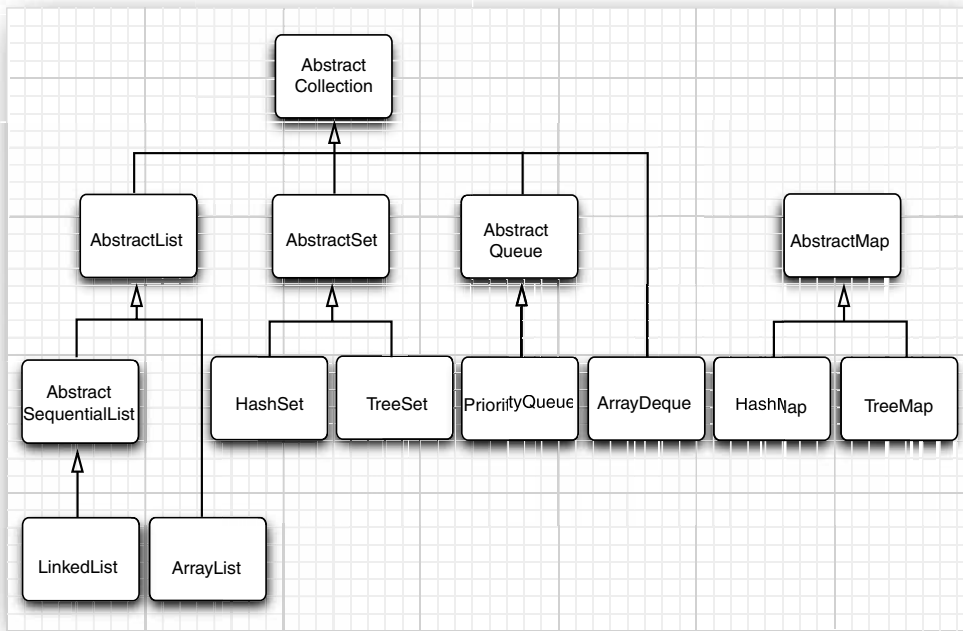
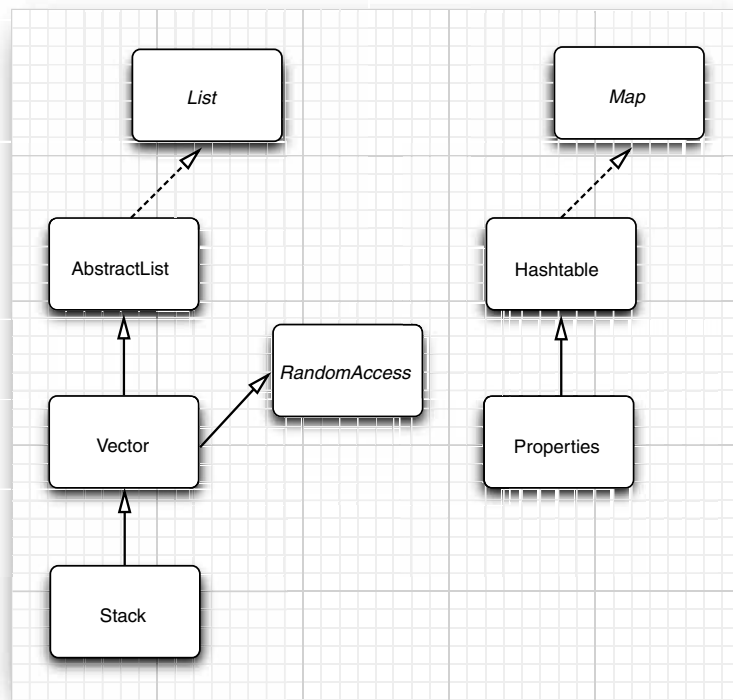


Figure 13–11 Classes in the collections framework

Finally, a number of “legacy” container classes have been present since the first release of Java, before there was a collections framework:

- Vector
- Stack
- Hashtable
- Properties

They have been integrated into the collections framework—see Figure 13–12. We discuss these classes later in this chapter.



**Figure 13–12** Legacy classes in the collections framework

### **Views and Wrappers**

If you look at Figure 13–10 and Figure 13–11, you might think it is overkill to have lots of interfaces and abstract classes to implement a modest number of concrete collection classes. However, these figures don’t tell the whole story. By using *views*, you can obtain other objects that implement the `Collection` or `Map` interfaces. You saw one example of this with the `keySet` method of the map classes. At first glance, it appears as if the method creates a new set, fills it with all keys of the map, and returns it. However, that is not the

case. Instead, the `keySet` method returns an object of a class that implements the `Set` interface and whose methods manipulate the original map. Such a collection is called a *view*.

The technique of views has a number of useful applications in the collections framework. We discuss these applications in the following sections.

### Lightweight Collection Wrappers

The static `asList` method of the `Arrays` class returns a `List` wrapper around a plain Java array. This method lets you pass the array to a method that expects a list or collection argument. For example:

```
Card[] cardDeck = new Card[52];
. . .
List<Card> cardList = Arrays.asList(cardDeck);
```

The returned object is *not* an `ArrayList`. It is a view object with `get` and `set` methods that access the underlying array. All methods that would change the size of the array (such as `add` and the `remove` method of the associated iterator) throw an `UnsupportedOperationException`.

As of Java SE 5.0, the `asList` method is declared to have a variable number of arguments. Instead of passing an array, you can also pass individual elements. For example:

```
List<String> names = Arrays.asList("Amy", "Bob", "Carl");
```

The method call

```
Collections.nCopies(n, anObject)
```

returns an immutable object that implements the `List` interface and gives the illusion of having `n` elements, each of which appears as `anObject`.

For example, the following call creates a `List` containing 100 strings, all set to "DEFAULT":

```
List<String> settings = Collections.nCopies(100, "DEFAULT");
```

There is very little storage cost—the object is stored only once. This is a cute application of the view technique.



**NOTE:** The `Collections` class contains a number of utility methods with parameters or return values that are collections. Do not confuse it with the `Collection` interface.

The method call

```
Collections.singleton(anObject)
```

returns a view object that implements the `Set` interface (unlike `nCopies`, which produces a `List`). The returned object implements an immutable single-element set without the overhead of data structure. The methods `singletonList` and `singletonMap` behave similarly.

### Subranges

You can form subrange views for a number of collections. For example, suppose you have a list `staff` and want to extract elements 10 to 19. You use the `subList` method to obtain a view into the subrange of the list.

```
List group2 = staff.subList(10, 20);
```

The first index is inclusive, the second exclusive—just like the parameters for the `substring` operation of the `String` class.

You can apply any operations to the subrange, and they automatically reflect the entire list. For example, you can erase the entire subrange:

```
group2.clear(); // staff reduction
```

The elements are now automatically cleared from the staff list, and `group2` is empty.

For sorted sets and maps, you use the sort order, not the element position, to form subranges. The `SortedSet` interface declares three methods:

```
SortedSet<E> subSet(E from, E to)
SortedSet<E> headSet(E to)
SortedSet<E> tailSet(E from)
```

These return the subsets of all elements that are larger than or equal to `from` and strictly smaller than `to`. For sorted maps, the similar methods

```
SortedMap<K, V> subMap(K from, K to)
SortedMap<K, V> headMap(K to)
SortedMap<K, V> tailMap(K from)
```

return views into the maps consisting of all entries in which the *keys* fall into the specified ranges.

The `NavigableSet` interface that was introduced in Java SE 6 gives more control over these subrange operations. You can specify whether the bounds are included:

```
NavigableSet<E> subSet(E from, boolean fromInclusive, E to, boolean toInclusive)
NavigableSet<E> headSet(E to, boolean toInclusive)
NavigableSet<E> tailSet(E from, boolean fromInclusive)
```

### Unmodifiable Views

The `Collections` class has methods that produce *unmodifiable views* of collections. These views add a runtime check to an existing collection. If an attempt to modify the collection is detected, then an exception is thrown and the collection remains untouched.

You obtain unmodifiable views by six methods:

```
Collections.unmodifiableCollection
Collections.unmodifiableList
Collections.unmodifiableSet
Collections.unmodifiableSortedSet
Collections.unmodifiableMap
Collections.unmodifiableSortedMap
```

Each method is defined to work on an interface. For example, `Collections.unmodifiableList` works with an `ArrayList`, a `LinkedList`, or any other class that implements the `List` interface.

For example, suppose you want to let some part of your code look at, but not touch, the contents of a collection. Here is what you could do:

```
List<String> staff = new LinkedList<String>();
. . .
lookAt(new Collections.unmodifiableList(staff));
```

The `Collections.unmodifiableList` method returns an object of a class implementing the `List` interface. Its accessor methods retrieve values from the `staff` collection. Of course, the `lookAt` method can call all methods of the `List` interface, not just the accessors. But all mutator methods (such as `add`) have been redefined to throw an `UnsupportedOperationException` instead of forwarding the call to the underlying collection.

The unmodifiable view does not make the collection itself immutable. You can still modify the collection through its original reference (staff, in our case). And you can still call mutator methods on the elements of the collection.

Because the views wrap the *interface* and not the actual collection object, you only have access to those methods that are defined in the interface. For example, the `LinkedList` class has convenience methods, `addFirst` and `addLast`, that are not part of the `List` interface. These methods are not accessible through the unmodifiable view.



**CAUTION:** The `unmodifiableCollection` method (as well as the `synchronizedCollection` and `checkedCollection` methods discussed later in this section) returns a collection whose `equals` method does *not* invoke the `equals` method of the underlying collection. Instead, it inherits the `equals` method of the `Object` class, which just tests whether the objects are identical. If you turn a set or list into just a collection, you can no longer test for equal contents. The view acts in this way because equality testing is not well defined at this level of the hierarchy. The views treat the `hashCode` method in the same way.

However, the `unmodifiableSet` and `unmodifiableList` class use the `equals` and `hashCode` methods of the underlying collections.

### Synchronized Views

If you access a collection from multiple threads, you need to ensure that the collection is not accidentally damaged. For example, it would be disastrous if one thread tried to add to a hash table while another thread was rehashing the elements.

Instead of implementing thread-safe collection classes, the library designers used the view mechanism to make regular collections thread safe. For example, the `staticSynchronizedMap` method in the `Collections` class can turn any map into a `Map` with synchronized access methods:

```
Map<String, Employee> map = Collections.synchronizedMap(new HashMap<String, Employee>());
```

You can now access the `map` object from multiple threads. The methods such as `get` and `put` are serialized—each method call must be finished completely before another thread can call another method. We discuss the issue of synchronized access to data structures in greater detail in Chapter 14.

### Checked Views

Java SE 5.0 added a set of “checked” views that are intended as debugging support for a problem that can occur with generic types. As explained in Chapter 12, it is actually possible to smuggle elements of the wrong type into a generic collection. For example:

```
ArrayList<String> strings = new ArrayList<String>();
ArrayList rawList = strings; // get warning only, not an error, for compatibility with legacy code
rawList.add(new Date()); // now strings contains a Date object!
```

The erroneous `add` command is not detected at runtime. Instead, a class cast exception will happen later when another part of the code calls `get` and casts the result to a `String`.

A checked view can detect this problem. Define a safe list as follows:

```
List<String> safeStrings = Collections.checkedList(strings, String.class);
```



The view's `add` method checks that the inserted object belongs to the given class and immediately throws a `ClassCastException` if it does not. The advantage is that the error is reported at the correct location:

```
ArrayList rawList = safeStrings;
rawList.add(new Date()); // Checked list throws a ClassCastException
```



**CAUTION:** The checked views are limited by the runtime checks that the virtual machine can carry out. For example, if you have an `ArrayList<Pair<String>>`, you cannot protect it from inserting a `Pair<Date>` since the virtual machine has a single “raw” `Pair` class.

### A Note on Optional Operations

A view usually has some restriction—it may be read-only, it may not be able to change the size, or it may support removal, but not insertion, as is the case for the key view of a map. A restricted view throws an `UnsupportedOperationException` if you attempt an inappropriate operation.

In the API documentation for the collection and iterator interfaces, many methods are described as “optional operations.” This seems to be in conflict with the notion of an interface. After all, isn't the purpose of an interface to lay out the methods that a class *must* implement? Indeed, this arrangement is unsatisfactory from a theoretical perspective. A better solution might have been to design separate interfaces for read-only views and views that can't change the size of a collection. However, that would have tripled the number of interfaces, which the designers of the library found unacceptable.

Should you extend the technique of “optional” methods to your own designs? We think not. Even though collections are used frequently, the coding style for implementing them is not typical for other problem domains. The designers of a collection class library have to resolve a particularly brutal set of conflicting requirements. Users want the library to be easy to learn, convenient to use, completely generic, idiot proof, and at the same time as efficient as hand-coded algorithms. It is plainly impossible to achieve all these goals simultaneously, or even to come close. But in your own programming problems, you will rarely encounter such an extreme set of constraints. You should be able to find solutions that do not rely on the extreme measure of “optional” interface operations.

### API java.util.Collections 1.2

- `static <E> Collection unmodifiableCollection(Collection<E> c)`
  - `static <E> List unmodifiableList(List<E> c)`
  - `static <E> Set unmodifiableSet(Set<E> c)`
  - `static <E> SortedSet unmodifiableSortedSet(SortedSet<E> c)`
  - `static <K, V> Map unmodifiableMap(Map<K, V> c)`
  - `static <K, V> SortedMap unmodifiableSortedMap(SortedMap<K, V> c)`
- constructs views of the collection whose mutator methods throw an `UnsupportedOperationException`.

- static `<E> Collection<E> synchronizedCollection(Collection<E> c)`
- static `<E> List synchronizedList(List<E> c)`
- static `<E> Set synchronizedSet(Set<E> c)`
- static `<E> SortedSet synchronizedSortedSet(SortedSet<E> c)`
- static `<K, V> Map<K, V> synchronizedMap(Map<K, V> c)`
- static `<K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> c)`  
constructs views of the collection whose methods are synchronized.
- static `<E> Collection checkedCollection(Collection<E> c, Class<E> elementType)`
- static `<E> List checkedList(List<E> c, Class<E> elementType)`
- static `<E> Set checkedSet(Set<E> c, Class<E> elementType)`
- static `<E> SortedSet checkedSortedSet(SortedSet<E> c, Class<E> elementType)`
- static `<K, V> Map checkedMap(Map<K, V> c, Class<K> keyType, Class<V> valueType)`
- static `<K, V> SortedMap checkedSortedMap(SortedMap<K, V> c, Class<K> keyType, Class<V> valueType)`  
constructs views of the collection whose methods throw a `ClassCastException` if an element of the wrong type is inserted.
- static `<E> List<E> nCopies(int n, E value)`
- static `<E> Set<E> singleton(E value)`  
constructs a view of the object as either an unmodifiable list with `n` identical elements, or a set with a single element.

**API** `java.util.Arrays` 1.2

- static `<E> List<E> asList(E... array)`  
returns a list view of the elements in an array that is modifiable but not resizable.

**API** `java.util.List<E>` 1.2

- `List<E> subList(int firstIncluded, int firstExcluded)`  
returns a list view of the elements within a range of positions.

**API** `java.util.SortedSet<E>` 1.2

- `SortedSet<E> subSet(E firstIncluded, E firstExcluded)`
- `SortedSet<E> headSet(E firstExcluded)`
- `SortedSet<E> tailSet(E firstIncluded)`  
returns a view of the elements within a range.

**API** `java.util.NavigableSet<E>` 6

- `NavigableSet<E> subSet(E from, boolean fromIncluded, E to, boolean toIncluded)`
- `NavigableSet<E> headSet(E to, boolean toIncluded)`
- `NavigableSet<E> tailSet(E from, boolean fromIncluded)`  
returns a view of the elements within a range. The `boolean` flags determine whether the bound is included in the view.

**API** `java.util.SortedMap<K, V>` 1.2

- `SortedMap<K, V> subMap(K firstIncluded, K firstExcluded)`
  - `SortedMap<K, V> headMap(K firstExcluded)`
  - `SortedMap<K, V> tailMap(K firstIncluded)`
- returns a map view of the entries whose keys are within a range.

**API** `java.util.NavigableMap<K, V>` 6

- `NavigableMap<K, V> subMap(K from, boolean fromIncluded, K to, boolean toIncluded)`
  - `NavigableMap<K, V> headMap(K from, boolean fromIncluded)`
  - `NavigableMap<K, V> tailMap(K to, boolean toIncluded)`
- returns a map view of the entries whose keys are within a range. The boolean flags determine whether the bound is included in the view

**Bulk Operations**

So far, most of our examples used an iterator to traverse a collection, one element at a time. However, you can often avoid iteration by using one of the *bulk operations* in the library.

Suppose you want to find the *intersection* of two sets, the elements that two sets have in common. First, make a new set to hold the result.

```
Set<String> result = new HashSet<String>(a);
```

Here, you use the fact that every collection has a constructor whose parameter is another collection that holds the initialization values.

Now, use the `retainAll` method:

```
result.retainAll(b);
```

It retains all elements that also happen to be in `b`. You have formed the intersection without programming a loop.

You can carry this idea further and apply a bulk operation to a *view*. For example, suppose you have a map that maps employee IDs to employee objects and you have a set of the IDs of all employees that are to be terminated.

```
Map<String, Employee> staffMap = . . . ;
Set<String> terminatedIDs = . . . ;
```

Simply form the key set and remove all IDs of terminated employees.

```
staffMap.keySet().removeAll(terminatedIDs);
```

Because the key set is a view into the map, the keys and associated employee names are automatically removed from the map.

By using a subrange view, you can restrict bulk operations to sublists and subsets. For example, suppose you want to add the first 10 elements of a list to another container.

Form a sublist to pick out the first 10:

```
relocated.addAll(staff.subList(0, 10));
```

The subrange can also be a target of a mutating operation.

```
staff.subList(0, 10).clear();
```

### Converting between Collections and Arrays

Because large portions of the Java platform API were designed before the collections framework was created, you occasionally need to translate between traditional arrays and the more modern collections.

If you have an array, you need to turn it into a collection. The `Arrays.asList` wrapper serves this purpose. For example:

```
String[] values = . . . ;
HashSet<String> staff = new HashSet<String>(Arrays.asList(values));
```

Obtaining an array from a collection is a bit trickier. Of course, you can use the `toArray` method:

```
Object[] values = staff.toArray();
```

But the result is an array of *objects*. Even if you know that your collection contained objects of a specific type, you cannot use a cast:

```
String[] values = (String[]) staff.toArray(); // Error!
```

The array returned by the `toArray` method was created as an `Object[]` array, and you cannot change its type. Instead, you use a variant of the `toArray` method. Give it an array of length 0 of the type that you'd like. The returned array is then created *as the same array type*:

```
String[] values = staff.toArray(new String[0]);
```

If you like, you can construct the array to have the correct size:

```
staff.toArray(new String[staff.size()]);
```

In this case, no new array is created.



**NOTE:** You may wonder why you don't simply pass a `Class` object (such as `String.class`) to the `toArray` method. However, this method does "double duty," both to fill an existing array (provided it is long enough) and to create a new array.

### Algorithms

Generic collection interfaces have a great advantage—you only need to implement your algorithms once. For example, consider a simple algorithm to compute the maximum element in a collection. Traditionally, programmers would implement such an algorithm as a loop. Here is how you find the largest element of an array.

```
if (a.length == 0) throw new NoSuchElementException();
T largest = a[0];
for (int i = 1; i < a.length; i++)
    if (largest.compareTo(a[i]) < 0)
        largest = a[i];
```

Of course, to find the maximum of an array list, you would write the code slightly differently.

```
if (v.size() == 0) throw new NoSuchElementException();
T largest = v.get(0);
for (int i = 1; i < v.size(); i++)
    if (largest.compareTo(v.get(i)) < 0)
        largest = v.get(i);
```

What about a linked list? You don't have efficient random access in a linked list, but you can use an iterator.

```
if (l.isEmpty()) throw new NoSuchElementException();
Iterator<T> iter = l.iterator();
T largest = iter.next();
while (iter.hasNext())
{
    T next = iter.next();
    if (largest.compareTo(next) < 0)
        largest = next;
}
```

These loops are tedious to write, and they are just a bit error prone. Is there an off-by-one error? Do the loops work correctly for empty containers? For containers with only one element? You don't want to test and debug this code every time, but you also don't want to implement a whole slew of methods, such as these:

```
static <T extends Comparable> T max(T[] a)
static <T extends Comparable> T max(ArrayList<T> v)
static <T extends Comparable> T max(LinkedList<T> l)
```

That's where the collection interfaces come in. Think of the *minimal* collection interface that you need to efficiently carry out the algorithm. Random access with `get` and `set` comes higher in the food chain than simple iteration. As you have seen in the computation of the maximum element in a linked list, random access is not required for this task. Computing the maximum can be done simply by iteration through the elements. Therefore, you can implement the `max` method to take *any* object that implements the `Collection` interface.

```
public static <T extends Comparable> T max(Collection<T> c)
{
    if (c.isEmpty()) throw new NoSuchElementException();
    Iterator<T> iter = c.iterator();
    T largest = iter.next();
    while (iter.hasNext())
    {
        T next = iter.next();
        if (largest.compareTo(next) < 0)
            largest = next;
    }
    return largest;
}
```

Now you can compute the maximum of a linked list, an array list, or an array, with a single method.

That's a powerful concept. In fact, the standard C++ library has dozens of useful algorithms, each of which operates on a generic collection. The Java library is not quite so rich, but it does contain the basics: sorting, binary search, and some utility algorithms.

### **Sorting and Shuffling**

Computer old-timers will sometimes reminisce about how they had to use punched cards and how they actually had to program by hand algorithms for sorting. Nowadays,

of course, sorting algorithms are part of the standard library for most programming languages, and the Java programming language is no exception.

The `sort` method in the `Collections` class sorts a collection that implements the `List` interface.

```
List<String> staff = new LinkedList<String>();
// fill collection . . .;
Collections.sort(staff);
```

This method assumes that the list elements implement the `Comparable` interface. If you want to sort the list in some other way, you can pass a `Comparator` object as a second parameter. (We discussed comparators in the section “Object Comparison” on page 673.) Here is how you can sort a list of items:

```
Comparator<Item> itemComparator = new
    Comparator<Item>()
    {
        public int compare(Item a, Item b)
        {
            return a.partNumber - b.partNumber;
        }
    };
Collections.sort(items, itemComparator);
```

If you want to sort a list in *descending* order, then use the static convenience method `Collections.reverseOrder()`. It returns a comparator that returns `b.compareTo(a)`. For example,

```
Collections.sort(staff, Collections.reverseOrder())
```

sorts the elements in the list `staff` in reverse order, according to the ordering given by the `compareTo` method of the element type. Similarly,

```
Collections.sort(items, Collections.reverseOrder(itemComparator))
```

reverses the ordering of the `itemComparator`.

You may wonder how the `sort` method sorts a list. Typically, when you look at a sorting algorithm in a book on algorithms, it is presented for arrays and uses random element access. However, random access in a list can be inefficient. You can actually sort lists efficiently by using a form of merge sort (see, for example, *Algorithms in C++* by Robert Sedgwick [Addison-Wesley, 1998, pp. 366–369]). However, the implementation in the Java programming language does not do that. It simply dumps all elements into an array, sorts the array by using a different variant of merge sort, and then copies the sorted sequence back into the list.

The merge sort algorithm used in the collections library is a bit slower than *quick sort*, the traditional choice for a general-purpose sorting algorithm. However, it has one major advantage: It is *stable*, that is, it doesn’t switch equal elements. Why do you care about the order of equal elements? Here is a common scenario. Suppose you have an employee list that you already sorted by name. Now you sort by salary. What happens to employees with equal salary? With a stable sort, the ordering by name is preserved. In other words, the outcome is a list that is sorted first by salary, then by name.

Because collections need not implement all of their “optional” methods, all methods that receive collection parameters must describe when it is safe to pass a collection to an algorithm. For example, you clearly cannot pass an `unmodifiableList` list to the `sort`

algorithm. What kind of list *can* you pass? According to the documentation, the list must be modifiable but need not be resizable.

The terms are defined as follows:

- A list is *modifiable* if it supports the `set` method.
- A list is *resizable* if it supports the `add` and `remove` operations.

The `Collections` class has an algorithm `shuffle` that does the opposite of sorting—it randomly permutes the order of the elements in a list. For example:

```
ArrayList<Card> cards = . . . ;
Collections.shuffle(cards);
```

If you supply a list that does not implement the `RandomAccess` interface, then the `shuffle` method copies the elements into an array, shuffles the array, and copies the shuffled elements back into the list.

The program in Listing 13–6 fills an array list with 49 `Integer` objects containing the numbers 1 through 49. It then randomly shuffles the list and selects the first 6 values from the shuffled list. Finally, it sorts the selected values and prints them.

#### Listing 13–6 ShuffleTest.java

```
1. import java.util.*;
2.
3. /**
4.  * This program demonstrates the random shuffle and sort algorithms.
5.  * @version 1.10 2004-08-02
6.  * @author Cay Horstmann
7.  */
8. public class ShuffleTest
9. {
10.     public static void main(String[] args)
11.     {
12.         List<Integer> numbers = new ArrayList<Integer>();
13.         for (int i = 1; i <= 49; i++)
14.             numbers.add(i);
15.         Collections.shuffle(numbers);
16.         List<Integer> winningCombination = numbers.subList(0, 6);
17.         Collections.sort(winningCombination);
18.         System.out.println(winningCombination);
19.     }
20. }
```

#### API java.util.Collections 1.2

- `static <T extends Comparable<? super T>> void sort(List<T> elements)`
- `static <T> void sort(List<T> elements, Comparator<? super T> c)`  
sorts the elements in the list, using a stable sort algorithm. The algorithm is guaranteed to run in  $O(n \log n)$  time, where  $n$  is the length of the list.

- `static void shuffle(List<?> elements)`
- `static void shuffle(List<?> elements, Random r)`  
randomly shuffles the elements in the list. This algorithm runs in  $O(n a(n))$  time, where  $n$  is the length of the list and  $a(n)$  is the average time to access an element.
- `static <T> Comparator<T> reverseOrder()`  
returns a comparator that sorts elements in the reverse order of the one given by the `compareTo` method of the `Comparable` interface.
- `static <T> Comparator<T> reverseOrder(Comparator<T> comp)`  
returns a comparator that sorts elements in the reverse order of the one given by `comp`.

### Binary Search

To find an object in an array, you normally visit all elements until you find a match. However, if the array is sorted, then you can look at the middle element and check whether it is larger than the element that you are trying to find. If so, you keep looking in the first half of the array; otherwise, you look in the second half. That cuts the problem in half. You keep going in the same way. For example, if the array has 1024 elements, you will locate the match (or confirm that there is none) after 10 steps, whereas a linear search would have taken you an average of 512 steps if the element is present, and 1024 steps to confirm that it is not.

The `binarySearch` of the `Collections` class implements this algorithm. Note that the collection must already be sorted or the algorithm will return the wrong answer. To find an element, supply the collection (which must implement the `List` interface—more on that in the note below) and the element to be located. If the collection is not sorted by the `compareTo` element of the `Comparable` interface, then you must supply a comparator object as well.

```
i = Collections.binarySearch(c, element);
i = Collections.binarySearch(c, element, comparator);
```

A return value of  $\geq 0$  from the `binarySearch` method denotes the index of the matching object. That is, `c.get(i)` is equal to `element` under the comparison order. If the value is negative, then there is no matching element. However, you can use the return value to compute the location where you *should* insert `element` into the collection to keep it sorted. The insertion location is

```
insertionPoint = -i - 1;
```

It isn't simply `-i` because then the value of 0 would be ambiguous. In other words, the operation

```
if (i < 0)
    c.add(-i - 1, element);
```

adds the element in the correct place.

To be worthwhile, binary search requires random access. If you have to iterate one by one through half of a linked list to find the middle element, you have lost all advantage of the binary search. Therefore, the `binarySearch` algorithm reverts to a linear search if you give it a linked list.





NOTE: Java SE 1.3 had no separate interface for an ordered collection with efficient random access, and the `binarySearch` method employed a very crude device, checking whether the list parameter extended the `AbstractSequentialList` class. This was fixed in Java SE 1.4. Now the `binarySearch` method checks whether the list parameter implements the `RandomAccess` interface. If it does, then the method carries out a binary search. Otherwise, it uses a linear search.

#### API `java.util.Collections` 1.2

- `static <T extends Comparable<? super T>> int binarySearch(List<T> elements, T key)`
- `static <T> int binarySearch(List<T> elements, T key, Comparator<? super T> c)`  
searches for a key in a sorted list, using a linear search if `elements` extends the `AbstractSequentialList` class, and a binary search in all other cases. The methods are guaranteed to run in  $O(a(n) \log n)$  time, where  $n$  is the length of the list and  $a(n)$  is the average time to access an element. The methods return either the index of the key in the list, or a negative value  $i$  if the key is not present in the list. In that case, the key should be inserted at index  $-i - 1$  for the list to stay sorted.

#### Simple Algorithms

The `Collections` class contains several simple but useful algorithms. Among them is the example from the beginning of this section, finding the maximum value of a collection. Others include copying elements from one list to another, filling a container with a constant value, and reversing a list. Why supply such simple algorithms in the standard library? Surely most programmers could easily implement them with simple loops. We like the algorithms because they make life easier for the programmer *reading* the code. When you read a loop that was implemented by someone else, you have to decipher the original programmer's intentions. When you see a call to a method such as `Collections.max`, you know right away what the code does.

The following API notes describe the simple algorithms in the `Collections` class.

#### API `java.util.Collections` 1.2

- `static <T extends Comparable<? super T>> T min(Collection<T> elements)`
- `static <T extends Comparable<? super T>> T max(Collection<T> elements)`
- `static <T> min(Collection<T> elements, Comparator<? super T> c)`
- `static <T> max(Collection<T> elements, Comparator<? super T> c)`  
returns the smallest or largest element in the collection. (The parameter bounds are simplified for clarity.)
- `static <T> void copy(List<? super T> to, List<T> from)`  
copies all elements from a source list to the same positions in the target list. The target list must be at least as long as the source list.
- `static <T> void fill(List<? super T> l, T value)`  
sets all positions of a list to the same value.

- static `<T> boolean addAll(Collection<? super T> c, T... values)` **5.0**  
adds all values to the given collection and returns true if the collection changed as a result.
- static `<T> boolean replaceAll(List<T> l, T oldValue, T newValue)` **1.4**  
replaces all elements equal to `oldValue` with `newValue`.
- static `int indexOfSubList(List<?> l, List<?> s)` **1.4**
- static `int lastIndexOfSubList(List<?> l, List<?> s)` **1.4**  
returns the index of the first or last sublist of `l` equalling `s`, or `-1` if no sublist of `l` equals `s`. For example, if `l` is `[s, t, a, r]` and `s` is `[t, a, r]`, then both methods return the index `1`.
- static `void swap(List<?> l, int i, int j)` **1.4**  
swaps the elements at the given offsets.
- static `void reverse(List<?> l)`  
reverses the order of the elements in a list. For example, reversing the list `[t, a, r]` yields the list `[r, a, t]`. This method runs in  $O(n)$  time, where  $n$  is the length of the list.
- static `void rotate(List<?> l, int d)` **1.4**  
rotates the elements in the list, moving the entry with index `i` to position `(i + d) % l.size()`. For example, rotating the list `[t, a, r]` by `2` yields the list `[a, r, t]`. This method runs in  $O(n)$  time, where  $n$  is the length of the list.
- static `int frequency(Collection<?> c, Object o)` **5.0**  
returns the count of elements in `c` that equal the object `o`.
- `boolean disjoint(Collection<?> c1, Collection<?> c2)` **5.0**  
returns true if the collections have no elements in common.

### Writing Your Own Algorithms

If you write your own algorithm (or in fact, any method that has a collection as a parameter), you should work with *interfaces*, not concrete implementations, whenever possible. For example, suppose you want to fill a `JMenu` with a set of menu items. Traditionally, such a method might have been implemented like this:

```
void fillMenu(JMenu menu, ArrayList<JMenuItem> items)
{
    for (JMenuItem item : items)
        menu.addItem(item);
}
```

However, you now constrained the caller of your method—the caller must supply the choices in an `ArrayList`. If the choices happen to be in another container, they first need to be repackaged. It is much better to accept a more general collection.

You should ask yourself this: What is the most general collection interface that can do the job? In this case, you just need to visit all elements, a capability of the `basicCollection` interface. Here is how you can rewrite the `fillMenu` method to accept collections of any kind.

```
void fillMenu(JMenu menu, Collection<JMenuItem> items)
{
    for (JMenuItem item : items)
        menu.addItem(item);
}
```

Now, anyone can call this method, with an `ArrayList` or a `LinkedList`, or even with an array, wrapped with the `Arrays.asList` wrapper.



**NOTE:** If it is such a good idea to use collection interfaces as method parameters, why doesn't the Java library follow this rule more often? For example, the `JComboBox` class has two constructors:

```
JComboBox(Object[] items)
JComboBox(Vector<?> items)
```

The reason is simply timing. The Swing library was created before the collections library.

If you write a method that *returns* a collection, you may also want to return an interface instead of a class because you can then change your mind and reimplement the method later with a different collection.

For example, let's write a method `getAllItems` that returns all items of a menu.

```
List<MenuItem> getAllItems(JMenu menu)
{
    ArrayList<MenuItem> items = new ArrayList<MenuItem>()
    for (int i = 0; i < menu.getItemCount(); i++)
        items.add(menu.getItem(i));
    return items;
}
```

Later, you can decide that you don't want to *copy* the items but simply provide a view into them. You achieve this by returning an anonymous subclass of `AbstractList`.

```
List<MenuItem> getAllItems(final JMenu menu)
{
    return new
        AbstractList<MenuItem>()
        {
            public MenuItem get(int i)
            {
                return item.getItem(i);
            }
            public int size()
            {
                return item.getItemCount();
            }
        };
}
```


Of course, this is an advanced technique. If you employ it, be careful to document exactly which "optional" operations are supported. In this case, you must advise the caller that the returned object is an unmodifiable list.

## Legacy Collections

In this section, we discuss the collection classes that existed in the Java programming language since the beginning: the `Hashtable` class and its useful `Properties` subclass, the `Stack` subclass of `Vector`, and the `BitSet` class.

**The Hashtable Class**

The classic `Hashtable` class serves the same purpose as the `HashMap` and has essentially the same interface. Just like methods of the `Vector` class, the `Hashtable` methods are synchronized. If you do not require synchronization or compatibility with legacy code, you should use the `HashMap` instead.

 **NOTE:** The name of the class is `Hashtable`, with a lowercase `t`. Under Windows, you'll get strange error messages if you use `HashTable`, because the Windows file system is not case-sensitive but the Java compiler is.

**Enumerations**


The legacy collections use the `Enumeration` interface for traversing sequences of elements. The `Enumeration` interface has two methods, `hasMoreElements` and `nextElement`. These are entirely analogous to the `hasNext` and `next` methods of the `Iterator` interface.

For example, the `elements` method of the `Hashtable` class yields an object for enumerating the values in the table:

```
Enumeration<Employee> e = staff.elements();
while (e.hasMoreElements())
{
    Employee e = e.nextElement();
    . . .
}
```

You will occasionally encounter a legacy method that expects an enumeration parameter. The static method `Collections.enumeration` yields an enumeration object that enumerates the elements in the collection. For example:

```
List<InputStream> streams = . . . ;
SequenceInputStream in = new SequenceInputStream(Collections.enumeration(streams));
// the SequenceInputStream constructor expects an enumeration
```

 **NOTE:** In C++, it is quite common to use iterators as parameters. Fortunately, in programming for the Java platform, very few programmers use this idiom. It is much smarter to pass around the collection than to pass an iterator. The collection object is more useful. The recipients can always obtain the iterator from the collection when they need to do so, plus they have all the collection methods at their disposal. However, you will find enumerations in some legacy code because they were the only available mechanism for generic collections until the collections framework appeared in Java SE 1.2.

**API** `java.util.Enumeration<E>` 1.0

- `boolean hasMoreElements()`  
returns `true` if there are more elements yet to be inspected.
- `E nextElement()`  
returns the next element to be inspected. Do not call this method if `hasMoreElements()` returned `false`.

**API** `java.util.Hashtable<K, V>` 1.0

- `Enumeration<K> keys()`  
returns an enumeration object that traverses the keys of the hash table.
- `Enumeration<V> elements()`  
returns an enumeration object that traverses the elements of the hash table.

**API** `java.util.Vector<E>` 1.0

- `Enumeration<E> elements()`  
returns an enumeration object that traverses the elements of the vector.

**Property Maps**

A *property map* is a map structure of a very special type. It has three particular characteristics:

- The keys and values are strings.
- The table can be saved to a file and loaded from a file.
- A secondary table for defaults is used.

The Java platform class that implements a property map is called `Properties`.

Property maps are commonly used in specifying configuration options for programs—see Chapter 10.

**API** `java.util.Properties` 1.0

- `Properties()`  
creates an empty property map.
- `Properties(Properties defaults)`  
creates an empty property map with a set of defaults.
- `String getProperty(String key)`  
gets a property association; returns the string associated with the key, or the string associated with the key in the default table if it wasn't present in the map.
- `String getProperty(String key, String defaultValue)`  
gets a property with a default value if the key is not found; returns the string associated with the key, or the default string if it wasn't present in the map.
- `void load(InputStream in)`  
loads a property map from an `InputStream`.
- `void store(OutputStream out, String commentString)`  
stores a property map to an `OutputStream`.

**Stacks**

Since version 1.0, the standard library had a `Stack` class with the familiar `push` and `pop` methods. However, the `Stack` class extends the `Vector` class, which is not satisfactory from a theoretical perspective—you can apply such un-stack-like operations as `insert` and `remove` to insert and remove values anywhere, not just at the top of the stack.

**API** `java.util.Stack<E>` 1.0

- `E push(E item)`  
pushes `item` onto the stack and returns `item`.
- `E pop()`  
pops and returns the top item of the stack. Don't call this method if the stack is empty.
- `E peek()`  
returns the top of the stack without popping it. Don't call this method if the stack is empty.

**Bit Sets**

The Java platform `BitSet` class stores a sequence of bits. (It is not a *set* in the mathematical sense—bit *vector* or bit *array* would have been more appropriate terms.) Use a bit set if you need to store a sequence of bits (for example, flags) efficiently. Because a bit set packs the bits into bytes, it is far more efficient to use a bit set than to use an `ArrayList` of `Boolean` objects.

The `BitSet` class gives you a convenient interface for reading, setting, or resetting individual bits. Use of this interface avoids the masking and other bit-fiddling operations that would be necessary if you stored bits in `int` or `long` variables.

For example, for a `BitSet` named `bucketOfBits`,

```
bucketOfBits.get(i)
```

returns `true` if the *i*'th bit is on, and `false` otherwise. Similarly,

```
bucketOfBits.set(i)
```

turns the *i*'th bit on. Finally,

```
bucketOfBits.clear(i)
```

turns the *i*'th bit off.

---

**C++** C++ NOTE: The C++ `bitset` template has the same functionality as the Java platform `BitSet`.

---

**API** `java.util.BitSet` 1.0

- `BitSet(int initialCapacity)`  
constructs a bit set.
- `int length()`  
returns the “logical length” of the bit set: 1 plus the index of the highest set bit.
- `boolean get(int bit)`  
gets a bit.
- `void set(int bit)`  
sets a bit.
- `void clear(int bit)`  
clears a bit.

- `void and(BitSet set)`  
logically ANDs this bit set with another.
- `void or(BitSet set)`  
logically ORs this bit set with another.
- `void xor(BitSet set)`  
logically XORs this bit set with another.
- `void andNot(BitSet set)`  
clears all bits in this bit set that are set in the other bit set.

### The “Sieve of Eratosthenes” Benchmark

As an example of using bit sets, we want to show you an implementation of the “sieve of Eratosthenes” algorithm for finding prime numbers. (A prime number is a number like 2, 3, or 5 that is divisible only by itself and 1, and the sieve of Eratosthenes was one of the first methods discovered to enumerate these fundamental building blocks.) This isn’t a terribly good algorithm for finding the number of primes, but for some reason it has become a popular benchmark for compiler performance. (It isn’t a good benchmark either, because it mainly tests bit operations.)

Oh well, we bow to tradition and include an implementation. This program counts all prime numbers between 2 and 2,000,000. (There are 148,933 primes, so you probably don’t want to print them all out.)

Without going into too many details of this program, the key is to march through a bit set with 2 million bits. We first turn on all the bits. After that, we turn off the bits that are multiples of numbers known to be prime. The positions of the bits that remain after this process are themselves the prime numbers. Listing 13–7 illustrates this program in the Java programming language, and Listing 13–8 is the C++ code.



NOTE: Even though the sieve isn’t a good benchmark, we couldn’t resist timing the two implementations of the algorithm. Here are the timing results on a 1.66-GHz dual core ThinkPad with 2 GB of RAM, running Ubuntu 7.04.

- C++ (g++ 4.1.2): 360 milliseconds
- Java (Java SE 6): 105 milliseconds

We have run this test for seven editions of *Core Java*, and in the last three editions. Java easily beat C++. In all fairness, if one cranks up the optimization level in the C++ compiler, it beats Java with a time of 60 milliseconds. Java could only match that if the program ran long enough to trigger the Hotspot just-in-time compiler.

### Listing 13–7 Sieve.java

```

1. import java.util.*;
2.
3. /**
4.  * This program runs the Sieve of Erathostenes benchmark. It computes all primes up to 2,000,000.
5.  * @version 1.21 2004-08-03
6.  * @author Cay Horstmann
7.  */

```

**Listing 13-7** Sieve.java (continued)

```
8. public class Sieve
9. {
10.     public static void main(String[] s)
11.     {
12.         int n = 2000000;
13.         long start = System.currentTimeMillis();
14.         BitSet b = new BitSet(n + 1);
15.         int count = 0;
16.         int i;
17.         for (i = 2; i <= n; i++)
18.             b.set(i);
19.         i = 2;
20.         while (i * i <= n)
21.         {
22.             if (b.get(i))
23.             {
24.                 count++;
25.                 int k = 2 * i;
26.                 while (k <= n)
27.                 {
28.                     b.clear(k);
29.                     k += i;
30.                 }
31.             }
32.             i++;
33.         }
34.         while (i <= n)
35.         {
36.             if (b.get(i)) count++;
37.             i++;
38.         }
39.         long end = System.currentTimeMillis();
40.         System.out.println(count + " primes");
41.         System.out.println((end - start) + " milliseconds");
42.     }
43. }
```

**Listing 13-8** Sieve.cpp

```
1. /**
2.     @version 1.21 2004-08-03
3.     @author Cay Horstmann
4. */
5.
6. #include <bitset>
7. #include <iostream>
8. #include <ctime>
```



**Listing 13–8** Sieve.cpp (continued)

```
9.
10. using namespace std;
11.
12. int main()
13. {
14.     const int N = 2000000;
15.     clock_t cstart = clock();
16.
17.     bitset<N + 1> b;
18.     int count = 0;
19.     int i;
20.     for (i = 2; i <= N; i++)
21.         b.set(i);
22.     i = 2;
23.     while (i * i <= N)
24.     {
25.         if (b.test(i))
26.         {
27.             count++;
28.             int k = 2 * i;
29.             while (k <= N)
30.             {
31.                 b.reset(k);
32.                 k += i;
33.             }
34.         }
35.         i++;
36.     }
37.     while (i <= N)
38.     {
39.         if (b.test(i))
40.             count++;
41.         i++;
42.     }
43.
44.     clock_t cend = clock();
45.     double millis = 1000.0
46.         * (cend - cstart) / CLOCKS_PER_SEC;
47.
48.     cout << count << " primes\n"
49.         << millis << " milliseconds\n";
50.
51.     return 0;
52. }
```

This completes our tour through the Java collection framework. As you have seen, the Java library offers a wide variety of collection classes for your programming needs. In the final chapter of this book, we will cover the important topic of concurrent programming.



---

*Chapter*

14

# MULTITHREADING

- ▼ WHAT ARE THREADS?
- ▼ INTERRUPTING THREADS
- ▼ THREAD STATES
- ▼ THREAD PROPERTIES
- ▼ SYNCHRONIZATION
- ▼ BLOCKING QUEUES
- ▼ THREAD-SAFE COLLECTIONS
- ▼ CALLABLES AND FUTURES
- ▼ EXECUTORS
- ▼ SYNCHRONIZERS
- ▼ THREADS AND SWING

**Y**ou are probably familiar with *multitasking* in your operating system: the ability to have more than one program working at what seems like the same time. For example, you can print while editing or downloading your email. Nowadays, you are likely to have a computer with more than one CPU, but the number of concurrently executing processes is not limited by the number of CPUs. The operating system assigns CPU time slices to each process, giving the impression of parallel activity.

Multithreaded programs extend the idea of multitasking by taking it one level lower: individual programs will appear to do multiple tasks at the same time. Each task is usually called a *thread*—which is short for thread of control. Programs that can run more than one thread at once are said to be *multithreaded*.

So, what is the difference between multiple *processes* and multiple *threads*? The essential difference is that while each process has a complete set of its own variables, threads share the same data. This sounds somewhat risky, and indeed it can be, as you will see later in this chapter. However, shared variables make communication between threads more efficient and easier to program than interprocess communication. Moreover, on some operating systems, threads are more “lightweight” than processes—it takes less overhead to create and destroy individual threads than it does to launch new processes.

Multithreading is extremely useful in practice. For example, a browser should be able to simultaneously download multiple images. A web server needs to be able to serve concurrent requests. Graphical user interface (GUI) programs have a separate thread for gathering user interface events from the host operating environment. This chapter shows you how to add multithreading capability to your Java applications.

Multithreading changed dramatically in Java SE 5.0, with the addition of a large number of classes and interfaces that provide high-quality implementations of the mechanisms that most application programmers will need. In this chapter, we explain the features that were added to Java SE 5.0 as well as the classic synchronization mechanisms, and help you choose between them.

Fair warning: multithreading can get very complex. In this chapter, we cover all the tools that an application programmer is likely to need. However, for more intricate system-level programming, we suggest that you turn to a more advanced reference, such as *Java Concurrency in Practice* by Brian Goetz (Addison-Wesley Professional, 2006).

### What Are Threads?

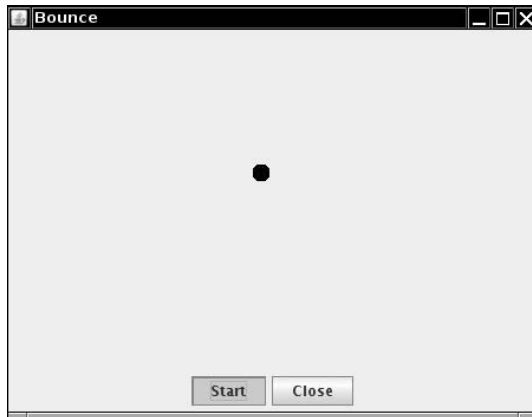
Let us start by looking at a program that does not use multiple threads and that, as a consequence, makes it difficult for the user to perform several tasks with that program. After we dissect it, we then show you how easy it is to have this program run separate threads. This program animates a bouncing ball by continually moving the ball, finding out if it bounces against a wall, and then redrawing it. (See Figure 14-1.)

As soon as you click the Start button, the program launches a ball from the upper-left corner of the screen and the ball begins bouncing. The handler of the Start button calls the `addBall` method. That method contains a loop running through 1,000 moves. Each call to `move` moves the ball by a small amount, adjusts the direction if it bounces against a wall, and then redraws the panel.

```
Ball ball = new Ball();
panel.add(ball);
```

```
for (int i = 1; i <= STEPS; i++)
{
    ball.move(panel.getBounds());
    panel.paint(panel.getGraphics());
    Thread.sleep(DELAY);
}
```

The static `sleep` method of the `Thread` class pauses for the given number of milliseconds.



**Figure 14–1** Using a thread to animate a bouncing ball

The call to `Thread.sleep` does not create a new thread—`sleep` is a static method of the `Thread` class that temporarily stops the activity of the current thread.

The `sleep` method can throw an `InterruptedException`. We discuss this exception and its proper handling later. For now, we simply terminate the bouncing if this exception occurs.

If you run the program, the ball bounces around nicely, but it completely takes over the application. If you become tired of the bouncing ball before it has finished its 1,000 moves and click the `Close` button, the ball continues bouncing anyway. You cannot interact with the program until the ball has finished bouncing.



**NOTE:** If you carefully look over the code at the end of this section, you will notice the call `comp.paint(comp.getGraphics())`

inside the `addBall` method of the `BounceFrame` class. That is pretty strange—normally, you'd call `repaint` and let the AWT worry about getting the graphics context and doing the painting. But if you try to call `comp.repaint()` in this program, you'll find that the panel is never repainted because the `addBall` method has completely taken over all processing. Also note that the ball component extends `JPanel`; this makes it easier to erase the background. In the next program, in which we use a separate thread to compute the ball position, we can go back to the familiar use of `repaint` and `JComponent`.

Obviously, the behavior of this program is rather poor. You would not want the programs that you use behaving in this way when you ask them to do a time-consuming job. After all, when you are reading data over a network connection, it is all too common to be stuck in a task that you would *really* like to interrupt. For example, suppose you download a large image and decide, after seeing a piece of it, that you do not need or want to see the rest; you certainly would like to be able to click a Stop or Back button to interrupt the loading process. In the next section, we show you how to keep the user in control by running crucial parts of the code in a separate *thread*.

Listings 14–1 through 14–3 show the code for the program.

**Listing 14–1** Bounce.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * Shows an animated bouncing ball.
7.  * @version 1.33 2007-05-17
8.  * @author Cay Horstmann
9.  */
10. public class Bounce
11. {
12.     public static void main(String[] args)
13.     {
14.         EventQueue.invokeLater(new Runnable()
15.         {
16.             public void run()
17.             {
18.                 JFrame frame = new BounceFrame();
19.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20.                 frame.setVisible(true);
21.             }
22.         });
23.     }
24. }
25.
26. /**
27.  * The frame with ball component and buttons.
28.  */
29. class BounceFrame extends JFrame
30. {
31.     /**
32.     * Constructs the frame with the component for showing the bouncing ball and Start and
33.     * Close buttons
34.     */
```

**Listing 14-1** Bounce.java (continued)

```
35. public BounceFrame()
36. {
37.     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
38.     setTitle("Bounce");
39.
40.     comp = new BallComponent();
41.     add(comp, BorderLayout.CENTER);
42.     JPanel buttonPanel = new JPanel();
43.     addButton(buttonPanel, "Start", new ActionListener()
44.     {
45.         public void actionPerformed(ActionEvent event)
46.         {
47.             addBall();
48.         }
49.     });
50.
51.     addButton(buttonPanel, "Close", new ActionListener()
52.     {
53.         public void actionPerformed(ActionEvent event)
54.         {
55.             System.exit(0);
56.         }
57.     });
58.     add(buttonPanel, BorderLayout.SOUTH);
59. }
60.
61. /**
62.  * Adds a button to a container.
63.  * @param c the container
64.  * @param title the button title
65.  * @param listener the action listener for the button
66.  */
67. public void addButton(Container c, String title, ActionListener listener)
68. {
69.     JButton button = new JButton(title);
70.     c.add(button);
71.     button.addActionListener(listener);
72. }
73.
74. /**
75.  * Adds a bouncing ball to the panel and makes it bounce 1,000 times.
76.  */
77. public void addBall()
78. {
79.     try
80.     {
81.         Ball ball = new Ball();
82.         comp.add(ball);
83.
```

**Listing 14-1** Bounce.java (continued)

```
84.     for (int i = 1; i <= STEPS; i++)
85.     {
86.         ball.move(comp.getBounds());
87.         comp.paint(comp.getGraphics());
88.         Thread.sleep(DELAY);
89.     }
90. }
91. catch (InterruptedException e)
92. {
93. }
94. }
95.
96. private BallComponent comp;
97. public static final int DEFAULT_WIDTH = 450;
98. public static final int DEFAULT_HEIGHT = 350;
99. public static final int STEPS = 1000;
100. public static final int DELAY = 3;
101. }
```

**Listing 14-2** Ball.java

```
1. import java.awt.geom.*;
2.
3. /**
4.  * A ball that moves and bounces off the edges of a rectangle
5.  * @version 1.33 2007-05-17
6.  * @author Cay Horstmann
7.  */
8. public class Ball
9. {
10.     /**
11.      * Moves the ball to the next position, reversing direction if it hits one of the edges
12.      */
13.     public void move(Rectangle2D bounds)
14.     {
15.         x += dx;
16.         y += dy;
17.         if (x < bounds.getMinX())
18.         {
19.             x = bounds.getMinX();
20.             dx = -dx;
21.         }
22.         if (x + XSIZE >= bounds.getMaxX())
23.         {
24.             x = bounds.getMaxX() - XSIZE;
25.             dx = -dx;
26.         }

```



**Listing 14-2** Ball.java (continued)

```
27.     if (y < bounds.getMinY())
28.     {
29.         y = bounds.getMinY();
30.         dy = -dy;
31.     }
32.     if (y + YSIZE >= bounds.getMaxY())
33.     {
34.         y = bounds.getMaxY() - YSIZE;
35.         dy = -dy;
36.     }
37. }
38.
39. /**
40.  * Gets the shape of the ball at its current position.
41.  */
42. public Ellipse2D getShape()
43. {
44.     return new Ellipse2D.Double(x, y, XSIZE, YSIZE);
45. }
46.
47. private static final int XSIZE = 15;
48. private static final int YSIZE = 15;
49. private double x = 0;
50. private double y = 0;
51. private double dx = 1;
52. private double dy = 1;
53. }
```

**Listing 14-3** BallComponent.java

```
1. import java.awt.*;
2. import java.util.*;
3. import javax.swing.*;
4.
5. /**
6.  * The component that draws the balls.
7.  * @version 1.33 2007-05-17
8.  * @author Cay Horstmann
9.  */
10. public class BallComponent extends JPanel
11. {
12.     /**
13.      * Add a ball to the component.
14.      * @param b the ball to add
15.      */
16.     public void add(Ball b)
17.     {
```

**Listing 14–3** BallComponent.java (continued)

```

18.     balls.add(b);
19.   }
20.
21.   public void paintComponent(Graphics g)
22.   {
23.       super.paintComponent(g); // erase background
24.       Graphics2D g2 = (Graphics2D) g;
25.       for (Ball b : balls)
26.       {
27.           g2.fill(b.getShape());
28.       }
29.   }
30.
31.   private ArrayList<Ball> balls = new ArrayList<Ball>();
32. }

```

**API** java.lang.Thread 1.0

- static void sleep(long millis)  
sleeps for the given number of milliseconds.

*Parameters:*    millis                      The number of milliseconds to sleep

**Using Threads to Give Other Tasks a Chance**

We will make our bouncing-ball program more responsive by running the code that moves the ball in a separate thread. In fact, you will be able to launch multiple balls. Each of them is moved by its own thread. In addition, the AWT *event dispatch thread* continues running in parallel, taking care of user interface events. Because each thread gets a chance to run, the event dispatch thread has the opportunity to notice when a user clicks the Close button while the balls are bouncing. The thread can then process the “close” action.

We use ball-bouncing code as an example to give you a visual impression of the need for concurrency. In general, you will always want to be wary of any long-running computation. Your computation is likely to be a part of some bigger framework, such as a GUI or web framework. Whenever the framework calls one of your methods, there is usually an expectation of a quick return. If you need to do any task that takes a long time, you should use a separate thread.

Here is a simple procedure for running a task in a separate thread:

1. Place the code for the task into the `run` method of a class that implements the `Runnable` interface. That interface is very simple, with a single method:

```

public interface Runnable
{
    void run();
}

```

You simply implement a class, like this:

```
class MyRunnable implements Runnable
{
    public void run()
    {
        task code
    }
}
```

2. Construct an object of your class:  
`Runnable r = new MyRunnable();`
3. Construct a Thread object from the Runnable:  
`Thread t = new Thread(r);`
4. Start the thread:  
`t.start();`

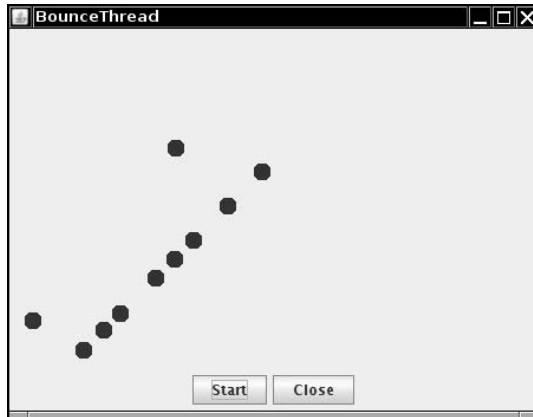
To make our bouncing-ball program into a separate thread, we need only implement a class `BallRunnable` and place the code for the animation inside the `run` method, as in the following code:

```
class BallRunnable implements Runnable
{
    . . .
    public void run()
    {
        try
        {
            for (int i = 1; i <= STEPS; i++)
            {
                ball.move(component.getBounds());
                component.repaint();
                Thread.sleep(DELAY);
            }
        }
        catch (InterruptedException exception)
        {
        }
    }
    . . .
}
```

Again, we need to catch an `InterruptedException` that the `sleep` method threatens to throw. We discuss this exception in the next section. Typically, interruption is used to request that a thread terminates. Accordingly, our `run` method exits when an `InterruptedException` occurs.

Whenever the Start button is clicked, the `addBall` method launches a new thread (see Figure 14–2):

```
Ball b = new Ball();
panel.add(b);
Runnable r = new BallRunnable(b, panel);
Thread t = new Thread(r);
t.start();
```



**Figure 14-2** Running multiple threads

That's all there is to it! You now know how to run tasks in parallel. The remainder of this chapter tells you how to control the interaction between threads.

The complete code is shown in Listing 14-4.



**NOTE:** You can also define a thread by forming a subclass of the Thread class, like this:

```
class MyThread extends Thread
{
    public void run()
    {
        task code
    }
}
```

Then you construct an object of the subclass and call its start method. However, this approach is no longer recommended. You should decouple the *task* that is to be run in parallel from the *mechanism* of running it. If you have many tasks, it is too expensive to create a separate thread for each one of them. Instead, you can use a thread pool—see “Executors” on page 778.



**CAUTION:** Do *not* call the run method of the Thread class or the Runnable object. Calling the run method directly merely executes the task in the *same* thread—no new thread is started. Instead, call the Thread.start method. It will create a new thread that executes the run method.

**Listing 14-4** BounceThread.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6.  * Shows animated bouncing balls.
7.  * @version 1.33 2007-05-17
8.  * @author Cay Horstmann
9.  */
10. public class BounceThread
11. {
12.     public static void main(String[] args)
13.     {
14.         EventQueue.invokeLater(new Runnable()
15.         {
16.             public void run()
17.             {
18.                 JFrame frame = new BounceFrame();
19.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20.                 frame.setVisible(true);
21.             }
22.         });
23.     }
24. }
25.
26. /**
27.  * A runnable that animates a bouncing ball.
28.  */
29. class BallRunnable implements Runnable
30. {
31.     /**
32.      * Constructs the runnable.
33.      * @aBall the ball to bounce
34.      * @aPanel the component in which the ball bounces
35.      */
36.     public BallRunnable(Ball aBall, Component aComponent)
37.     {
38.         ball = aBall;
39.         component = aComponent;
40.     }
41.
42.     public void run()
43.     {
44.         try
45.         {
46.             for (int i = 1; i <= STEPS; i++)
47.             {
48.                 ball.move(component.getBounds());
```

**Listing 14-4** BounceThread.java (continued)

```
49.         component.repaint();
50.         Thread.sleep(DELAY);
51.     }
52.     }
53.     catch (InterruptedException e)
54.     {
55.     }
56. }
57.
58. private Ball ball;
59. private Component component;
60. public static final int STEPS = 1000;
61. public static final int DELAY = 5;
62. }
63.
64. /**
65.  * The frame with panel and buttons.
66.  */
67. class BounceFrame extends JFrame
68. {
69.     /**
70.     * Constructs the frame with the component for showing the bouncing ball and Start and
71.     * Close buttons
72.     */
73.     public BounceFrame()
74.     {
75.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
76.         setTitle("BounceThread");
77.
78.         comp = new BallComponent();
79.         add(comp, BorderLayout.CENTER);
80.         JPanel buttonPanel = new JPanel();
81.         addButton(buttonPanel, "Start", new ActionListener()
82.         {
83.             public void actionPerformed(ActionEvent event)
84.             {
85.                 addBall();
86.             }
87.         });
88.
89.         addButton(buttonPanel, "Close", new ActionListener()
90.         {
91.             public void actionPerformed(ActionEvent event)
92.             {
93.                 System.exit(0);
94.             }
95.         });
96.         add(buttonPanel, BorderLayout.SOUTH);
97.     }
98. }
```

**Listing 14-4** BounceThread.java (continued)

```
99.  /**
100.   * Adds a button to a container.
101.   * @param c the container
102.   * @param title the button title
103.   * @param listener the action listener for the button
104.   */
105. public void addButton(Container c, String title, ActionListener listener)
106. {
107.     JButton button = new JButton(title);
108.     c.add(button);
109.     button.addActionListener(listener);
110. }
111.
112. /**
113.   * Adds a bouncing ball to the canvas and starts a thread to make it bounce
114.   */
115. public void addBall()
116. {
117.     Ball b = new Ball();
118.     comp.add(b);
119.     Runnable r = new BallRunnable(b, comp);
120.     Thread t = new Thread(r);
121.     t.start();
122. }
123.
124. private BallComponent comp;
125. public static final int DEFAULT_WIDTH = 450;
126. public static final int DEFAULT_HEIGHT = 350;
127. public static final int STEPS = 1000;
128. public static final int DELAY = 3;
129. }
```

**API** java.lang.Thread 1.0

- Thread(Runnable target)  
constructs a new thread that calls the run() method of the specified target.
- void start()  
starts this thread, causing the run() method to be called. This method will return immediately. The new thread runs concurrently.
- void run()  
calls the run method of the associated Runnable.

**API** java.lang.Runnable 1.0

- void run()  
must be overridden and supplied with instructions for the task that you want to have executed.

### Interrupting Threads

A thread terminates when its `run` method returns, by executing a return statement, after executing the last statement in the method body, or if an exception occurs that is not caught in the method. In the initial release of Java, there also was a `stop` method that another thread could call to terminate a thread. However, that method is now deprecated. We discuss the reason in the section “Why the `stop` and `suspend` Methods Are Deprecated” on page 762.

There is a way to *force* a thread to terminate. However, the `interrupt` method can be used to *request* termination of a thread.

When the `interrupt` method is called on a thread, the *interrupted status* of the thread is set. This is a `boolean` flag that is present in every thread. Each thread should occasionally check whether it has been interrupted.

To find out whether the interrupted status was set, first call the static `Thread.currentThread` method to get the current thread and then call the `isInterrupted` method:

```
while (!Thread.currentThread().isInterrupted() && more work to do)
{
    do more work
}
```

However, if a thread is blocked, it cannot check the interrupted status. This is where the `InterruptedException` comes in. When the `interrupt` method is called on a thread that blocks on a call such as `sleep` or `wait`, the blocking call is terminated by an `InterruptedException`. (There are blocking I/O calls that cannot be interrupted; you should consider interruptible alternatives. See Chapters 1 and 3 of Volume II for details.)

There is no language requirement that a thread that is interrupted should terminate. Interrupting a thread simply grabs its attention. The interrupted thread can decide how to react to the interruption. Some threads are so important that they should handle the exception and continue. But quite commonly, a thread will simply want to interpret an interruption as a request for termination. The `run` method of such a thread has the following form:

```
public void run()
{
    try
    {
        . . .
        while (!Thread.currentThread().isInterrupted() && more work to do)
        {
            do more work
        }
    }
    catch(InterruptedException e)
    {
        // thread was interrupted during sleep or wait
    }
    finally
    {
```



```

        cleanup, if required
    }
    // exiting the run method terminates the thread
}

```

The `isInterrupted` check is neither necessary nor useful if you call the `sleep` method (or another interruptible method) after every work iteration. If you call the `sleep` method when the interrupted status is set, it doesn't sleep. Instead, it clears the status (!) and throws an `InterruptedException`. Therefore, if your loop calls `sleep`, don't check the interrupted status. Instead, catch the `InterruptedException`, like this:

```

public void run()
{
    try
    {
        . . .
        while (more work to do)
        {
            do more work
            Thread.sleep(delay);
        }
    }
    catch(InterruptedException e)
    {
        // thread was interrupted during sleep
    }
    finally
    {
        cleanup, if required
    }
    // exiting the run method terminates the thread
}

```



**NOTE:** There are two very similar methods, `interrupted` and `isInterrupted`. The `interrupted` method is a static method that checks whether the *current* thread has been interrupted. Furthermore, calling the `interrupted` method *clears* the interrupted status of the thread. On the other hand, the `isInterrupted` method is an instance method that you can use to check whether any thread has been interrupted. Calling it does not change the interrupted status.

You'll find lots of published code in which the `InterruptedException` is squelched at a low level, like this:

```

void mySubTask()
{
    . . .
    try { sleep(delay); }
    catch (InterruptedException e) {} // DON'T IGNORE!
    . . .
}

```

Don't do that! If you can't think of anything good to do in the catch clause, you still have two reasonable choices:

- In the catch clause, call `Thread.currentThread().interrupt()` to set the interrupted status. Then the caller can test it.

```
void mySubTask()
{
    . . .
    try { sleep(delay); }
    catch (InterruptedException e) { Thread.currentThread().interrupt(); }
    . . .
}
```

- Or, even better, tag your method with `throws InterruptedException` and drop the try block. Then the caller (or, ultimately, the `run` method) can catch it.

```
void mySubTask() throws InterruptedException
{
    . . .
    sleep(delay);
    . . .
}
```

#### API `java.lang.Thread` 1.0

- `void interrupt()`  
sends an interrupt request to a thread. The interrupted status of the thread is set to true. If the thread is currently blocked by a call to `sleep`, then an `InterruptedException` is thrown.
- `static boolean interrupted()`  
tests whether the *current* thread (that is, the thread that is executing this instruction) has been interrupted. Note that this is a static method. The call has a side effect—it resets the interrupted status of the current thread to false.
- `boolean isInterrupted()`  
tests whether a thread has been interrupted. Unlike the static `interrupted` method, this call does not change the interrupted status of the thread.
- `static Thread currentThread()`  
returns the `Thread` object representing the currently executing thread.

### Thread States

Threads can be in one of six states:

- New
- Runnable
- Blocked
- Waiting
- Timed waiting
- Terminated

Each of these states is explained in the sections that follow.

To determine the current state of a thread, simply call the `getState` method.

### **New Threads**

When you create a thread with the `new` operator—for example, `new Thread(r)`—the thread is not yet running. This means that it is in the *new* state. When a thread is in the new state, the program has not started executing code inside of it. A certain amount of bookkeeping needs to be done before a thread can run.

### **Runnable Threads**

Once you invoke the `start` method, the thread is in the *runnable* state. A runnable thread may or may not actually be running. It is up to the operating system to give the thread time to run. (The Java specification does not call this a separate state, though. A running thread is still in the runnable state.)

Once a thread is running, it doesn't necessarily keep running. In fact, it is desirable if running threads occasionally pause so that other threads have a chance to run. The details of thread scheduling depend on the services that the operating system provides. Preemptive scheduling systems give each runnable thread a slice of time to perform its task. When that slice of time is exhausted, the operating system *preempts* the thread and gives another thread an opportunity to work (see Figure 14-4 on page 741). When selecting the next thread, the operating system takes into account the thread *priorities*—see “Thread Priorities” on page 733 for more information.

All modern desktop and server operating systems use preemptive scheduling. However, small devices such as cell phones may use cooperative scheduling. In such a device, a thread loses control only when it calls the `yield` method, or it is blocked or waiting.

On a machine with multiple processors, each processor can run a thread, and you can have multiple threads run in parallel. Of course, if there are more threads than processors, the scheduler still has to do time-slicing.

Always keep in mind that a runnable thread may or may not be running at any given time. (This is why the state is called “runnable” and not “running.”)

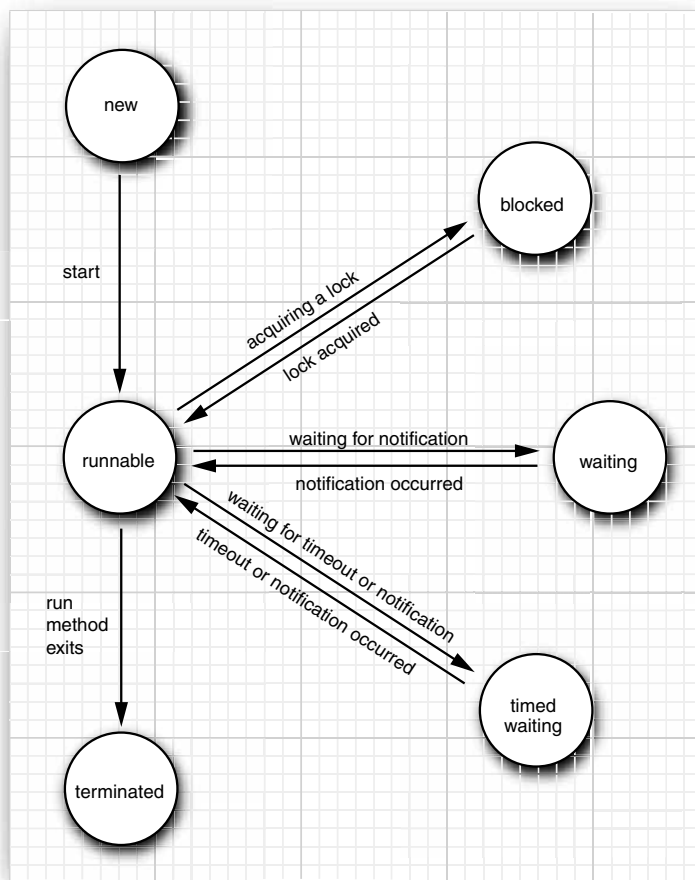
### **Blocked and Waiting Threads**

When a thread is blocked or waiting, it is temporarily inactive. It doesn't execute any code and it consumes minimal resources. It is up to the thread scheduler to reactivate it. The details depend on how the inactive state was reached.

- When the thread tries to acquire an intrinsic object lock (but not a `Lock` in the `java.util.concurrent` library) that is currently held by another thread, it becomes *blocked*. (We discuss `java.util.concurrent` locks in the section “Lock Objects” on page 742 and intrinsic object locks in the section “The synchronized Keyword” on page 750.) The thread becomes unblocked when all other threads have relinquished the lock and the thread scheduler has allowed this thread to hold it.
- When the thread waits for another thread to notify the scheduler of a condition, it enters the *waiting* state. We discuss conditions in the “Condition Objects” section beginning on page 745. This happens by calling the `Object.wait` or `Thread.join` method, or by waiting for a `Lock` or `Condition` in the `java.util.concurrent` library. In practice, the difference between the blocked and waiting state is not significant.
- Several methods have a timeout parameter. Calling them causes the thread to enter the *timed waiting* state. This state persists either until the timeout expired or the

appropriate notification has been received. Methods with timeout include `Thread.sleep` and the timed versions of `Object.wait`, `Thread.join`, `Lock.tryLock`, and `Condition.await`.

Figure 14–3 shows the states that a thread can have and the possible transitions from one state to another. When a thread is blocked or waiting (or, of course, when it terminates), another thread will be scheduled to run. When a thread is reactivated (for example, because its timeout has expired or it has succeeded in acquiring a lock), the scheduler checks to see if it has a higher priority than the currently running threads. If so, it preempts one of the current threads and picks a new thread to run.



**Figure 14–3** Thread states

### Terminated Threads

A thread is terminated for one of two reasons:

- It dies a natural death because the `run` method exits normally.
- It dies abruptly because an uncaught exception terminates the `run` method.

In particular, you can kill a thread by invoking its `stop` method. That method throws a `ThreadDeath` error object that kills the thread. However, the `stop` method is deprecated, and you should never call it in your own code.

#### API `java.lang.Thread` 1.0

- `void join()`  
waits for the specified thread to terminate.
- `void join(long millis)`  
waits for the specified thread to die or for the specified number of milliseconds to pass.
- `Thread.State getState()` 5.0  
gets the state of this thread; one of `NEW`, `RUNNABLE`, `BLOCKED`, `WAITING`, `TIMED_WAITING`, or `TERMINATED`.
- `void stop()`  
stops the thread. This method is deprecated.
- `void suspend()`  
suspends this thread's execution. This method is deprecated.
- `void resume()`  
resumes this thread. This method is only valid after `suspend()` has been invoked. This method is deprecated.

### Thread Properties

In the following sections, we discuss miscellaneous properties of threads: thread priorities, daemon threads, thread groups, and handlers for uncaught exceptions.

#### Thread Priorities

In the Java programming language, every thread has a *priority*. By default, a thread inherits the priority of the thread that constructed it. You can increase or decrease the priority of any thread with the `setPriority` method. You can set the priority to any value between `MIN_PRIORITY` (defined as 1 in the `Thread` class) and `MAX_PRIORITY` (defined as 10). `NORM_PRIORITY` is defined as 5.

Whenever the thread-scheduler has a chance to pick a new thread, it prefers threads with higher priority. However, thread priorities are *highly system dependent*. When the virtual machine relies on the thread implementation of the host platform, the Java thread priorities are mapped to the priority levels of the host platform, which may have more or fewer thread priority levels.

For example, Windows has seven priority levels. Some of the Java priorities will map to the same operating system level. In the Sun JVM for Linux, thread priorities are ignored altogether—all threads have the same priority.

Beginning programmers sometimes overuse thread priorities. There are few reasons ever to tweak priorities. You should certainly never structure your programs so that their correct functioning depends on priority levels.



**CAUTION:** If you do use priorities, you should be aware of a common beginner's error. If you have several threads with a high priority that don't become inactive, the lower-priority threads may *never* execute. Whenever the scheduler decides to run a new thread, it will choose among the highest-priority threads first, even though that may starve the lower-priority threads completely.



#### java.lang.Thread 1.0

- void setPriority(int newPriority)  
sets the priority of this thread. The priority must be between Thread.MIN\_PRIORITY and Thread.MAX\_PRIORITY. Use Thread.NORM\_PRIORITY for normal priority.
- static int MIN\_PRIORITY  
is the minimum priority that a Thread can have. The minimum priority value is 1.
- static int NORM\_PRIORITY  
is the default priority of a Thread. The default priority is 5.
- static int MAX\_PRIORITY  
is the maximum priority that a Thread can have. The maximum priority value is 10.
- static void yield()  
causes the currently executing thread to yield. If there are other runnable threads with a priority at least as high as the priority of this thread, they will be scheduled next. Note that this is a static method.

### Daemon Threads

You can turn a thread into a *daemon thread* by calling

```
t.setDaemon(true);
```

There is nothing demonic about such a thread. A daemon is simply a thread that has no other role in life than to serve others. Examples are timer threads that send regular “timer ticks” to other threads or threads that clean up stale cache entries. When only daemon threads remain, the virtual machine exits. There is no point in keeping the program running if all remaining threads are daemons.

Daemon threads are sometimes mistakenly used by beginners who don't want to think about shutdown actions. However, this can be dangerous. A daemon thread should never access a persistent resource such as a file or database since it can terminate at any time, even in the middle of an operation.



#### java.lang.Thread 1.0

- void setDaemon(boolean isDaemon)  
marks this thread as a daemon thread or a user thread. This method must be called before the thread is started.

### Handlers for Uncaught Exceptions

The run method of a thread cannot throw any checked exceptions, but it can be terminated by an unchecked exception. In that case, the thread dies.

However, there is no catch clause to which the exception can be propagated. Instead, just before the thread dies, the exception is passed to a handler for uncaught exceptions.

The handler must belong to a class that implements the `Thread.UncaughtExceptionHandler` interface. That interface has a single method,

```
void uncaughtException(Thread t, Throwable e)
```

As of Java SE 5.0, you can install a handler into any thread with the `setUncaughtExceptionHandler` method. You can also install a default handler for all threads with the static method `setDefaultUncaughtExceptionHandler` of the `Thread` class. A replacement handler might use the logging API to send reports of uncaught exceptions into a log file.

If you don't install a default handler, the default handler is `null`. However, if you don't install a handler for an individual thread, the handler is the thread's `ThreadGroup` object.



**NOTE:** A thread group is a collection of threads that can be managed together. By default, all threads that you create belong to the same thread group, but it is possible to establish other groupings. Since Java SE 5.0 introduced better features for operating on collections of threads, you should not use thread groups in your own programs.

The `ThreadGroup` class implements the `Thread.UncaughtExceptionHandler` interface. Its `uncaughtException` method takes the following action:

1. If the thread group has a parent, then the `uncaughtException` method of the parent group is called.
2. Otherwise, if the `Thread.getDefaultExceptionHandler` method returns a non-`null` handler, it is called.
3. Otherwise, if the `Throwable` is an instance of `ThreadDeath`, nothing happens.
4. Otherwise, the name of the thread and the stack trace of the `Throwable` are printed on `System.err`.

That is the stack trace that you have undoubtedly seen many times in your programs.

#### API `java.lang.Thread` 1.0

- `static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler)` 5.0
- `static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()` 5.0  
sets or gets the default handler for uncaught exceptions.
- `void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler)` 5.0
- `Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()` 5.0  
sets or gets the handler for uncaught exceptions. If no handler is installed, the thread group object is the handler.

**API** `java.lang.Thread.UncaughtExceptionHandler` 5.0

- `void uncaughtException(Thread t, Throwable e)`  
defined to log a custom report when a thread is terminated with an uncaught exception.

*Parameters:*

<code>t</code>	The thread that was terminated due to an uncaught exception
<code>e</code>	The uncaught exception object

**API** `java.lang.ThreadGroup` 1.0

- `void uncaughtException(Thread t, Throwable e)`  
calls this method of the parent thread group if there is a parent, or calls the default handler of the `Thread` class if there is a default handler, or otherwise prints a stack trace to the standard error stream. (However, if `e` is a `ThreadDeath` object, the stack trace is suppressed. `ThreadDeath` objects are generated by the deprecated `stop` method.)

**Synchronization**

In most practical multithreaded applications, two or more threads need to share access to the same data. What happens if two threads have access to the same object and each calls a method that modifies the state of the object? As you might imagine, the threads can step on each other's toes. Depending on the order in which the data were accessed, corrupted objects can result. Such a situation is often called a *race condition*.

**An Example of a Race Condition**

To avoid corruption of shared data by multiple threads, you must learn how to *synchronize the access*. In this section, you'll see what happens if you do not use synchronization. In the next section, you'll see how to synchronize data access.

In the next test program, we simulate a bank with a number of accounts. We randomly generate transactions that move money between these accounts. Each account has one thread. Each transaction moves a random amount of money from the account serviced by the thread to another random account.

The simulation code is straightforward. We have the class `Bank` with the method `transfer`. This method transfers some amount of money from one account to another. (We don't yet worry about negative account balances.) Here is the code for the `transfer` method of the `Bank` class.

```
public void transfer(int from, int to, double amount)
    // CAUTION: unsafe when called from multiple threads
{
    System.out.print(Thread.currentThread());
    accounts[from] -= amount;
    System.out.printf(" %10.2f from %d to %d", amount, from, to);
    accounts[to] += amount;
    System.out.printf(" Total Balance: %10.2f\n", getTotalBalance());
}
```



Here is the code for the `TransferRunnable` class. Its `run` method keeps moving money out of a fixed bank account. In each iteration, the `run` method picks a random target account and a random amount, calls `transfer` on the bank object, and then sleeps.

```
class TransferRunnable implements Runnable
{
    . . .
    public void run()
    {
        try
        {
            int toAccount = (int) (bank.size() * Math.random());
            double amount = maxAmount * Math.random();
            bank.transfer(fromAccount, toAccount, amount);
            Thread.sleep((int) (DELAY * Math.random()));
        }
        catch (InterruptedException e) {}
    }
}
```

When this simulation runs, we do not know how much money is in any one bank account at any time. But we do know that the total amount of money in all the accounts should remain unchanged because all we do is move money from one account to another.

At the end of each transaction, the `transfer` method recomputes the total and prints it.

This program never finishes. Just press CTRL+C to kill the program.

Here is a typical printout:

```
. . .
Thread[Thread-11,5,main] 588.48 from 11 to 44 Total Balance: 100000.00
Thread[Thread-12,5,main] 976.11 from 12 to 22 Total Balance: 100000.00
Thread[Thread-14,5,main] 521.51 from 14 to 22 Total Balance: 100000.00
Thread[Thread-13,5,main] 359.89 from 13 to 81 Total Balance: 100000.00
. . .
Thread[Thread-36,5,main] 401.71 from 36 to 73 Total Balance: 99291.06
Thread[Thread-35,5,main] 691.46 from 35 to 77 Total Balance: 99291.06
Thread[Thread-37,5,main] 78.64 from 37 to 3 Total Balance: 99291.06
Thread[Thread-34,5,main] 197.11 from 34 to 69 Total Balance: 99291.06
Thread[Thread-36,5,main] 85.96 from 36 to 4 Total Balance: 99291.06
. . .
Thread[Thread-4,5,main]Thread[Thread-33,5,main] 7.31 from 31 to 32 Total Balance:
99979.24
627.50 from 4 to 5 Total Balance: 99979.24
. . .
```

As you can see, something is very wrong. For a few transactions, the bank balance remains at \$100,000, which is the correct total for 100 accounts of \$1,000 each. But after some time, the balance changes slightly. When you run this program, you may find that errors happen quickly or it may take a very long time for the balance to become corrupted. This situation does not inspire confidence, and you would probably not want to deposit your hard-earned money in this bank.

The program in Listings 14–5 through 14–7 provides the complete source code. See if you can spot the problems with the code. We will unravel the mystery in the next section.

**Listing 14–5** UnsynchBankTest.java

```
1. /**
2.  * This program shows data corruption when multiple threads access a data structure.
3.  * @version 1.30 2004-08-01
4.  * @author Cay Horstmann
5.  */
6. public class UnsynchBankTest
7. {
8.     public static void main(String[] args)
9.     {
10.         Bank b = new Bank(NACCOUNTS, INITIAL_BALANCE);
11.         int i;
12.         for (i = 0; i < NACCOUNTS; i++)
13.         {
14.             TransferRunnable r = new TransferRunnable(b, i, INITIAL_BALANCE);
15.             Thread t = new Thread(r);
16.             t.start();
17.         }
18.     }
19.
20.     public static final int NACCOUNTS = 100;
21.     public static final double INITIAL_BALANCE = 1000;
22. }
```

**Listing 14–6** Bank.java

```
1. /**
2.  * A bank with a number of bank accounts.
3.  * @version 1.30 2004-08-01
4.  * @author Cay Horstmann
5.  */
6. public class Bank
7. {
8.     /**
9.     * Constructs the bank.
10.    * @param n the number of accounts
11.    * @param initialBalance the initial balance for each account
12.    */
13.    public Bank(int n, double initialBalance)
14.    {
15.        accounts = new double[n];
16.        for (int i = 0; i < accounts.length; i++)
17.            accounts[i] = initialBalance;
18.    }
```

**Listing 14-6** Bank.java (continued)

```
19.
20.  /**
21.   * Transfers money from one account to another.
22.   * @param from the account to transfer from
23.   * @param to the account to transfer to
24.   * @param amount the amount to transfer
25.   */
26.  public void transfer(int from, int to, double amount)
27.  {
28.      if (accounts[from] < amount) return;
29.      System.out.print(Thread.currentThread());
30.      accounts[from] -= amount;
31.      System.out.printf(" %10.2f from %d to %d", amount, from, to);
32.      accounts[to] += amount;
33.      System.out.printf(" Total Balance: %10.2f\n", getTotalBalance());
34.  }
35.
36.  /**
37.   * Gets the sum of all account balances.
38.   * @return the total balance
39.   */
40.  public double getTotalBalance()
41.  {
42.      double sum = 0;
43.
44.      for (double a : accounts)
45.          sum += a;
46.
47.      return sum;
48.  }
49.
50.  /**
51.   * Gets the number of accounts in the bank.
52.   * @return the number of accounts
53.   */
54.  public int size()
55.  {
56.      return accounts.length;
57.  }
58.
59.  private final double[] accounts;
60. }
```

**Listing 14-7** TransferRunnable.java

```
1. /**
2.  * A runnable that transfers money from an account to other accounts in a bank.
3.  * @version 1.30 2004-08-01
4.  * @author Cay Horstmann
5.  */
6. public class TransferRunnable implements Runnable
7. {
8.     /**
9.     * Constructs a transfer runnable.
10.    * @param b the bank between whose account money is transferred
11.    * @param from the account to transfer money from
12.    * @param max the maximum amount of money in each transfer
13.    */
14.    public TransferRunnable(Bank b, int from, double max)
15.    {
16.        bank = b;
17.        fromAccount = from;
18.        maxAmount = max;
19.    }
20.
21.    public void run()
22.    {
23.        try
24.        {
25.            while (true)
26.            {
27.                int toAccount = (int) (bank.size() * Math.random());
28.                double amount = maxAmount * Math.random();
29.                bank.transfer(fromAccount, toAccount, amount);
30.                Thread.sleep((int) (DELAY * Math.random()));
31.            }
32.        }
33.        catch (InterruptedException e)
34.        {
35.        }
36.    }
37.
38.    private Bank bank;
39.    private int fromAccount;
40.    private double maxAmount;
41.    private int DELAY = 10;
42. }
```

---

**The Race Condition Explained**

In the previous section, we ran a program in which several threads updated bank account balances. After a while, errors crept in and some amount of money was either lost or spontaneously created. This problem occurs when two threads are

simultaneously trying to update an account. Suppose two threads simultaneously carry out the instruction

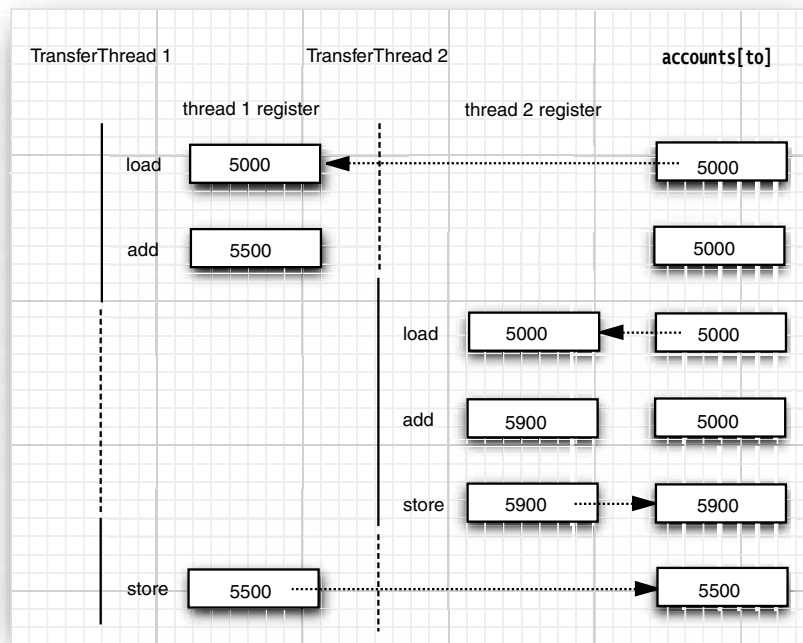
```
accounts[to] += amount;
```

The problem is that these are not *atomic* operations. The instruction might be processed as follows:

1. Load `accounts[to]` into a register.
2. Add `amount`.
3. Move the result back to `accounts[to]`.

Now, suppose the first thread executes Steps 1 and 2, and then it is preempted. Suppose the second thread awakens and updates the same entry in the account array. Then, the first thread awakens and completes its Step 3.

That action wipes out the modification of the other thread. As a result, the total is no longer correct. (See Figure 14-4.)



**Figure 14-4** Simultaneous access by two threads

Our test program detects this corruption. (Of course, there is a slight chance of false alarms if the thread is interrupted as it is performing the tests!)



NOTE: You can actually peek at the virtual machine bytecodes that execute each statement in our class. Run the command

```
javap -c -v Bank
```

to decompile the `Bank.class` file. For example, the line

```
accounts[to] += amount;
```

is translated into the following bytecodes:

```
aload_0
getfield    #2; //Field accounts:[D
iload_2
dup2
daload
dload_3
dadd
dastore
```

What these codes mean does not matter. The point is that the increment command is made up of several instructions, and the thread executing them can be interrupted at the point of any instruction.

What is the chance of this corruption occurring? We boosted the chance of observing the problem by interleaving the print statements with the statements that update the balance.

If you omit the print statements, the risk of corruption is quite a bit lower because each thread does so little work before going to sleep again, and it is unlikely that the scheduler will preempt it in the middle of the computation. However, the risk of corruption does not completely go away. If you run lots of threads on a heavily loaded machine, then the program will still fail even after you have eliminated the print statements. The failure may take a few minutes or hours or days to occur. Frankly, there are few things worse in the life of a programmer than an error that only manifests itself once every few days.

The real problem is that the work of the transfer method can be interrupted in the middle. If we could ensure that the method runs to completion before the thread loses control, then the state of the bank account object would never be corrupted.

### **Lock Objects**

Starting with Java SE 5.0, there are two mechanisms for protecting a code block from concurrent access. The Java language provides a `synchronized` keyword for this purpose, and Java SE 5.0 introduced the `ReentrantLock` class. The `synchronized` keyword automatically provides a lock as well as an associated “condition,” which makes it powerful and convenient for most cases that require explicit locking. However, we believe that it is easier to understand the `synchronized` keyword after you have seen locks and conditions in isolation. The `java.util.concurrent` framework provides separate classes for these fundamental mechanisms, which we explain here and in the section “Condition Objects” on page 745. Once you have understood these building blocks, we present the section “The `synchronized` Keyword” on page 750.

The basic outline for protecting a code block with a `ReentrantLock` is:

```
myLock.lock(); // a ReentrantLock object
try
{
    critical section
}
finally
{
    myLock.unlock(); // make sure the lock is unlocked even if an exception is thrown
}
```

This construct guarantees that only one thread at a time can enter the critical section. As soon as one thread locks the lock object, no other thread can get past the lock statement. When other threads call `lock`, they are deactivated until the first thread unlocks the lock object.



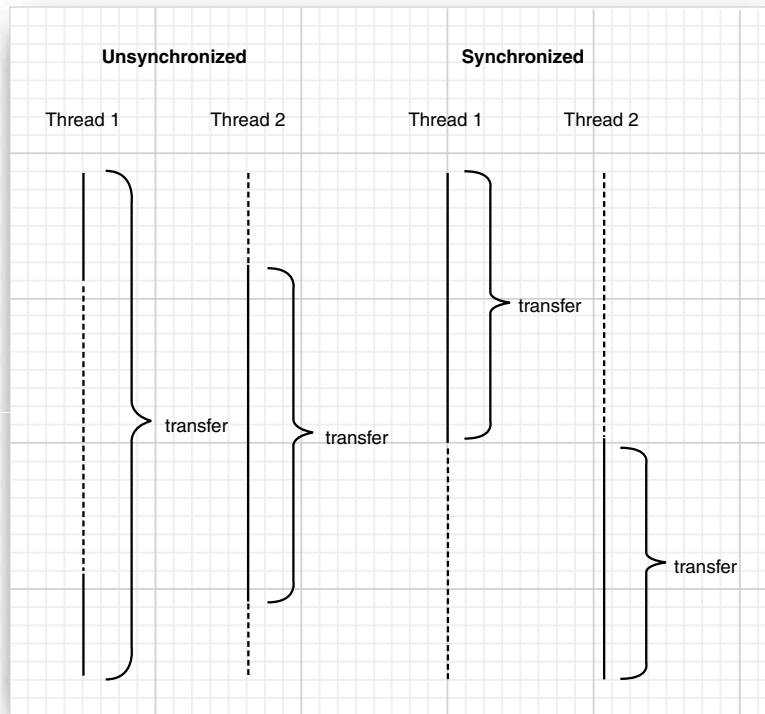
**CAUTION:** It is critically important that the `unlock` operation is enclosed in a `finally` clause. If the code in the critical section throws an exception, the lock must be unlocked. Otherwise, the other threads will be blocked forever.

Let us use a lock to protect the transfer method of the `Bank` class.

```
public class Bank
{
    public void transfer(int from, int to, int amount)
    {
        bankLock.lock();
        try
        {
            System.out.print(Thread.currentThread());
            accounts[from] -= amount;
            System.out.printf(" %10.2f from %d to %d", amount, from, to);
            accounts[to] += amount;
            System.out.printf(" Total Balance: %10.2f\n", getTotalBalance());
        }
        finally
        {
            bankLock.unlock();
        }
    }
    . . .
    private Lock bankLock = new ReentrantLock(); // ReentrantLock implements the Lock interface
}
```

Suppose one thread calls `transfer` and gets preempted before it is done. Suppose a second thread also calls `transfer`. The second thread cannot acquire the lock and is blocked in the call to the `lock` method. It is deactivated and must wait for the first thread to finish executing the `transfer` method. When the first thread unlocks the lock, then the second thread can proceed (see Figure 14–5).

Try it out. Add the locking code to the `transfer` method and run the program again. You can run it forever, and the bank balance will not become corrupted.



**Figure 14-5 Comparison of unsynchronized and synchronized threads**

Note that each Bank object has its own `ReentrantLock` object. If two threads try to access the same Bank object, then the lock serves to serialize the access. However, if two threads access different Bank objects, then each thread acquires a different lock and neither thread is blocked. This is as it should be, because the threads cannot interfere with another when they manipulate different Bank instances.

The lock is called *reentrant* because a thread can repeatedly acquire a lock that it already owns. The lock keeps a *hold count* that keeps track of the nested calls to the lock method. The thread has to call `unlock` for every call to `lock` in order to relinquish the lock. Because of this feature, code that is protected by a lock can call another method that uses the same locks.

For example, the `transfer` method calls the `getTotalBalance` method, which also locks the `bankLock` object, which now has a hold count of 2. When the `getTotalBalance` method exits, the hold count is back to 1. When the `transfer` method exits, the hold count is 0, and the thread relinquishes the lock.

In general, you will want to protect blocks of code that update or inspect a shared object. You are then assured that these operations run to completion before another thread can use the same object.



---

**X** CAUTION: You need to be careful that code in a critical section is not bypassed through the throwing of an exception. If an exception is thrown before the end of the section, then the finally clause will relinquish the lock but the object may be in a damaged state.

---

**API** `java.util.concurrent.locks.Lock 5.0`

- `void lock()`  
acquires this lock; blocks if the lock is currently owned by another thread.
- `void unlock()`  
releases this lock.

**API** `java.util.concurrent.locks.ReentrantLock 5.0`

- `ReentrantLock()`  
constructs a reentrant lock that can be used to protect a critical section.
- `ReentrantLock(boolean fair)`  
constructs a lock with the given fairness policy. A fair lock favors the thread that has been waiting for the longest time. However, this fairness guarantee can be a significant drag on performance. Therefore, by default, locks are not required to be fair.

---

**X** CAUTION: It sounds nicer to be fair, but fair locks are *a lot slower* than regular locks. You should only enable fair locking if you truly know what you are doing and have a specific reason why fairness is essential for your problem. Even if you use a fair lock, you have no guarantee that the thread scheduler is fair. If the thread scheduler chooses to neglect a thread that has been waiting a long time for the lock, then it doesn't get the chance to be treated fairly by the lock.

---

### Condition Objects

Often, a thread enters a critical section, only to discover that it can't proceed until a condition is fulfilled. You use a *condition object* to manage threads that have acquired a lock but cannot do useful work. In this section, we introduce the implementation of condition objects in the Java library. (For historical reasons, condition objects are often called *condition variables*.)

Let us refine our simulation of the bank. We do not want to transfer money out of an account that does not have the funds to cover the transfer. Note that we cannot use code like

```
if (bank.getBalance(from) >= amount)
    bank.transfer(from, to, amount);
```

It is entirely possible that the current thread will be deactivated between the successful outcome of the test and the call to transfer.

```
if (bank.getBalance(from) >= amount)
    // thread might be deactivated at this point
    bank.transfer(from, to, amount);
```

By the time the thread is running again, the account balance may have fallen below the withdrawal amount. You must make sure that no other thread can modify the balance between the test and the transfer action. You do so by protecting both the test and the transfer action with a lock:

```
public void transfer(int from, int to, int amount)
{
    bankLock.lock();
    try
    {
        while (accounts[from] < amount)
        {
            // wait
            . . .
        }
        // transfer funds
        . . .
    }
    finally
    {
        bankLock.unlock();
    }
}
```

Now, what do we do when there is not enough money in the account? We wait until some other thread has added funds. But this thread has just gained exclusive access to the `bankLock`, so no other thread has a chance to make a deposit. This is where condition objects come in.

A lock object can have one or more associated condition objects. You obtain a condition object with the `newCondition` method. It is customary to give each condition object a name that evokes the condition that it represents. For example, here we set up a condition object to represent the “sufficient funds” condition.

```
class Bank
{
    public Bank()
    {
        . . .
        sufficientFunds = bankLock.newCondition();
    }
    . . .
    private Condition sufficientFunds;
}
```

If the transfer method finds that sufficient funds are not available, it calls `sufficientFunds.await()`;

The current thread is now deactivated and gives up the lock. This lets in another thread that can, we hope, increase the account balance.

There is an essential difference between a thread that is waiting to acquire a lock and a thread that has called `await`. Once a thread calls the `await` method, it enters a *wait set* for that condition. The thread is *not* made runnable when the lock is available. Instead, it stays deactivated until another thread has called the `signalAll` method on the same condition.

When another thread transfers money, then it should call

```
sufficientFunds.signalAll();
```

This call reactivates all threads that are waiting for the condition. When the threads are removed from the wait set, they are again runnable and the scheduler will eventually activate them again. At that time, they will attempt to reenter the object. As soon as the lock is available, one of them will acquire the lock *and continue where it left off*, returning from the call to `await`.

At this time, the thread should test the condition again. There is no guarantee that the condition is now fulfilled—the `signalAll` method merely signals to the waiting threads that it *may be* fulfilled at this time and that it is worth checking for the condition again.



NOTE: In general, a call to `await` should be inside a loop of the form

```
while (!ok to proceed)
    condition.await();
```

It is crucially important that *some* other thread calls the `signalAll` method eventually. When a thread calls `await`, it has no way of reactivating itself. It puts its faith in the other threads. If none of them bother to reactivate the waiting thread, it will never run again. This can lead to unpleasant *deadlock* situations. If all other threads are blocked and the last active thread calls `await` without unblocking one of the others, then it also blocks. No thread is left to unblock the others, and the program hangs.

When should you call `signalAll`? The rule of thumb is to call `signalAll` whenever the state of an object changes in a way that might be advantageous to waiting threads. For example, whenever an account balance changes, the waiting threads should be given another chance to inspect the balance. In our example, we call `signalAll` when we have finished the funds transfer.

```
public void transfer(int from, int to, int amount)
{
    bankLock.lock();
    try
    {
        while (accounts[from] < amount)
            sufficientFunds.await();
        // transfer funds
        . . .
        sufficientFunds.signalAll();
    }
    finally
    {
        bankLock.unlock();
    }
}
```

Note that the call to `signalAll` does not immediately activate a waiting thread. It only unblocks the waiting threads so that they can compete for entry into the object after the current thread has exited the synchronized method.

Another method, `signal`, unblocks only a single thread from the wait set, chosen at random. That is more efficient than unblocking all threads, but there is a danger. If the randomly chosen thread finds that it still cannot proceed, then it becomes blocked again. If no other thread calls `signal` again, then the system deadlocks.



**CAUTION:** A thread can only call `await`, `signalAll`, or `signal` on a condition when it owns the lock of the condition.

If you run the sample program in Listing 14–8, you will notice that nothing ever goes wrong. The total balance stays at \$100,000 forever. No account ever has a negative balance. (Again, you need to press CTRL+C to terminate the program.) You may also notice that the program runs a bit slower—this is the price you pay for the added bookkeeping involved in the synchronization mechanism.

In practice, using conditions correctly can be quite challenging. Before you start implementing your own condition objects, you should consider using one of the constructs described in “Synchronizers” on page 785.

**Listing 14–8** Bank.java

```

1. import java.util.concurrent.locks.*;
2.
3. /**
4.  * A bank with a number of bank accounts that uses locks for serializing access.
5.  * @version 1.30 2004-08-01
6.  * @author Cay Horstmann
7.  */
8. public class Bank
9. {
10.     /**
11.      * Constructs the bank.
12.      * @param n the number of accounts
13.      * @param initialBalance the initial balance for each account
14.      */
15.     public Bank(int n, double initialBalance)
16.     {
17.         accounts = new double[n];
18.         for (int i = 0; i < accounts.length; i++)
19.             accounts[i] = initialBalance;
20.         bankLock = new ReentrantLock();
21.         sufficientFunds = bankLock.newCondition();
22.     }
23.
24.     /**
25.      * Transfers money from one account to another.
26.      * @param from the account to transfer from
27.      * @param to the account to transfer to
28.      * @param amount the amount to transfer
29.      */

```

**Listing 14-8** Bank.java (continued)

```
30. public void transfer(int from, int to, double amount) throws InterruptedException
31. {
32.     bankLock.lock();
33.     try
34.     {
35.         while (accounts[from] < amount)
36.             sufficientFunds.await();
37.         System.out.print(Thread.currentThread());
38.         accounts[from] -= amount;
39.         System.out.printf(" %10.2f from %d to %d", amount, from, to);
40.         accounts[to] += amount;
41.         System.out.printf(" Total Balance: %10.2f\n", getTotalBalance());
42.         sufficientFunds.signalAll();
43.     }
44.     finally
45.     {
46.         bankLock.unlock();
47.     }
48. }
49.
50. /**
51.  * Gets the sum of all account balances.
52.  * @return the total balance
53.  */
54. public double getTotalBalance()
55. {
56.     bankLock.lock();
57.     try
58.     {
59.         double sum = 0;
60.
61.         for (double a : accounts)
62.             sum += a;
63.
64.         return sum;
65.     }
66.     finally
67.     {
68.         bankLock.unlock();
69.     }
70. }
71.
72. /**
73.  * Gets the number of accounts in the bank.
74.  * @return the number of accounts
75.  */
76. public int size()
77. {
78.     return accounts.length;
79. }
```

**Listing 14–8** Bank.java (continued)

```

80.
81. private final double[] accounts;
82. private Lock bankLock;
83. private Condition sufficientFunds;
84. }

```

**API** `java.util.concurrent.locks.Lock` 5.0

- `Condition newCondition()`  
returns a condition object that is associated with this lock.

**API** `java.util.concurrent.locks.Condition` 5.0

- `void await()`  
puts this thread on the wait set for this condition.
- `void signalAll()`  
unblocks all threads in the wait set for this condition.
- `void signal()`  
unblocks one randomly selected thread in the wait set for this condition.

**The synchronized Keyword**

In the preceding sections, you saw how to use `Lock` and `Condition` objects. Before going any further, let us summarize the key points about locks and conditions:

- A lock protects sections of code, allowing only one thread to execute the code at a time.
- A lock manages threads that are trying to enter a protected code segment.
- A lock can have one or more associated condition objects.
- Each condition object manages threads that have entered a protected code section but that cannot proceed.

The `Lock` and `Condition` interfaces were added to Java SE 5.0 to give programmers a high degree of control over locking. However, in most situations, you don't need that control, and you can use a mechanism that is built into the Java language. Ever since version 1.0, *every object* in Java has an intrinsic lock. If a method is declared with the `synchronized` keyword, then the object's lock protects the entire method. That is, to call the method, a thread must acquire the intrinsic object lock.

In other words,

```

public synchronized void method()
{
    method body
}

```

is the equivalent of

```

public void method()
{

```

```

    this.intrinsicLock.lock();
    try
    {
        method body
    }
    finally { this.intrinsicLock.unlock(); }
}

```

For example, instead of using an explicit lock, we can simply declare the `transfer` method of the `Bank` class as synchronized.

The intrinsic object lock has a single associated condition. The `wait` method adds a thread to the wait set, and the `notifyAll/notify` methods unblock waiting threads. In other words, calling `wait` or `notifyAll` is the equivalent of

```

intrinsicCondition.await();
intrinsicCondition.signalAll();

```



**NOTE:** The `wait`, `notifyAll`, and `notify` methods are final methods of the `Object` class. The `Condition` methods had to be named `await`, `signalAll`, and `signal` so that they don't conflict with those methods.

For example, you can implement the `Bank` class in Java like this:

```

class Bank
{
    public synchronized void transfer(int from, int to, int amount) throws InterruptedException
    {
        while (accounts[from] < amount)
            wait(); // wait on intrinsic object lock's single condition
        accounts[from] -= amount;
        accounts[to] += amount;
        notifyAll(); // notify all threads waiting on the condition
    }
    public synchronized double getTotalBalance() { . . . }
    private double[] accounts;
}

```

As you can see, using the `synchronized` keyword yields code that is much more concise. Of course, to understand this code, you have to know that each object has an intrinsic lock, and that the lock has an intrinsic condition. The lock manages the threads that try to enter a synchronized method. The condition manages the threads that have called `wait`.



**TIP:** Synchronized methods are relatively straightforward. However, beginners often struggle with conditions. Before you use `wait/notifyAll`, you should consider using one of the constructs described in “Synchronizers” on page 785.

It is also legal to declare static methods as synchronized. If such a method is called, it acquires the intrinsic lock of the associated class object. For example, if the `Bank` class has a static synchronized method, then the lock of the `Bank.class` object is locked when it is called. As a result, no other thread can call this or any other synchronized static method of the same class.

The intrinsic locks and conditions have some limitations. Among them:

- You cannot interrupt a thread that is trying to acquire a lock.
- You cannot specify a timeout when trying to acquire a lock.
- Having a single condition per lock can be inefficient.

What should you use in your code—Lock and Condition objects or synchronized methods? Here is our recommendation:

- It is best to use neither Lock/Condition nor the synchronized keyword. In many situations, you can use one of the mechanisms of the `java.util.concurrent` package that do all the locking for you. For example, in “Blocking Queues” on page 764, you will see how to use a blocking queue to synchronize threads that work on a common task.
- If the `synchronized` keyword works for your situation, by all means, use it. You write less code and have less room for error. Listing 14–9 shows the bank example, implemented with synchronized methods.
- Use Lock/Condition if you specifically need the additional power that these constructs give you.

**Listing 14–9** Bank.java

```

1. /**
2.  * A bank with a number of bank accounts that uses synchronization primitives.
3.  * @version 1.30 2004-08-01
4.  * @author Cay Horstmann
5.  */
6. public class Bank
7. {
8.     /**
9.      * Constructs the bank.
10.     * @param n the number of accounts
11.     * @param initialBalance the initial balance for each account
12.     */
13.     public Bank(int n, double initialBalance)
14.     {
15.         accounts = new double[n];
16.         for (int i = 0; i < accounts.length; i++)
17.             accounts[i] = initialBalance;
18.     }
19.
20.     /**
21.     * Transfers money from one account to another.
22.     * @param from the account to transfer from
23.     * @param to the account to transfer to
24.     * @param amount the amount to transfer
25.     */
26.     public synchronized void transfer(int from, int to, double amount) throws InterruptedException
27.     {

```



**Listing 14-9** Bank.java (continued)

```
28.     while (accounts[from] < amount)
29.         wait();
30.     System.out.print(Thread.currentThread());
31.     accounts[from] -= amount;
32.     System.out.printf(" %10.2f from %d to %d", amount, from, to);
33.     accounts[to] += amount;
34.     System.out.printf(" Total Balance: %10.2f\n", getTotalBalance());
35.     notifyAll();
36. }
37.
38. /**
39.  * Gets the sum of all account balances.
40.  * @return the total balance
41.  */
42. public synchronized double getTotalBalance()
43. {
44.     double sum = 0;
45.
46.     for (double a : accounts)
47.         sum += a;
48.
49.     return sum;
50. }
51.
52. /**
53.  * Gets the number of accounts in the bank.
54.  * @return the number of accounts
55.  */
56. public int size()
57. {
58.     return accounts.length;
59. }
60.
61. private final double[] accounts;
62. }
```

**API** java.lang.Object 1.0

- void notifyAll()  
unblocks the threads that called wait on this object. This method can only be called from within a synchronized method or block. The method throws an `IllegalMonitorStateException` if the current thread is not the owner of the object's lock.
- void notify()  
unblocks one randomly selected thread among the threads that called wait on this object. This method can only be called from within a synchronized method or block. The method throws an `IllegalMonitorStateException` if the current thread is not the owner of the object's lock.

- `void wait()`  
causes a thread to wait until it is notified. This method can only be called from within a synchronized method. It throws an `IllegalMonitorStateException` if the current thread is not the owner of the object's lock.
- `void wait(long millis)`
- `void wait(long millis, int nanos)`  
causes a thread to wait until it is notified or until the specified amount of time has passed. These methods can only be called from within a synchronized method. They throw an `IllegalMonitorStateException` if the current thread is not the owner of the object's lock.

*Parameters:*    `millis`                    The number of milliseconds  
                  `nanos`                        The number of nanoseconds, < 1,000,000

### **Synchronized Blocks**

As we just discussed, every Java object has a lock. A thread can acquire the lock by calling a synchronized method. There is a second mechanism for acquiring the lock, by entering a *synchronized block*. When a thread enters a block of the form

```
synchronized (obj) // this is the syntax for a synchronized block
{
    critical section
}
```

then it acquires the lock for `obj`.

You will sometimes find “ad hoc” locks, such as

```
public class Bank
{
    public void transfer(int from, int to, int amount)
    {
        synchronized (lock) // an ad-hoc lock
        {
            accounts[from] -= amount;
            accounts[to] += amount;
        }
        System.out.println(. . .);
    }
    . . .
    private double[] accounts;
    private Object lock = new Object();
}
```

Here, the lock object is created only to use the lock that every Java object possesses.

Sometimes, programmers use the lock of an object to implement additional atomic operations, a practice known as *client-side locking*. Consider, for example, the `Vector` class, a list whose methods are synchronized. Now suppose we stored our bank balances in a `Vector<Double>`. Here is a naive implementation of a transfer method:

```
public void transfer(Vector<Double> accounts, int from, int to, int amount) // ERROR
{
    accounts.set(from, accounts.get(from) - amount);
```

```
        accounts.set(to, accounts.get(to) + amount);
        System.out.println( . . . );
    }
}
```

The `get` and `set` methods of the `Vector` class are synchronized, but that doesn't help us. It is entirely possible for a thread to be preempted in the `transfer` method after the first call to `get` has been completed. Another thread may then store a different value into the same position. However, we can hijack the lock:

```
public void transfer(Vector<Double> accounts, int from, int to, int amount)
{
    synchronized (accounts)
    {
        accounts.set(from, accounts.get(from) - amount);
        accounts.set(to, accounts.get(to) + amount);
    }
    System.out.println( . . . );
}
```

This approach works, but it is entirely dependent on the fact that the `Vector` class uses the intrinsic lock for all of its mutator methods. However, is this really a fact? The documentation of the `Vector` class makes no such promise. You have to carefully study the source code and hope that future versions do not introduce unsynchronized mutators. As you can see, client-side locking is very fragile and not generally recommended.

### **The Monitor Concept**

Locks and conditions are powerful tools for thread synchronization, but they are not very object oriented. For many years, researchers have looked for ways to make multithreading safe without forcing programmers to think about explicit locks. One of the most successful solutions is the *monitor* concept that was pioneered by Per Brinch Hansen and Tony Hoare in the 1970s. In the terminology of Java, a monitor has these properties:

- A monitor is a class with only private fields.
- Each object of that class has an associated lock.
- All methods are locked by that lock. In other words, if a client calls `obj.method()`, then the lock for `obj` is automatically acquired at the beginning of the method call and relinquished when the method returns. Because all fields are private, this arrangement ensures that no thread can access the fields while another thread manipulates them.
- The lock can have any number of associated conditions.

Earlier versions of monitors had a single condition, with a rather elegant syntax. You can simply call `await accounts[from] >= balance` without using an explicit condition variable. However, research showed that indiscriminate retesting of conditions can be inefficient. This problem is solved with explicit condition variables, each managing a separate set of threads.

The Java designers loosely adapted the monitor concept. *Every object* in Java has an intrinsic lock and an intrinsic condition. If a method is declared with the `synchronized` keyword, then it acts like a monitor method. The condition variable is accessed by calling `wait/notifyAll/notify`.

However, a Java object differs from a monitor in three important ways, compromising thread safety:

- Fields are not required to be `private`.
- Methods are not required to be synchronized.
- The intrinsic lock is available to clients.

This disrespect for security enraged Per Brinch Hansen. In a scathing review of the multithreading primitives in Java, he wrote: “It is astounding to me that Java’s insecure parallelism is taken seriously by the programming community, a quarter of a century after the invention of monitors and Concurrent Pascal. It has no merit.” [Java’s Insecure Parallelism, *ACM SIGPLAN Notices* 34:38–45, April 1999.]

### Volatile Fields

Sometimes, it seems excessive to pay the cost of synchronization just to read or write an instance field or two. After all, what can go wrong? Unfortunately, with modern processors and compilers, there is plenty of room for error:

- Computers with multiple processors can temporarily hold memory values in registers or local memory caches. As a consequence, threads running in different processors may see different values for the same memory location!
- Compilers can reorder instructions for maximum throughput. Compilers won’t choose an ordering that changes the meaning of the code, but they make the assumption that memory values are only changed when there are explicit instructions in the code. However, a memory value can be changed by another thread!

If you use locks to protect code that can be accessed by multiple threads, then you won’t have these problems. Compilers are required to respect locks by flushing local caches as necessary and not inappropriately reordering instructions. The details are explained in the Java Memory Model and Thread Specification developed by JSR 133 (see <http://www.jcp.org/en/jsr/detail?id=133>). Much of the specification is highly complex and technical, but the document also contains a number of clearly explained examples. A more accessible overview article by Brian Goetz is available at <http://www-106.ibm.com/developerworks/java/library/j-jtp02244.html>.



NOTE: Brian Goetz coined the following “synchronization motto”: “If you write a variable which may next be read by another thread, or you read a variable which may have last been written by another thread, you must use synchronization.”

The `volatile` keyword offers a lock-free mechanism for synchronizing access to an instance field. If you declare a field as `volatile`, then the compiler and the virtual machine take into account that the field may be concurrently updated by another thread.

For example, suppose an object has a `boolean` flag `done` that is set by one thread and queried by another thread. As we already discussed, you can use a lock:

```
public synchronized boolean isDone() { return done; }
public synchronized void setDone() { done = true; }
private boolean done;
```

Perhaps it is not a good idea to use the intrinsic object lock. The `isDone` and `setDone` methods can block if another thread has locked the object. If that is a concern, one can use a separate lock just for this variable. But this is getting to be a lot of trouble.

In this case, it is reasonable to declare the field as `volatile`:

```
public boolean isDone() { return done; }
public void setDone() { done = true; }
private volatile boolean done;
```



**CAUTION:** Volatile variables do not provide any atomicity. For example, the method

```
public void flipDone() { done = !done; } // not atomic
```

is not guaranteed to flip the value of the field.

In this very simple case, there is a third possibility, to use an `AtomicBoolean`. This class has methods `get` and `set` that are guaranteed to be atomic (as if they were synchronized). The implementation uses efficient machine-level instructions that guarantee atomicity without using locks. There are a number of wrapper classes in the `java.util.concurrent.atomic` package for atomic integers, floating point numbers, arrays, and so on. These classes are intended for systems programmers who produce concurrency utilities, not for the application programmer.

In summary, concurrent access to a field is safe in these three conditions:

- The field is `final`, and it is accessed after the constructor has completed.
- Every access to the field is protected by a common lock.
- The field is `volatile`.



**NOTE:** Prior to Java SE 5.0, the semantics of `volatile` were rather permissive. The language designers attempted to give implementors leeway in optimizing the performance of code that uses volatile fields. However, the old specification was so complex that implementors didn't always follow it, and it allowed confusing and undesirable behavior, such as immutable objects that weren't truly immutable.

### Deadlocks

Locks and conditions cannot solve all problems that might arise in multithreading. Consider the following situation:

Account 1: \$200

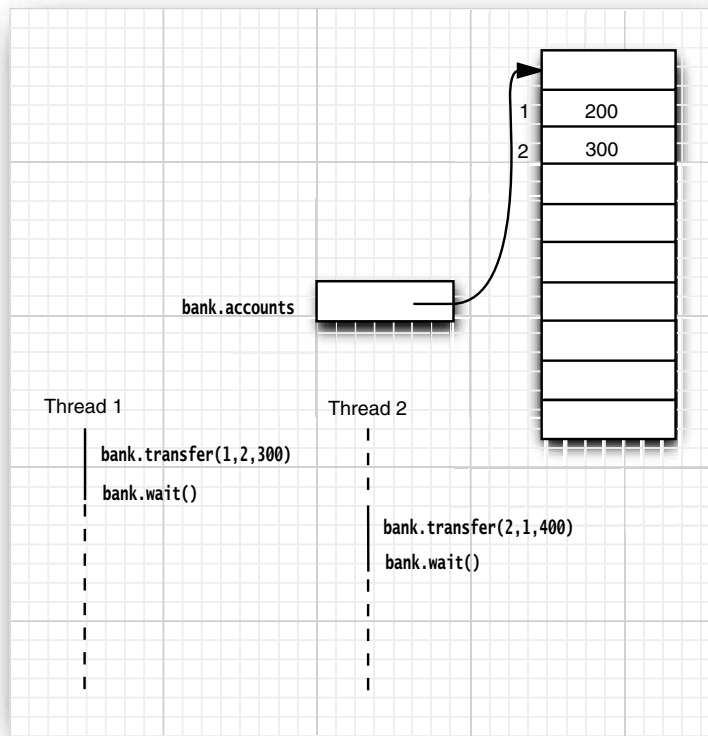
Account 2: \$300

Thread 1: Transfer \$300 from Account 1 to Account 2

Thread 2: Transfer \$400 from Account 2 to Account 1

As Figure 14–6 indicates, Threads 1 and 2 are clearly blocked. Neither can proceed because the balances in Accounts 1 and 2 are insufficient.

Is it possible that all threads are blocked because each is waiting for more money? Such a situation is called a *deadlock*.



**Figure 14-6** A deadlock situation

In our program, a deadlock cannot occur for a simple reason. Each transfer amount is for, at most, \$1,000. Because there are 100 accounts and a total of \$100,000 in them, at least one of the accounts must have more than \$1,000 at any time. The thread moving money out of that account can therefore proceed.

But if you change the `run` method of the threads to remove the \$1,000 transaction limit, deadlocks can occur quickly. Try it out. Set `NACCOUNTS` to 10. Construct each transfer runnable with a `max` value of `2 * INITIAL_BALANCE` and run the program. The program will run for a while and then hang.



**TIP:** When the program hangs, type `CTRL+\`. You will get a thread dump that lists all threads. Each thread has a stack trace, telling you where it is currently blocked. Alternatively, run `jconsole`, as described in Chapter 11, and consult the Threads panel (see Figure 14-7).

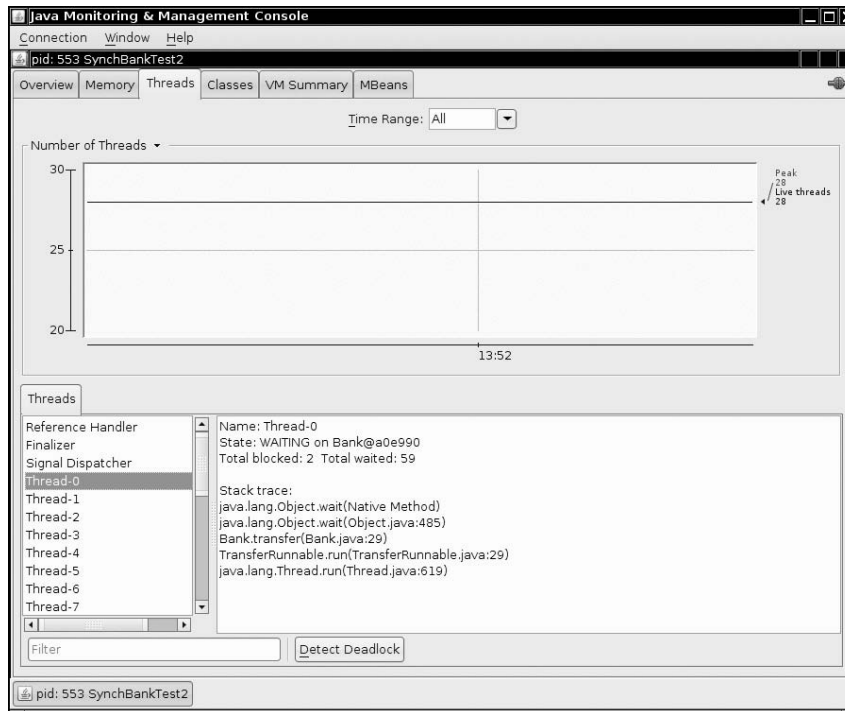


Figure 14-7 The Threads panel in jconsole

Another way to create a deadlock is to make the  $i$ 'th thread responsible for putting money into the  $i$ 'th account, rather than for taking it out of the  $i$ 'th account. In this case, there is a chance that all threads will gang up on one account, each trying to remove more money from it than it contains. Try it out. In the `SynchBankTest` program, turn to the `run` method of the `TransferRunnable` class. In the call to `transfer`, flip `fromAccount` and `toAccount`. Run the program and see how it deadlocks almost immediately.

Here is another situation in which a deadlock can occur easily: Change the `signalAll` method to `signal` in the `SynchBankTest` program. You will find that the program hangs eventually. (Again, it is best to set `NACCOUNTS` to 10 to observe the effect more quickly.) Unlike `signalAll`, which notifies all threads that are waiting for added funds, the `signal` method unblocks only one thread. If that thread can't proceed, all threads can be blocked. Consider the following sample scenario of a developing deadlock.

Account 1: \$1,990

All other accounts: \$990 each

Thread 1: Transfer \$995 from Account 1 to Account 2

All other threads: Transfer \$995 from their account to another account

Clearly, all threads but Thread 1 are blocked, because there isn't enough money in their accounts.

Thread 1 proceeds. Afterward, we have the following situation:

Account 1: \$995

Account 2: \$1,985

All other accounts: \$990 each

Then, Thread 1 calls `signal`. The `signal` method picks a thread at random to unblock. Suppose it picks Thread 3. That thread is awakened, finds that there isn't enough money in its account, and calls `await` again. But Thread 1 is still running. A new random transaction is generated, say,

Thread 1: Transfer \$997 to from Account 1 to Account 2

Now, Thread 1 also calls `await`, and *all* threads are blocked. The system has deadlocked.

The culprit here is the call to `signal`. It only unblocks one thread, and it may not pick the thread that is essential to make progress. (In our scenario, Thread 2 must proceed to take money out of Account 2.)

Unfortunately, there is nothing in the Java programming language to avoid or break these deadlocks. You must design your program to ensure that a deadlock situation cannot occur.

### **Lock Testing and Timeouts**

A thread blocks indefinitely when it calls the `lock` method to acquire a lock that is owned by another thread. You can be more cautious about acquiring a lock. The `tryLock` method tries to acquire a lock and returns `true` if it was successful. Otherwise, it immediately returns `false`, and the thread can go off and do something else.

```
if (myLock.tryLock())
    // now the thread owns the lock
    try { . . . }
    finally { myLock.unlock(); }
else
    // do something else
```

You can call `tryLock` with a timeout parameter, like this:

```
if (myLock.tryLock(100, TimeUnit.MILLISECONDS)) . . .
```

`TimeUnit` is an enumeration with values `SECONDS`, `MILLISECONDS`, `MICROSECONDS`, and `NANOSECONDS`.

The `lock` method cannot be interrupted. If a thread is interrupted while it is waiting to acquire a lock, the interrupted thread continues to be blocked until the lock is available. If a deadlock occurs, then the `lock` method can never terminate.

However, if you call `tryLock` with a timeout, then an `InterruptedException` is thrown if the thread is interrupted while it is waiting. This is clearly a useful feature because it allows a program to break up deadlocks.

You can also call the `lockInterruptibly` method. It has the same meaning as `tryLock` with an infinite timeout.

When you wait on a condition, you can also supply a timeout:

```
myCondition.await(100, TimeUnit.MILLISECONDS)
```



The `await` method returns if another thread has activated this thread by calling `signalAll` or `signal`, or if the timeout has elapsed, or if the thread was interrupted.

The `await` methods throw an `InterruptedException` if the waiting thread is interrupted. In the (perhaps unlikely) case that you'd rather continue waiting, use the `awaitUninterruptibly` method instead.

**API** `java.util.concurrent.locks.Lock` 5.0

- `boolean tryLock()`  
tries to acquire the lock without blocking; returns `true` if it was successful. This method grabs the lock if it is available even if it has a fair locking policy and other threads have been waiting.
- `boolean tryLock(long time, TimeUnit unit)`  
tries to acquire the lock, blocking no longer than the given time; returns `true` if it was successful.
- `void lockInterruptibly()`  
acquires the lock, blocking indefinitely. If the thread is interrupted, throws an `InterruptedException`.

**API** `java.util.concurrent.locks.Condition` 5.0

- `boolean await(long time, TimeUnit unit)`  
enters the wait set for this condition, blocking until the thread is removed from the wait set or the given time has elapsed. Returns `false` if the method returned because the time elapsed, `true` otherwise.
- `void awaitUninterruptibly()`  
enters the wait set for this condition, blocking until the thread is removed from the wait set. If the thread is interrupted, this method does not throw an `InterruptedException`.

**Read/Write Locks**

The `java.util.concurrent.locks` package defines two lock classes, the `ReentrantLock` that we already discussed and the `ReentrantReadWriteLock` class. The latter is useful when there are many threads that read from a data structure and fewer threads that modify it. In that situation, it makes sense to allow shared access for the readers. Of course, a writer must still have exclusive access.

Here are the steps that are necessary to use read/write locks:

1. Construct a `ReentrantReadWriteLock` object:  

```
private ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
```
2. Extract read and write locks:  

```
private Lock readLock = rwl.readLock();  
private Lock writeLock = rwl.writeLock();
```
3. Use the read lock in all accessors:  

```
public double getTotalBalance()  
{
```

```

        readLock.lock();
        try { . . . }
        finally { readLock.unlock(); }
    }

```

4. Use the write lock in all mutators:

```

public void transfer(. . .)
{
    writeLock.lock();
    try { . . . }
    finally { writeLock.unlock(); }
}

```

**API** `java.util.concurrent.locks.ReentrantReadWriteLock` 5.0

- Lock `readLock()`  
gets a read lock that can be acquired by multiple readers, excluding all writers.
- Lock `writeLock()`  
gets a write lock that excludes all other readers and writers.

**Why the stop and suspend Methods Are Deprecated**

The initial release of Java defined a `stop` method that simply terminates a thread, and a `suspend` method that blocks a thread until another thread calls `resume`. The `stop` and `suspend` methods have something in common: Both attempt to control the behavior of a given thread without the thread's cooperation.

Both of these methods have been deprecated since Java SE 1.2. The `stop` method is inherently unsafe, and experience has shown that the `suspend` method frequently leads to deadlocks. In this section, you will see why these methods are problematic and what you can do to avoid problems.

Let us turn to the `stop` method first. This method terminates all pending methods, including the `run` method. When a thread is stopped, it immediately gives up the locks on all objects that it has locked. This can leave objects in an inconsistent state. For example, suppose a `TransferThread` is stopped in the middle of moving money from one account to another, after the withdrawal and before the deposit. Now the bank object is *damaged*. Since the lock has been relinquished, the damage is observable from the other threads that have not been stopped.

When a thread wants to stop another thread, it has no way of knowing when the `stop` method is safe and when it leads to damaged objects. Therefore, the method has been deprecated. You should interrupt a thread when you want it to stop. The interrupted thread can then stop when it is safe to do so.



**NOTE:** Some authors claim that the `stop` method has been deprecated because it can cause objects to be permanently locked by a stopped thread. However, that claim is not valid. A stopped thread exits all synchronized methods it has called—technically, by throwing a `ThreadDeath` exception. As a consequence, the thread relinquishes the intrinsic object locks that it holds.

Next, let us see what is wrong with the `suspend` method. Unlike `stop`, `suspend` won't damage objects. However, if you suspend a thread that owns a lock, then the lock is unavailable until the thread is resumed. If the thread that calls the `suspend` method tries to acquire the same lock, then the program deadlocks: The suspended thread waits to be resumed, and the suspending thread waits for the lock.

This situation occurs frequently in graphical user interfaces. Suppose we have a graphical simulation of our bank. A button labeled `Pause` suspends the transfer threads, and a button labeled `Resume` resumes them.

```
pauseButton.addActionListener(new
    ActionListener()
    {
        public void actionPerformed(ActionEvent event)
        {
            for (int i = 0; i < threads.length; i++)
                threads[i].suspend(); // Don't do this
        }
    });
resumeButton.addActionListener(. . .); // calls resume on all transfer threads
```

Suppose a `paintComponent` method paints a chart of each account, calling a `getBalances` method to get an array of balances.

As you will see in the section “Threads and Swing” on page 794, both the button actions and the repainting occur in the same thread, the *event dispatch thread*. Consider the following scenario:

1. One of the transfer threads acquires the lock of the bank object.
2. The user clicks the `Pause` button.
3. All transfer threads are suspended; one of them still holds the lock on the bank object.
4. For some reason, the account chart needs to be repainted.
5. The `paintComponent` method calls the `getBalances` method.
6. That method tries to acquire the lock of the bank object.

Now the program is frozen.

The event dispatch thread can't proceed because the lock is owned by one of the suspended threads. Thus, the user can't click the `Resume` button, and the threads won't ever resume.

If you want to safely suspend a thread, introduce a variable `suspendRequested` and test it in a safe place of your `run` method—somewhere your thread doesn't lock objects that other threads need. When your thread finds that the `suspendRequested` variable has been set, it should keep waiting until it becomes available again.

The following code framework implements that design:

```
public void run()
{
    while (. . .)
    {
        . . .
    }
}
```

```
        if (suspendRequested)
        {
            suspendLock.lock();
            try { while (suspendRequested) suspendCondition.await(); }
            finally { suspendLock.unlock(); }
        }
    }
}
public void requestSuspend() { suspendRequested = true; }
public void requestResume()
{
    suspendRequested = false;
    suspendLock.lock();
    try { suspendCondition.signalAll(); }
    finally { suspendLock.unlock(); }
}
private volatile boolean suspendRequested = false;
private Lock suspendLock = new ReentrantLock();
private Condition suspendCondition = suspendLock.newCondition();
```

### Blocking Queues

You have now seen the low-level building blocks that form the foundations of concurrent programming in Java. However, for practical programming, you want to stay away from the low-level constructs whenever possible. It is much easier and safer to use higher level structures that have been implemented by concurrency experts.

Many threading problems can be formulated elegantly and safely by using one or more queues. Producer threads insert items into the queue, and consumer threads retrieve them. The queue lets you safely hand over data from one thread to another. For example, consider our bank transfer program. Rather than accessing the bank object directly, the transfer threads insert transfer instruction objects into a queue. Another thread removes the instructions from the queue and carries out the transfers. Only that thread has access to the internals of the bank object. No synchronization is necessary. (Of course, the implementors of the threadsafe queue classes had to worry about locks and conditions, but that was their problem, not yours.)

A *blocking queue* causes a thread to block when you try to add an element when the queue is currently full or to remove an element when the queue is empty. Blocking queues are a useful tool for coordinating the work of multiple threads. Worker threads can periodically deposit intermediate results in a blocking queue. Other worker threads remove the intermediate results and modify them further. The queue automatically balances the workload. If the first set of threads runs slower than the second, the second set blocks while waiting for the results. If the first set of threads runs faster, the queue fills up until the second set catches up. Table 14–1 shows the methods for blocking queues.

**Table 14–1 Blocking Queue Methods**

Method	Normal Action	Action in Special Circumstances
<code>add</code>	Adds an element	Throws an <code>IllegalStateException</code> if the queue is full
<code>element</code>	Returns the head element	Throws a <code>NoSuchElementException</code> if the queue is empty
<code>offer</code>	Adds an element and returns <code>true</code>	Returns <code>false</code> if the queue is full
<code>peek</code>	Returns the head element	Returns <code>null</code> if the queue was empty
<code>poll</code>	Removes and returns the head element	Returns <code>null</code> if the queue was empty
<code>put</code>	Adds an element	Blocks if the queue is full
<code>remove</code>	Removes and returns the head element	Throws a <code>NoSuchElementException</code> if the queue is empty
<code>take</code>	Removes and returns the head element	Blocks if the queue is empty

The blocking queue methods fall into three categories, depending on their action when the queue is full or empty. If you use the queue as a thread management tool, you will want to use the `put` and `take` methods. The `add`, `remove`, and `element` operations throw an exception when you try to add to a full queue or get the head of an empty queue. Of course, in a multithreaded program, the queue might become full or empty at any time, so you will instead want to use the `offer`, `poll`, and `peek` methods. These methods simply return with a failure indicator instead of throwing an exception if they cannot carry out their tasks.



**NOTE:** The `poll` and `peek` methods return `null` to indicate failure. Therefore, it is illegal to insert `null` values into these queues.

There are also variants of the `offer` and `poll` methods with a timeout. For example, the call

```
boolean success = q.offer(x, 100, TimeUnit.MILLISECONDS);
```

tries for 100 milliseconds to insert an element to the tail of the queue. If it succeeds, it returns `true`; otherwise, it returns `false` when it times out. Similarly, the call

```
Object head = q.poll(100, TimeUnit.MILLISECONDS)
```

tries for 100 milliseconds to remove the head of the queue. If it succeeds, it returns the head; otherwise, it returns `null` when it times out.

The `put` method blocks if the queue is full, and the `take` method blocks if the queue is empty. These are the equivalents of `offer` and `poll` with no timeout.

The `java.util.concurrent` package supplies several variations of blocking queues. By default, the `LinkedBlockingQueue` has no upper bound on its capacity, but a maximum capacity can be optionally specified. The `LinkedBlockingDeque` is a double-ended version. The

`ArrayBlockingQueue` is constructed with a given capacity and an optional parameter to require fairness. If fairness is specified, then the longest-waiting threads are given preferential treatment. As always, fairness exacts a significant performance penalty, and you should only use it if your problem specifically requires it.

The `PriorityBlockingQueue` is a priority queue, not a first-in/first-out queue. Elements are removed in order of their priority. The queue has unbounded capacity, but retrieval will block if the queue is empty. (See Chapter 13 for more information on priority queues.)

Finally, a `DelayQueue` contains objects that implement the `Delayed` interface:

```
interface Delayed extends Comparable<Delayed>
{
    long getDelay(TimeUnit unit);
}
```

The `getDelay` method returns the remaining delay of the object. A negative value indicates that the delay has elapsed. Elements can only be removed from a `DelayQueue` if their delay has elapsed. You also need to implement the `compareTo` method. The `DelayQueue` uses that method to sort the entries.

The program in Listing 14–10 shows how to use a blocking queue to control a set of threads. The program searches through all files in a directory and its subdirectories, printing lines that contain a given keyword.

A producer thread enumerates all files in all subdirectories and places them in a blocking queue. This operation is fast, and the queue would quickly fill up with all files in the file system if it was not bounded.

We also start a large number of search threads. Each search thread takes a file from the queue, opens it, prints all lines containing the keyword, and then takes the next file. We use a trick to terminate the application when no further work is required. In order to signal completion, the enumeration thread places a dummy object into the queue. (This is similar to a dummy suitcase with a label “last bag” in a baggage claim belt.) When a search thread takes the dummy, it puts it back and terminates.

Note that no explicit thread synchronization is required. In this application, we use the queue data structure as a synchronization mechanism.

**Listing 14–10** `BlockingQueueTest.java`

```
1. import java.io.*;
2. import java.util.*;
3. import java.util.concurrent.*;
4.
5. /**
6.  * @version 1.0 2004-08-01
7.  * @author Cay Horstmann
8.  */
9. public class BlockingQueueTest
10. {
11.     public static void main(String[] args)
12.     {
```

**Listing 14–10** BlockingQueueTest.java (continued)

```
13. Scanner in = new Scanner(System.in);
14. System.out.print("Enter base directory (e.g. /usr/local/jdk1.6.0/src): ");
15. String directory = in.nextLine();
16. System.out.print("Enter keyword (e.g. volatile): ");
17. String keyword = in.nextLine();
18.
19. final int FILE_QUEUE_SIZE = 10;
20. final int SEARCH_THREADS = 100;
21.
22. BlockingQueue<File> queue = new ArrayBlockingQueue<File>(FILE_QUEUE_SIZE);
23.
24. FileEnumerationTask enumerator = new FileEnumerationTask(queue, new File(directory));
25. new Thread(enumerator).start();
26. for (int i = 1; i <= SEARCH_THREADS; i++)
27.     new Thread(new SearchTask(queue, keyword)).start();
28. }
29. }
30.
31. /**
32.  * This task enumerates all files in a directory and its subdirectories.
33.  */
34. class FileEnumerationTask implements Runnable
35. {
36.     /**
37.     * Constructs a FileEnumerationTask.
38.     * @param queue the blocking queue to which the enumerated files are added
39.     * @param startingDirectory the directory in which to start the enumeration
40.     */
41.     public FileEnumerationTask(BlockingQueue<File> queue, File startingDirectory)
42.     {
43.         this.queue = queue;
44.         this.startingDirectory = startingDirectory;
45.     }
46.
47.     public void run()
48.     {
49.         try
50.         {
51.             enumerate(startingDirectory);
52.             queue.put(DUMMY);
53.         }
54.         catch (InterruptedException e)
55.         {
56.         }
57.     }
58.
59.     /**
60.     * Recursively enumerates all files in a given directory and its subdirectories
61.     * @param directory the directory in which to start
62.     */
```

**Listing 14–10** BlockingQueueTest.java (continued)

```
63. public void enumerate(File directory) throws InterruptedException
64. {
65.     File[] files = directory.listFiles();
66.     for (File file : files)
67.     {
68.         if (file.isDirectory()) enumerate(file);
69.         else queue.put(file);
70.     }
71. }
72.
73. public static File DUMMY = new File("");
74.
75. private BlockingQueue<File> queue;
76. private File startingDirectory;
77. }
78.
79. /**
80.  * This task searches files for a given keyword.
81.  */
82. class SearchTask implements Runnable
83. {
84.     /**
85.      * Constructs a SearchTask.
86.      * @param queue the queue from which to take files
87.      * @param keyword the keyword to look for
88.      */
89.     public SearchTask(BlockingQueue<File> queue, String keyword)
90.     {
91.         this.queue = queue;
92.         this.keyword = keyword;
93.     }
94.
95.     public void run()
96.     {
97.         try
98.         {
99.             boolean done = false;
100.            while (!done)
101.            {
102.                File file = queue.take();
103.                if (file == FileEnumerationTask.DUMMY)
104.                {
105.                    queue.put(file);
106.                    done = true;
107.                }
108.                else search(file);
109.            }
110.        }
111.        catch (IOException e)
112.        {
```



**Listing 14–10** BlockingQueueTest.java (continued)

```
113.     e.printStackTrace();
114.     }
115.     catch (InterruptedException e)
116.     {
117.     }
118. }
119.
120. /**
121.  * Searches a file for a given keyword and prints all matching lines.
122.  * @param file the file to search
123.  */
124. public void search(File file) throws IOException
125. {
126.     Scanner in = new Scanner(new FileInputStream(file));
127.     int lineNumber = 0;
128.     while (in.hasNextLine())
129.     {
130.         lineNumber++;
131.         String line = in.nextLine();
132.         if (line.contains(keyword)) System.out.printf("%s:%d:%s%n", file.getPath(),
133.             lineNumber, line);
134.     }
135.     in.close();
136. }
137.
138. private BlockingQueue<File> queue;
139. private String keyword;
140. }
```

**API** `java.util.concurrent.ArrayBlockingQueue<E>` 5.0

- `ArrayBlockingQueue(int capacity)`
- `ArrayBlockingQueue(int capacity, boolean fair)`  
constructs a blocking queue with the given capacity and fairness settings. The queue is implemented as a circular array.

**API** `java.util.concurrent.LinkedBlockingQueue<E>` 5.0**API** `java.util.concurrent.LinkedBlockingDeque<E>` 6

- `LinkedBlockingQueue()`
- `LinkedBlockingDeque()`  
constructs an unbounded blocking queue or deque, implemented as a linked list.
- `LinkedBlockingQueue(int capacity)`
- `LinkedBlockingDeque(int capacity)`  
constructs a bounded blocking queue or deque with the given capacity, implemented as a linked list.

**API** `java.util.concurrent.DelayQueue<E extends Delayed>` 5.0

- `DelayQueue()`  
constructs an unbounded bounded blocking queue of `Delayed` elements. Only elements whose delay has expired can be removed from the queue.

**API** `java.util.concurrent.Delayed` 5.0

- `long getDelay(TimeUnit unit)`  
gets the delay for this object, measured in the given time unit.

**API** `java.util.concurrent.PriorityBlockingQueue<E>` 5.0

- `PriorityBlockingQueue()`
- `PriorityBlockingQueue(int initialCapacity)`
- `PriorityBlockingQueue(int initialCapacity, Comparator<? super E> comparator)`  
constructs an unbounded blocking priority queue implemented as a heap.

<i>Parameters</i>	<code>initialCapacity</code>	The initial capacity of the priority queue. Default is 11.
	<code>comparator</code>	The comparator used to compare elements. If not specified, the elements must implement the <code>Comparable</code> interface.

**API** `java.util.concurrent.BlockingQueue<E>` 5.0

- `void put(E element)`  
adds the element, blocking if necessary.
- `E take()`  
removes and returns the head element, blocking if necessary.
- `boolean offer(E element, long time, TimeUnit unit)`  
adds the given element and returns `true` if successful, blocking if necessary until the element has been added or the time has elapsed.
- `E poll(long time, TimeUnit unit)`  
removes and returns the head element, blocking if necessary until an element is available or the time has elapsed. Returns `null` upon failure.

**API** `java.util.concurrent.BlockingDeque<E>` 6

- `void putFirst(E element)`
- `void putLast(E element)`  
adds the element, blocking if necessary.
- `E takeFirst()`
- `E takeLast()`  
removes and returns the head or tail element, blocking if necessary.

- `boolean offerFirst(E element, long time, TimeUnit unit)`
- `boolean offerLast(E element, long time, TimeUnit unit)`  
adds the given element and returns true if successful, blocking if necessary until the element has been added or the time has elapsed.
- `E pollFirst(long time, TimeUnit unit)`
- `E pollLast(long time, TimeUnit unit)`  
removes and returns the head or tail element, blocking if necessary until an element is available or the time has elapsed. Returns `null` upon failure.

### Thread-Safe Collections

If multiple threads concurrently modify a data structure such as a hash table, then it is easily possible to damage the data structure. (See Chapter 13 for more information on hash tables.) For example, one thread may begin to insert a new element. Suppose it is preempted while it is in the middle of rerouting the links between the hash table's buckets. If another thread starts traversing the same list, it may follow invalid links and create havoc, perhaps throwing exceptions or being trapped in an infinite loop.

You can protect a shared data structure by supplying a lock, but it is usually easier to choose a thread-safe implementation instead. The blocking queues that we discussed in the preceding section are, of course, thread-safe collections. In the following sections, we discuss the other thread-safe collections that the Java library provides.

#### Efficient Maps, Sets, and Queues

The `java.util.concurrent` package supplies efficient implementations for maps, sorted sets, and queues: `ConcurrentHashMap`, `ConcurrentSkipListMap`, `ConcurrentSkipListSet`, and `ConcurrentLinkedQueue`.

These collections use sophisticated algorithms that minimize contention by allowing concurrent access to different parts of the data structure.

Unlike in most collections, the `size` method does not necessarily operate in constant time. Determining the current size of one of these collections usually requires traversal.

The collections return *weakly consistent* iterators. That means that the iterators may or may not reflect all modifications that are made after they were constructed, but they will not return a value twice and they will not throw a `ConcurrentModificationException`.



**NOTE:** In contrast, an iterator of a collection in the `java.util` package throws a `ConcurrentModificationException` when the collection has been modified after construction of the iterator.

The concurrent hash map can efficiently support a large number of readers and a fixed number of writers. By default, it is assumed that there are up to 16 *simultaneous* writer threads. There can be many more writer threads, but if more than 16 write at the same time, the others are temporarily blocked. You can specify a higher number in the constructor, but it is unlikely that you will need to.

The `ConcurrentHashMap` and `ConcurrentSkipListMap` classes have useful methods for atomic insertion and removal of associations. The `putIfAbsent` method atomically adds a new association provided there wasn't one before. This is useful for a cache that is accessed by multiple threads, to ensure that only one thread adds an item into the cache:

```
cache.putIfAbsent(key, value);
```

The opposite operation is `remove` (which perhaps should have been called `removeIfPresent`). The call

```
cache.remove(key, value)
```

atomically removes the key and value if they are present in the map. Finally,

```
cache.replace(key, oldValue, newValue)
```

atomically replaces the old value with the new one, provided the old value was associated with the given key.

**API** `java.util.concurrent.ConcurrentLinkedQueue<E>` 5.0

- `ConcurrentLinkedQueue<E>()`  
constructs an unbounded, nonblocking queue that can be safely accessed by multiple threads.

**API** `java.util.concurrent.ConcurrentSkipListSet<E>` 6

- `ConcurrentSkipListSet<E>()`
- `ConcurrentSkipListSet<E>(Comparator<? super E> comp)`  
constructs a sorted set that can be safely accessed by multiple threads. The first constructor requires that the elements implement the `Comparable` interface.

**API** `java.util.concurrent.ConcurrentHashMap<K, V>` 5.0

**API** `java.util.concurrent.ConcurrentSkipListMap<K, V>` 6

- `ConcurrentHashMap<K, V>()`
  - `ConcurrentHashMap<K, V>(int initialCapacity)`
  - `ConcurrentHashMap<K, V>(int initialCapacity, float loadFactor, int concurrencyLevel)`  
constructs a hash map that can be safely accessed by multiple threads.
- |                   |                               |   |
|-------------------|-------------------------------|---|
| <i>Parameters</i> | <code>initialCapacity</code>  | The initial capacity for this collection. Default is 16.  |
|                   | <code>loadFactor</code>       | Controls resizing: If the average load per bucket exceeds this factor, the table is resized. Default is 0.75. |
|                   | <code>concurrencyLevel</code> | The estimated number of concurrent writer threads.  |
- `ConcurrentSkipListMap<K, V>()`
  - `ConcurrentSkipListSet<K, V>(Comparator<? super K> comp)`  
constructs a sorted map that can be safely accessed by multiple threads. The first constructor requires that the keys implement the `Comparable` interface.
  - `V putIfAbsent(K key, V value)`  
if the key is not yet present in the map, associates the given value with the given key and returns `null`. Otherwise returns the existing value associated with the key.

- `boolean remove(K key, V value)`  
if the given key is currently associated with this value, removes the given key and value and returns true. Otherwise returns false.
- `boolean replace(K key, V oldValue, V newValue)`  
if the given key is currently associated with `oldValue`, associates it with `newValue`. Otherwise, returns false.

### Copy on Write Arrays

The `CopyOnWriteArrayList` and `CopyOnWriteArraySet` are thread-safe collections in which all mutators make a copy of the underlying array. This arrangement is useful if the number of threads that iterate over the collection greatly outnumbers the threads that mutate it. When you construct an iterator, it contains a reference to the current array. If the array is later mutated, the iterator still has the old array, but the collection's array is replaced. As a consequence, the older iterator has a consistent (but potentially outdated) view that it can access without any synchronization expense.

### Older Thread-Safe Collections

Ever since the initial release of Java, the `Vector` and `Hashtable` classes provided thread-safe implementations of a dynamic array and a hash table. In Java SE 1.2, these classes were declared obsolete and replaced by the `ArrayList` and `HashMap` classes. Those classes are not thread-safe. Instead, a different mechanism is supplied in the collections library. Any collection class can be made thread-safe by means of a *synchronization wrapper*:

```
List<E> synchArrayList = Collections.synchronizedList(new ArrayList<E>());
Map<K, V> synchHashMap = Collections.synchronizedMap(new HashMap<K, V>());
```

The methods of the resulting collections are protected by a lock, providing thread-safe access.

You should make sure that no thread accesses the data structure through the original unsynchronized methods. The easiest way to ensure this is not to save any reference to the original object. Simply construct a collection and immediately pass it to the wrapper, as we did in our examples.

You still need to use “client-side” locking if you want to *iterate* over the collection while another thread has the opportunity to mutate it:

```
synchronized (synchHashMap)
{
    Iterator<K> iter = synchHashMap.keySet().iterator();
    while (iter.hasNext()) . . . ;
}
```

You must use the same code if you use a “for each” loop because the loop uses an iterator. Note that the iterator actually fails with a `ConcurrentModificationException` if another thread mutates the collection while the iteration is in progress. The synchronization is still required so that the concurrent modification can be reliably detected.

You are usually better off using the collections defined in the `java.util.concurrent` package instead of the synchronization wrappers. In particular, the `ConcurrentHashMap` map has been carefully implemented so that multiple threads can access it without blocking each other, provided they access different buckets. One exception is an array list that is frequently mutated. In that case, a synchronized `ArrayList` can outperform a `CopyOnWriteArrayList`.

**API** java.util.Collections 1.2

- static <E> Collection<E> synchronizedCollection(Collection<E> c)
  - static <E> List synchronizedList(List<E> c)
  - static <E> Set synchronizedSet(Set<E> c)
  - static <E> SortedSet synchronizedSortedSet(SortedSet<E> c)
  - static <K, V> Map<K, V> synchronizedMap(Map<K, V> c)
  - static <K, V> SortedMap<K, V> synchronizedSortedMap(SortedMap<K, V> c)
- constructs views of the collection whose methods are synchronized.

**Callables and Futures**

A `Runnable` encapsulates a task that runs asynchronously; you can think of it as an asynchronous method with no parameters and no return value. A `Callable` is similar to a `Runnable`, but it returns a value. The `Callable` interface is a parameterized type, with a single method call.

```
public interface Callable<V>
{
    V call() throws Exception;
}
```

The type parameter is the type of the returned value. For example, a `Callable<Integer>` represents an asynchronous computation that eventually returns an `Integer` object.

A `Future` holds the *result* of an asynchronous computation. You can start a computation, give someone the `Future` object, and forget about it. The owner of the `Future` object can obtain the result when it is ready.

The `Future` interface has the following methods:

```
public interface Future<V>
{
    V get() throws . . .;
    V get(long timeout, TimeUnit unit) throws . . .;
    void cancel(boolean mayInterrupt);
    boolean isCancelled();
    boolean isDone();
}
```

A call to the first `get` method blocks until the computation is finished. The second method throws a `TimeoutException` if the call timed out before the computation finished. If the thread running the computation is interrupted, both methods throw an `InterruptedException`. If the computation has already finished, then `get` returns immediately.

The `isDone` method returns `false` if the computation is still in progress, `true` if it is finished.

You can cancel the computation with the `cancel` method. If the computation has not yet started, it is canceled and will never start. If the computation is currently in progress, then it is interrupted if the `mayInterrupt` parameter is `true`.

The `FutureTask` wrapper is a convenient mechanism for turning a `Callable` into both a `Future` and a `Runnable`—it implements both interfaces. For example:

```
Callable<Integer> myComputation = . . .;
FutureTask<Integer> task = new FutureTask<Integer>(myComputation);
Thread t = new Thread(task); // it's a Runnable
```

```
t.start();
. . .
Integer result = task.get(); // it's a Future
```

The program in Listing 14–11 puts these concepts to work. This program is similar to the preceding example that found files containing a given keyword. However, now we will merely count the number of matching files. Thus, we have a long-running task that yields an integer value—an example of a `Callable<Integer>`.

```
class MatchCounter implements Callable<Integer>
{
    public MatchCounter(File directory, String keyword) { . . . }
    public Integer call() { . . . } // returns the number of matching files
}
```

Then we construct a `FutureTask` object from the `MatchCounter` and use it to start a thread.

```
FutureTask<Integer> task = new FutureTask<Integer>(counter);
Thread t = new Thread(task);
t.start();
```

Finally, we print the result.

```
System.out.println(task.get() + " matching files.");
```

Of course, the call to `get` blocks until the result is actually available.

Inside the `call` method, we use the same mechanism recursively. For each subdirectory, we produce a new `MatchCounter` and launch a thread for it. We also stash the `FutureTask` objects away in an `ArrayList<Future<Integer>>`. At the end, we add up all results:

```
for (Future<Integer> result : results)
    count += result.get();
```

Each call to `get` blocks until the result is available. Of course, the threads run in parallel, so there is a good chance that the results will all be available at about the same time.

#### Listing 14–11 FutureTest.java

```
1. import java.io.*;
2. import java.util.*;
3. import java.util.concurrent.*;
4.
5. /**
6.  * @version 1.0 2004-08-01
7.  * @author Cay Horstmann
8.  */
9. public class FutureTest
10. {
11.     public static void main(String[] args)
12.     {
13.         Scanner in = new Scanner(System.in);
14.         System.out.print("Enter base directory (e.g. /usr/local/jdk5.0/src): ");
15.         String directory = in.nextLine();
16.         System.out.print("Enter keyword (e.g. volatile): ");
17.         String keyword = in.nextLine();
18.
```

**Listing 14–11** FutureTest.java (continued)

```
19.     MatchCounter counter = new MatchCounter(new File(directory), keyword);
20.     FutureTask<Integer> task = new FutureTask<Integer>(counter);
21.     Thread t = new Thread(task);
22.     t.start();
23.     try
24.     {
25.         System.out.println(task.get() + " matching files.");
26.     }
27.     catch (ExecutionException e)
28.     {
29.         e.printStackTrace();
30.     }
31.     catch (InterruptedException e)
32.     {
33.     }
34. }
35. }
36.
37. /**
38.  * This task counts the files in a directory and its subdirectories that contain a given keyword.
39.  */
40. class MatchCounter implements Callable<Integer>
41. {
42.     /**
43.     * Constructs a MatchCounter.
44.     * @param directory the directory in which to start the search
45.     * @param keyword the keyword to look for
46.     */
47.     public MatchCounter(File directory, String keyword)
48.     {
49.         this.directory = directory;
50.         this.keyword = keyword;
51.     }
52.
53.     public Integer call()
54.     {
55.         count = 0;
56.         try
57.         {
58.             File[] files = directory.listFiles();
59.             ArrayList<Future<Integer>> results = new ArrayList<Future<Integer>>();
60.
61.             for (File file : files)
62.                 if (file.isDirectory())
63.                 {
64.                     MatchCounter counter = new MatchCounter(file, keyword);
65.                     FutureTask<Integer> task = new FutureTask<Integer>(counter);
66.                     results.add(task);
67.                     Thread t = new Thread(task);
68.                     t.start();
```



**Listing 14–11** FutureTest.java (continued)

```
69.     }
70.     else
71.     {
72.         if (search(file)) count++;
73.     }
74.
75.     for (Future<Integer> result : results)
76.     try
77.     {
78.         count += result.get();
79.     }
80.     catch (ExecutionException e)
81.     {
82.         e.printStackTrace();
83.     }
84. }
85. catch (InterruptedException e)
86. {
87. }
88. return count;
89. }
90.
91. /**
92.  * Searches a file for a given keyword.
93.  * @param file the file to search
94.  * @return true if the keyword is contained in the file
95.  */
96. public boolean search(File file)
97. {
98.     try
99.     {
100.         Scanner in = new Scanner(new FileInputStream(file));
101.         boolean found = false;
102.         while (!found && in.hasNextLine())
103.         {
104.             String line = in.nextLine();
105.             if (line.contains(keyword)) found = true;
106.         }
107.         in.close();
108.         return found;
109.     }
110.     catch (IOException e)
111.     {
112.         return false;
113.     }
114. }
115.
116. private File directory;
117. private String keyword;
118. private int count;
119. }
```

**API** `java.util.concurrent.Callable<V>` 5.0

- `V call()`  
runs a task that yields a result.

**API** `java.util.concurrent.Future<V>` 5.0

- `V get()`
- `V get(long time, TimeUnit unit)`  
gets the result, blocking until it is available or the given time has elapsed. The second method throws a `TimeoutException` if it was unsuccessful.
- `boolean cancel(boolean mayInterrupt)`  
attempts to cancel the execution of this task. If the task has already started and the `mayInterrupt` parameter is `true`, it is interrupted. Returns `true` if the cancellation was successful.
- `boolean isCancelled()`  
returns `true` if the task was canceled before it completed.
- `boolean isDone()`  
returns `true` if the task completed, through normal completion, cancellation, or an exception.

**API** `java.util.concurrent.FutureTask<V>` 5.0

- `FutureTask(Callable<V> task)`
- `FutureTask(Runnable task, V result)`  
constructs an object that is both a `Future<V>` and a `Runnable`.

**Executors**

Constructing a new thread is somewhat expensive because it involves interaction with the operating system. If your program creates a large number of short-lived threads, then it should instead use a *thread pool*. A thread pool contains a number of idle threads that are ready to run. You give a `Runnable` to the pool, and one of the threads calls the `run` method. When the `run` method exits, the thread doesn't die but stays around to serve the next request.

Another reason to use a thread pool is to throttle the number of concurrent threads. Creating a huge number of threads can greatly degrade performance and even crash the virtual machine. If you have an algorithm that creates lots of threads, then you should use a "fixed" thread pool that bounds the total number of concurrent threads.

The `Executors` class has a number of static factory methods for constructing thread pools; see Table 14-2 for a summary.

**Table 14–2 Executors Factory Methods**

Method	Description
<code>newCachedThreadPool</code>	New threads are created as needed; idle threads are kept for 60 seconds.
<code>newFixedThreadPool</code>	The pool contains a fixed set of threads; idle threads are kept indefinitely.
<code>newSingleThreadExecutor</code>	A “pool” with a single thread that executes the submitted tasks sequentially (similar to the Swing event dispatch thread).
<code>newScheduledThreadPool</code>	A fixed-thread pool for scheduled execution; a replacement for <code>java.util.Timer</code> .
<code>newSingleThreadScheduledExecutor</code>	A single-thread “pool” for scheduled execution.

### Thread Pools

Let us look at the first three methods in Table 14–2. We discuss the remaining methods in the section “Scheduled Execution” on page 783. The `newCachedThreadPool` method constructs a thread pool that executes each task immediately, using an existing idle thread when available and creating a new thread otherwise. The `newFixedThreadPool` method constructs a thread pool with a fixed size. If more tasks are submitted than there are idle threads, then the unserved tasks are placed on a queue. They are run when other tasks have completed. The `newSingleThreadExecutor` is a degenerate pool of size 1: A single thread executes the submitted tasks, one after another. These three methods return an object of the `ThreadPoolExecutor` class that implements the `ExecutorService` interface.

You can submit a `Runnable` or `Callable` to an `ExecutorService` with one of the following methods:

```
Future<?> submit(Runnable task)
Future<T> submit(Runnable task, T result)
Future<T> submit(Callable<T> task)
```

The pool will run the submitted task at its earliest convenience. When you call `submit`, you get back a `Future` object that you can use to query the state of the task.

The first `submit` method returns an odd-looking `Future<?>`. You can use such an object to call `isDone`, `cancel`, or `isCancelled`. But the `get` method simply returns `null` upon completion.

The second version of `submit` also submits a `Runnable`, and the `get` method of the `Future` returns the given `result` object upon completion.

The third version submits a `Callable`, and the returned `Future` gets the result of the computation when it is ready.

When you are done with a thread pool, call `shutdown`. This method initiates the shutdown sequence for the pool. An executor that is shut down accepts no new tasks. When all tasks are finished, the threads in the pool die. Alternatively, you can call `shutdownNow`. The pool then cancels all tasks that have not yet begun and attempts to interrupt the running threads.

Here, in summary, is what you do to use a connection pool:

1. Call the static `newCachedThreadPool` or `newFixedThreadPool` method of the `Executors` class.
2. Call `submit` to submit `Runnable` or `Callable` objects.
3. If you want to be able to cancel a task or if you submit `Callable` objects, hang on to the returned `Future` objects.
4. Call `shutdown` when you no longer want to submit any tasks.

For example, the preceding example program produced a large number of short-lived threads, one per directory. The program in Listing 14–12 uses a thread pool to launch the tasks instead.

For informational purposes, this program prints out the largest pool size during execution. This information is not available through the `ExecutorService` interface. For that reason, we had to cast the pool object to the `ThreadPoolExecutor` class.

**Listing 14–12** ThreadPoolTest.java

```
1. import java.io.*;
2. import java.util.*;
3. import java.util.concurrent.*;
4.
5. /**
6.  * @version 1.0 2004-08-01
7.  * @author Cay Horstmann
8.  */
9. public class ThreadPoolTest
10. {
11.     public static void main(String[] args) throws Exception
12.     {
13.         Scanner in = new Scanner(System.in);
14.         System.out.print("Enter base directory (e.g. /usr/local/jdk5.0/src): ");
15.         String directory = in.nextLine();
16.         System.out.print("Enter keyword (e.g. volatile): ");
17.         String keyword = in.nextLine();
18.
19.         ExecutorService pool = Executors.newCachedThreadPool();
20.
21.         MatchCounter counter = new MatchCounter(new File(directory), keyword, pool);
22.         Future<Integer> result = pool.submit(counter);
23.
24.         try
25.         {
26.             System.out.println(result.get() + " matching files.");
27.         }
28.         catch (ExecutionException e)
29.         {
30.             e.printStackTrace();
31.         }
32.         catch (InterruptedException e)
33.         {
```

**Listing 14–12** ThreadPoolTest.java (continued)

```
34.     }
35.     pool.shutdown();
36.
37.     int largestPoolSize = ((ThreadPoolExecutor) pool).getLargestPoolSize();
38.     System.out.println("largest pool size=" + largestPoolSize);
39. }
40. }
41.
42. /**
43.  * This task counts the files in a directory and its subdirectories that contain a given keyword.
44.  */
45. class MatchCounter implements Callable<Integer>
46. {
47.     /**
48.      * Constructs a MatchCounter.
49.      * @param directory the directory in which to start the search
50.      * @param keyword the keyword to look for
51.      * @param pool the thread pool for submitting subtasks
52.      */
53.     public MatchCounter(File directory, String keyword, ExecutorService pool)
54.     {
55.         this.directory = directory;
56.         this.keyword = keyword;
57.         this.pool = pool;
58.     }
59.
60.     public Integer call()
61.     {
62.         count = 0;
63.         try
64.         {
65.             File[] files = directory.listFiles();
66.             ArrayList<Future<Integer>> results = new ArrayList<Future<Integer>>();
67.
68.             for (File file : files)
69.                 if (file.isDirectory())
70.                 {
71.                     MatchCounter counter = new MatchCounter(file, keyword, pool);
72.                     Future<Integer> result = pool.submit(counter);
73.                     results.add(result);
74.                 }
75.                 else
76.                 {
77.                     if (search(file)) count++;
78.                 }
79.
80.             for (Future<Integer> result : results)
81.                 try
82.                 {
```

**Listing 14–12** ThreadPoolTest.java (continued)

```
83.         count += result.get();
84.     }
85.     catch (ExecutionException e)
86.     {
87.         e.printStackTrace();
88.     }
89. }
90. catch (InterruptedException e)
91. {
92. }
93. return count;
94. }
95.
96. /**
97.  * Searches a file for a given keyword.
98.  * @param file the file to search
99.  * @return true if the keyword is contained in the file
100. */
101. public boolean search(File file)
102. {
103.     try
104.     {
105.         Scanner in = new Scanner(new FileInputStream(file));
106.         boolean found = false;
107.         while (!found && in.hasNextLine())
108.         {
109.             String line = in.nextLine();
110.             if (line.contains(keyword)) found = true;
111.         }
112.         in.close();
113.         return found;
114.     }
115.     catch (IOException e)
116.     {
117.         return false;
118.     }
119. }
120.
121. private File directory;
122. private String keyword;
123. private ExecutorService pool;
124. private int count;
125. }
```

---

**API** `java.util.concurrent.Executors` 5.0

- `ExecutorService newCachedThreadPool()`  
returns a cached thread pool that creates threads as needed and terminates threads that have been idle for 60 seconds.
- `ExecutorService newFixedThreadPool(int threads)`  
returns a thread pool that uses the given number of threads to execute tasks.
- `ExecutorService newSingleThreadExecutor()`  
returns an executor that executes tasks sequentially in a single thread.

**API** `java.util.concurrent.ExecutorService` 5.0

- `Future<T> submit(Callable<T> task)`
- `Future<T> submit(Runnable task, T result)`
- `Future<?> submit(Runnable task)`  
submits the given task for execution.
- `void shutdown()`  
shuts down the service, completing the already submitted tasks but not accepting new submissions.

**API** `java.util.concurrent.ThreadPoolExecutor` 5.0

- `int getLargestPoolSize()`  
returns the largest size of the thread pool during the life of this executor.

**Scheduled Execution**

The `ScheduledExecutorService` interface has methods for scheduled or repeated execution of tasks. It is a generalization of `java.util.Timer` that allows for thread pooling. The `newScheduledThreadPool` and `newSingleThreadScheduledExecutor` methods of the `Executors` class return objects that implement the `ScheduledExecutorService` interface.

You can schedule a `Runnable` or `Callable` to run once, after an initial delay. You can also schedule a `Runnable` to run periodically. See the API notes for details.

**API** `java.util.concurrent.Executors` 5.0

- `ScheduledExecutorService newScheduledThreadPool(int threads)`  
returns a thread pool that uses the given number of threads to schedule tasks.
- `ScheduledExecutorService newSingleThreadScheduledExecutor()`  
returns an executor that schedules tasks in a single thread.

**API** `java.util.concurrent.ScheduledExecutorService` 5.0

- `ScheduledFuture<V> schedule(Callable<V> task, long time, TimeUnit unit)`
- `ScheduledFuture<?> schedule(Runnable task, long time, TimeUnit unit)`  
schedules the given task after the given time has elapsed.

- `ScheduledFuture<?> scheduleAtFixedRate(Runnable task, long initialDelay, long period, TimeUnit unit)`  
schedules the given task to run periodically, every `period` units, after the initial delay has elapsed.
- `ScheduledFuture<?> scheduleWithFixedDelay(Runnable task, long initialDelay, long delay, TimeUnit unit)`  
schedules the given task to run periodically, with `delay` units between completion of one invocation and the start of the next, after the initial delay has elapsed.

### Controlling Groups of Tasks

You have seen how to use an executor service as a thread pool to increase the efficiency of task execution. Sometimes, an executor is used for a more tactical reason, simply to control a group of related tasks. For example, you can cancel all tasks in an executor with the `shutdownNow` method.

The `invokeAny` method submits all objects in a collection of `Callable` objects and returns the result of a completed task. You don't know which task that is—presumably, it was the one that finished most quickly. You would use this method for a search problem in which you are willing to accept any solution. For example, suppose that you need to factor a large integer—a computation that is required for breaking the RSA cipher. You could submit a number of tasks, each of which attempts a factorization by using numbers in a different range. As soon as one of these tasks has an answer, your computation can stop.

The `invokeAll` method submits all objects in a collection of `Callable` objects and returns a list of `Future` objects that represent the solutions to all tasks. You can process the results of the computation when they are available, like this:

```
List<Callable<T>> tasks = . . . ;
List<Future<T>> results = executor.invokeAll(tasks);
for (Future<T> result : results)
    processFurther(result.get());
```

A disadvantage with this approach is that you may wait needlessly if the first task happens to take a long time. It would make more sense to obtain the results in the order in which they are available. This can be arranged with the `ExecutorCompletionService`.

Start with an executor, obtained in the usual way. Then construct an `ExecutorCompletionService`. Submit tasks to the completion service. The service manages a blocking queue of `Future` objects, containing the results of the submitted tasks as they become available. Thus, a more efficient organization for the preceding computation is the following:

```
ExecutorCompletionService service = new ExecutorCompletionService(executor);
for (Callable<T> task : tasks) service.submit(task);
for (int i = 0; i < tasks.size(); i++)
    processFurther(service.take().get());
```

#### API `java.util.concurrent.ExecutorService 5.0`

- `T invokeAny(Collection<Callable<T>> tasks)`
- `T invokeAny(Collection<Callable<T>> tasks, long timeout, TimeUnit unit)`  
executes the given tasks and returns the result of one of them. The second method throws a `TimeoutException` if a timeout occurs.



- `List<Future<T>> invokeAll(Collection<Callable<T>> tasks)`
- `List<Future<T>> invokeAll(Collection<Callable<T>> tasks, long timeout, TimeUnit unit)` executes the given tasks and returns the results of all of them. The second method throws a `TimeoutException` if a timeout occurs.

**API** `java.util.concurrent.ExecutorCompletionService 5.0`

- `ExecutorCompletionService(Executor e)` constructs an executor completion service that collects the results of the given executor.
- `Future<T> submit(Callable<T> task)`
- `Future<T> submit(Runnable task, T result)` submits a task to the underlying executor.
- `Future<T> take()` removes the next completed result, blocking if no completed results are available.
- `Future<T> poll()`
- `Future<T> poll(long time, TimeUnit unit)` removes the next completed result or `null` if no completed results are available. The second method waits for the given time.

## Synchronizers

The `java.util.concurrent` package contains several classes that help manage a set of collaborating threads—see Table 14–3. These mechanisms have “canned functionality” for common rendezvous patterns between threads. If you have a set of collaborating threads that follows one of these behavior patterns, you should simply reuse the appropriate library class instead of trying to come up with a handcrafted collection of locks and conditions.

**Table 14–3 Synchronizers**

Class	What It Does	When To Use
<code>CyclicBarrier</code>	Allows a set of threads to wait until a predefined count of them has reached a common barrier, and then optionally executes a barrier action.	When a number of threads need to complete before their results can be used.
<code>CountDownLatch</code>	Allows a set of threads to wait until a count has been decremented to 0.	When one or more threads need to wait until a specified number of events have occurred.
<code>Exchanger</code>	Allows two threads to exchange objects when both are ready for the exchange.	When two threads work on two instances of the same data structure, one by filling an instance and the other by emptying the other.

**Table 14–3 Synchronizers (continued)**

Class	What It Does	When To Use
Semaphore	Allows a set of threads to wait until permits are available for proceeding.	To restrict the total number of threads that can access a resource. If permit count is one, use to block threads until another thread gives permission.
SynchronousQueue	Allows a thread to hand off an object to another thread.	To send an object from one thread to another when both are ready, without explicit synchronization.

**Semaphores**

Conceptually, a semaphore manages a number of *permits*. To proceed past the semaphore, a thread requests a permit by calling `acquire`. Only a fixed number of permits are available, limiting the number of threads that are allowed to pass. Other threads may issue permits by calling `release`. There are no actual permit objects. The semaphore simply keeps a count. Moreover, a permit doesn't have to be released by the thread that acquires it. In fact, any thread can issue any number of permits. If it issues more than the maximum available, the semaphore is simply set to the maximum count. This generality makes semaphores both very flexible and potentially confusing.

Semaphores were invented by Edsger Dijkstra in 1968, for use as a *synchronization primitive*. Dijkstra showed that semaphores can be efficiently implemented and that they are powerful enough to solve many common thread synchronization problems. In just about any operating systems textbook, you will find implementations of bounded queues using semaphores. Of course, application programmers shouldn't reinvent bounded queues. We suggest that you only use semaphores when their behavior maps well onto your synchronization problem, without your going through mental contortions.

One simple example is a semaphore with a permit count of 1. Such a semaphore can be used as a gate for one thread that is opened and closed by another thread. In the section "Example: Pausing and Resuming an Animation" on page 788, you will see an example in which a worker thread produces an animation. Occasionally, the worker thread waits for the user to press a button. The worker thread tries to acquire a permit, and it has to wait until the button click causes a permit to be issued.

**Countdown Latches**

A `CountDownLatch` lets a set of threads wait until a count has reached zero. The countdown latch is one-time only. Once the count has reached 0, you cannot increment it again.

A useful special case is a latch with a count of 1. This implements a one-time gate. Threads are held at the gate until another thread sets the count to 0.

Imagine, for example, a set of threads that need some initial data to do their work. The worker threads are started and wait at the gate. Another thread prepares the data. When it is ready, it calls `countDown`, and all worker threads proceed.

You can then use a second latch to check when all worker threads are done. Initialize the latch with the number of threads. Each worker thread counts down that latch just before

it terminates. Another thread that harvests the work results waits on the latch, and it proceeds as soon as all workers have terminated.

### **Barriers**

The `CyclicBarrier` class implements a rendezvous called a *barrier*. Consider a number of threads that are working on parts of a computation. When all parts are ready, the results need to be combined. When a thread is done with its part, we let it run against the barrier. Once all threads have reached the barrier, the barrier gives way and the threads can proceed.

Here are the details. First, construct a barrier, giving the number of participating threads:

```
CyclicBarrier barrier = new CyclicBarrier(nthreads);
```

Each thread does some work and calls `await` on the barrier upon completion:

```
public void run()
{
    doWork();
    barrier.await();
    . . .
}
```

The `await` method takes an optional timeout parameter:

```
barrier.await(100, TimeUnit.MILLISECONDS);
```

If any of the threads waiting for the barrier leaves the barrier, then the barrier *breaks*. (A thread can leave because it called `await` with a timeout or because it was interrupted.) In that case, the `await` method for all other threads throws a `BrokenBarrierException`. Threads that are already waiting have their `await` call terminated immediately.

You can supply an optional *barrier action* that is executed when all threads have reached the barrier:

```
Runnable barrierAction = . . .;
CyclicBarrier barrier = new CyclicBarrier(nthreads, barrierAction);
```

The action can harvest the result of the individual threads.

The barrier is called *cyclic* because it can be reused after all waiting threads have been released. In this regard, it differs from a `CountDownLatch`, which can only be used once.

### **Exchangers**

An `Exchanger` is used when two threads are working on two instances of the same data buffer. Typically, one thread fills the buffer, and the other consumes its contents. When both are done, they exchange their buffers.

### **Synchronous Queues**

A synchronous queue is a mechanism that pairs up producer and consumer threads. When a thread calls `put` on a `SynchronousQueue`, it blocks until another thread calls `take`, and vice versa. Unlike the case with an `Exchanger`, data are only transferred in one direction, from the producer to the consumer.

Even though the `SynchronousQueue` class implements the `BlockingQueue` interface, it is not conceptually a queue. It does not contain any elements—its `size` method always returns 0.

**Example: Pausing and Resuming an Animation**

Consider a program that does some work, updates the screen display, then waits for the user to look at the result and press a button to continue, and then does the next unit of work.

A semaphore with a permit count of 1 can be used to synchronize the worker thread and the event dispatch thread. The worker thread calls `acquire` whenever it is ready to pause. The GUI thread calls `release` whenever the user clicks the Continue button.

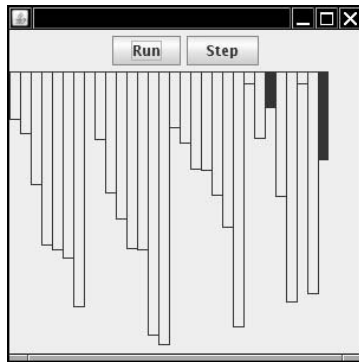
What happens if the user clicks the button multiple times while the worker thread is ready? Because only one permit is available, the permit count stays at 1.

The program in Listing 14–13 puts this idea to work. The program animates a sorting algorithm. A worker thread sorts an array, stopping periodically and waiting for the user to give permission to proceed. The user can admire a painting of the current state of the algorithm and press the Continue button to allow the worker thread to go to the next step.

We didn't want to bore you with the code for a sorting algorithm, so we simply call `Arrays.sort`, which implements the merge sort algorithm. To pause the algorithm, we supply a `Comparator` object that waits for the semaphore. Thus, the animation is paused whenever the algorithm compares two elements. We paint the current values of the array and highlight the elements that are being compared (see Figure 14–8).



**NOTE:** The animation shows the merging of smaller sorted ranges into larger ones, but it is not entirely accurate. The mergesort algorithm uses a second array for holding temporary values that we do not get to see. The point of this example is not to delve into sorting algorithms, but to show how to use a semaphore for pausing a worker thread.



**Figure 14–8** Animating a sort algorithm

**Listing 14-13** AlgorithmAnimation.java

```
1. import java.awt.*;
2. import java.awt.geom.*;
3. import java.awt.event.*;
4. import java.util.*;
5. import java.util.concurrent.*;
6. import javax.swing.*;
7.
8. /**
9.  * This program animates a sort algorithm.
10.  * @version 1.01 2007-05-18
11.  * @author Cay Horstmann
12.  */
13. public class AlgorithmAnimation
14. {
15.     public static void main(String[] args)
16.     {
17.         EventQueue.invokeLater(new Runnable()
18.         {
19.             public void run()
20.             {
21.                 JFrame frame = new AnimationFrame();
22.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23.                 frame.setVisible(true);
24.             }
25.         });
26.     }
27. }
28.
29. /**
30.  * This frame shows the array as it is sorted, together with buttons to single-step the
31.  * animation or to run it without interruption.
32.  */
33. class AnimationFrame extends JFrame
34. {
35.     public AnimationFrame()
36.     {
37.         ArrayComponent comp = new ArrayComponent();
38.         add(comp, BorderLayout.CENTER);
39.
40.         final Sorter sorter = new Sorter(comp);
41.
42.         JButton runButton = new JButton("Run");
43.         runButton.addActionListener(new ActionListener()
44.         {
45.             public void actionPerformed(ActionEvent event)
46.             {
47.                 sorter.setRun();
48.             }
49.         });
50.     }
```

**Listing 14-13** AlgorithmAnimation.java (continued)

```
51.     JButton stepButton = new JButton("Step");
52.     stepButton.addActionListener(new ActionListener()
53.     {
54.         public void actionPerformed(ActionEvent event)
55.         {
56.             sorter.setStep();
57.         }
58.     });
59.
60.     JPanel buttons = new JPanel();
61.     buttons.add(runButton);
62.     buttons.add(stepButton);
63.     add(buttons, BorderLayout.NORTH);
64.     setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
65.
66.     Thread t = new Thread(sorter);
67.     t.start();
68. }
69.
70. private static final int DEFAULT_WIDTH = 300;
71. private static final int DEFAULT_HEIGHT = 300;
72. }
73.
74. /**
75.  * This runnable executes a sort algorithm. When two elements are compared, the algorithm
76.  * pauses and updates a component.
77.  */
78. class Sorter implements Runnable
79. {
80.     /**
81.     * Constructs a Sorter.
82.     * @param values the array to be sorted
83.     * @param comp the component on which to display the sorting progress
84.     */
85.     public Sorter(ArrayComponent comp)
86.     {
87.         values = new Double[VALUES_LENGTH];
88.         for (int i = 0; i < values.length; i++)
89.             values[i] = new Double(Math.random());
90.         this.component = comp;
91.         this.gate = new Semaphore(1);
92.         this.run = false;
93.     }
94.
95.     /**
96.     * Sets the sorter to "run" mode. Called on the event dispatch thread.
97.     */
98.     public void setRun()
99.     {
```

**Listing 14-13** AlgorithmAnimation.java (continued)

```
100.     run = true;
101.     gate.release();
102. }
103.
104. /**
105.  * Sets the sorter to "step" mode. Called on the event dispatch thread.
106.  */
107. public void setStep()
108. {
109.     run = false;
110.     gate.release();
111. }
112.
113. public void run()
114. {
115.     Comparator<Double> comp = new Comparator<Double>()
116.     {
117.         public int compare(Double i1, Double i2)
118.         {
119.             component.setValues(values, i1, i2);
120.             try
121.             {
122.                 if (run) Thread.sleep(DELAY);
123.                 else gate.acquire();
124.             }
125.             catch (InterruptedException exception)
126.             {
127.                 Thread.currentThread().interrupt();
128.             }
129.             return i1.compareTo(i2);
130.         }
131.     };
132.     Arrays.sort(values, comp);
133.     component.setValues(values, null, null);
134. }
135.
136. private Double[] values;
137. private ArrayComponent component;
138. private Semaphore gate;
139. private static final int DELAY = 100;
140. private volatile boolean run;
141. private static final int VALUES_LENGTH = 30;
142. }
143.
144. /**
145.  * This component draws an array and marks two elements in the array.
146.  */
147. class ArrayComponent extends JComponent
148. {
```

**Listing 14–13** AlgorithmAnimation.java (continued)

```

149.  /**
150.   * Sets the values to be painted. Called on the sorter thread.
151.   * @param values the array of values to display
152.   * @param marked1 the first marked element
153.   * @param marked2 the second marked element
154.   */
155.  public synchronized void setValues(Double[] values, Double marked1, Double marked2)
156.  {
157.      this.values = values.clone();
158.      this.marked1 = marked1;
159.      this.marked2 = marked2;
160.      repaint();
161.  }
162.
163.  public synchronized void paintComponent(Graphics g) // Called on the event dispatch thread
164.  {
165.      if (values == null) return;
166.      Graphics2D g2 = (Graphics2D) g;
167.      int width = getWidth() / values.length;
168.      for (int i = 0; i < values.length; i++)
169.      {
170.          double height = values[i] * getHeight();
171.          Rectangle2D bar = new Rectangle2D.Double(width * i, 0, width, height);
172.          if (values[i] == marked1 || values[i] == marked2) g2.fill(bar);
173.          else g2.draw(bar);
174.      }
175.  }
176.
177.  private Double marked1;
178.  private Double marked2;
179.  private Double[] values;
180. }

```

**API** java.util.concurrent.CyclicBarrier 5.0

- CyclicBarrier(int parties)
- CyclicBarrier(int parties, Runnable barrierAction)  
constructs a cyclic barrier for the given number of parties. The barrierAction is executed when all parties have called await on the barrier.
- int await()
- int await(long time, TimeUnit unit)  
waits until all parties have called await on the barrier or until the timeout has been reached, in which case a TimeoutException is thrown. Upon success, returns the arrival index of this party. The first party has index parties - 1, and the last party has index 0.



**API** `java.util.concurrent.CountDownLatch` 5.0

- `CountDownLatch(int count)`  
constructs a countdown latch with the given count.
- `void await()`  
waits for this latch to count down to 0.
- `boolean await(long time, TimeUnit unit)`  
waits for this latch to count down to 0 or for the timeout to elapse. Returns `true` if the count is 0, `false` if the timeout elapsed.
- `public void countDown()`  
counts down the counter of this latch.

**API** `java.util.concurrent.Exchanger<V>` 5.0

- `V exchange(V item)`
- `V exchange(V item, long time, TimeUnit unit)`  
blocks until another thread calls this method, and then exchanges the item with the other thread and returns the other thread's item. The second method throws a `TimeoutException` after the timeout has elapsed.

**API** `java.util.concurrent.SynchronousQueue<V>` 5.0

- `SynchronousQueue()`
- `SynchronousQueue(boolean fair)`  
constructs a synchronous queue that allows threads to hand off items. If `fair` is `true`, the queue favors the longest-waiting threads.
- `void put(V item)`  
blocks until another thread calls `take` to take this item.
- `V take()`  
blocks until another thread calls `put`. Returns the item that the other thread provided.

**API** `java.util.concurrent.Semaphore` 5.0

- `Semaphore(int permits)`
- `Semaphore(int permits, boolean fair)`  
constructs a semaphore with the given maximum number of permits. If `fair` is `true`, the queue favors the longest-waiting threads.
- `void acquire()`  
waits to acquire a permit.
- `boolean tryAcquire()`  
tries to acquire a permit; returns `false` if none is available.
- `boolean tryAcquire(long time, TimeUnit unit)`  
tries to acquire a permit within the given time; returns `false` if none is available.
- `void release()`  
releases a permit.

## Threads and Swing

As we mentioned in the introduction to this chapter, one of the reasons to use threads in your programs is to make your programs more responsive. When your program needs to do something time consuming, then you should fire up another worker thread instead of blocking the user interface.

However, you have to be careful what you do in a worker thread because, perhaps surprisingly, Swing is *not thread safe*. If you try to manipulate user interface elements from multiple threads, then your user interface can become corrupted.

To see the problem, run the upcoming test program in Listing 14–14. When you click the Bad button, a new thread is started whose `run` method tortures a combo box, randomly adding and removing values.

```
public void run()
{
    try
    {
        while (true)
        {
            int i = Math.abs(generator.nextInt());
            if (i % 2 == 0)
                combo.insertItemAt(new Integer(i), 0);
            else if (combo.getItemCount() > 0)
                combo.removeItemAt(i % combo.getItemCount());
            sleep(1);
        }
        catch (InterruptedException e) {}
    }
}
```

Try it out. Click the Bad button. Click the combo box a few times. Move the scrollbar. Move the window. Click the Bad button again. Keep clicking the combo box. Eventually, you should see an exception report (see Figure 14–9).

What is going on? When an element is inserted into the combo box, the combo box fires an event to update the display. Then, the display code springs into action, reading the current size of the combo box and preparing to display the values. But the worker thread keeps going—occasionally resulting in a reduction of the count of the values in the combo box. The display code then thinks that there are more values in the model than there actually are, asks for nonexistent values, and triggers an `ArrayIndexOutOfBoundsException` exception.

This situation could have been avoided by enabling programmers to lock the combo box object while displaying it. However, the designers of Swing decided not to expend any effort to make Swing thread safe, for two reasons. First, synchronization takes time, and nobody wanted to slow down Swing any further. More important, the Swing team checked out the experience other teams had with thread-safe user interface toolkits. What they found was not encouraging. Programmers using thread-safe toolkits turned out to be confused by the demands for synchronization and often created deadlock-prone programs.

```

Terminal
File Edit View Terminal Tabs Help
559)
    at javax.swing.JComponent.getPreferredSize(JComponent.java:1627)
    at javax.swing.ScrollPaneLayout.layoutContainer(ScrollPaneLayout.java:76
9)
    at java.awt.Container.layout(Container.java:1432)
    at java.awt.Container.layout(Container.java:1421)
    at java.awt.Container.layout(Container.java:1519)
    at java.awt.Container.layout(Container.java:1491)
    at javax.swing.RepaintManager.validateInvalidComponents(RepaintManager.j
ava:635)
    at javax.swing.SystemEventQueueUtilities$ComponentWorkRequest.run(System
EventQueueUtilities.java:127)
    at java.awt.event.InvocationEvent.dispatch(InvocationEvent.java:209)
    at java.awt.EventQueue.dispatchEvent(EventQueue.java:597)
    at java.awt.EventDispatchThread.pumpOneEventForFilters(EventDispatchThre
ad.java:273)
    at java.awt.EventDispatchThread.pumpEventsForFilter(EventDispatchThread.
java:183)
    at java.awt.EventDispatchThread.pumpEventsForHierarchy(EventDispatchThre
ad.java:173)
    at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:168)
    at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:160)
    at java.awt.EventDispatchThread.run(EventDispatchThread.java:121)

```

**Figure 14-9** Exception reports in the console

### Running Time-Consuming Tasks

When you use threads together with Swing, you have to follow two simple rules.

- If an action takes a long time, do it in a separate worker thread and never in the event dispatch thread.
- Do not touch Swing components in any thread other than the event dispatch thread.

The reason for the first rule is easy to understand. If you take a long time in the event dispatch thread, the application seems “dead” because it cannot respond to any events. In particular, the event dispatch thread should never make input/output calls, which might block indefinitely, and it should never call `sleep`. (If you need to wait for a specific amount of time, use timer events.)

The second rule is often called the *single-thread rule* for Swing programming. We discuss it further on page 806.

These two rules seem to be in conflict with each other. Suppose you fire up a separate thread to run a time-consuming task. You usually want to update the user interface to indicate progress while your thread is working. When your task is finished, you want to update the GUI again. But you can’t touch Swing components from your thread. For example, if you want to update a progress bar or a label text, then you can’t simply set its value from your thread.

To solve this problem, you can use two utility methods in any thread to add arbitrary actions to the event queue. For example, suppose you want to periodically update a label in a thread to indicate progress. You can’t call `label.setText` from your thread.

Instead, use the `invokeLater` and `invokeAndWait` methods of the `EventQueue` class to have that call executed in the event dispatching thread.

Here is what you do. You place the Swing code into the `run` method of a class that implements the `Runnable` interface. Then, you create an object of that class and pass it to the static `invokeLater` or `invokeAndWait` method. For example, here is how to update a label text:

```
EventQueue.invokeLater(new
    Runnable()
    {
        public void run()
        {
            label.setText(percentage + "% complete");
        }
    });
```

The `invokeLater` method returns immediately when the event is posted to the event queue. The `run` method is executed asynchronously. The `invokeAndWait` method waits until the `run` method has actually been executed.

In the situation of updating a progress label, the `invokeLater` method is more appropriate. Users would rather have the worker thread make more progress than have the most precise progress indicator.

Both methods execute the `run` method in the event dispatch thread. No new thread is created.

Listing 14-14 demonstrates how to use the `invokeLater` method to safely modify the contents of a combo box. If you click on the Good button, a thread inserts and removes numbers. However, the actual modification takes place in the event dispatching thread.

**Listing 14-14** `SwingThreadTest.java`

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.util.*;
4. import javax.swing.*;
5.
6. /**
7.  * This program demonstrates that a thread that runs in parallel with the event dispatch thread
8.  * can cause errors in Swing components.
9.  * @version 1.23 2007-05-17
10.  * @author Cay Horstmann
11.  */
12. public class SwingThreadTest
13. {
14.     public static void main(String[] args)
15.     {
16.         EventQueue.invokeLater(new Runnable()
17.         {
18.             public void run()
19.             {
```

**Listing 14-14** SwingThreadTest.java (continued)

```
20.         SwingThreadFrame frame = new SwingThreadFrame();
21.         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22.         frame.setVisible(true);
23.     }
24. });
25. }
26. }
27.
28. /**
29.  * This frame has two buttons to fill a combo box from a separate thread. The "Good" button
30.  * uses the event queue; the "Bad" button modifies the combo box directly.
31.  */
32. class SwingThreadFrame extends JFrame
33. {
34.     public SwingThreadFrame()
35.     {
36.         setTitle("SwingThreadTest");
37.
38.         final JComboBox combo = new JComboBox();
39.         combo.insertItemAt(Integer.MAX_VALUE, 0);
40.         combo.setPrototypeDisplayValue(combo.getItemAt(0));
41.         combo.setSelectedIndex(0);
42.
43.         JPanel panel = new JPanel();
44.
45.         JButton goodButton = new JButton("Good");
46.         goodButton.addActionListener(new ActionListener()
47.         {
48.             public void actionPerformed(ActionEvent event)
49.             {
50.                 new Thread(new GoodWorkerRunnable(combo)).start();
51.             }
52.         });
53.         panel.add(goodButton);
54.         JButton badButton = new JButton("Bad");
55.         badButton.addActionListener(new ActionListener()
56.         {
57.             public void actionPerformed(ActionEvent event)
58.             {
59.                 new Thread(new BadWorkerRunnable(combo)).start();
60.             }
61.         });
62.         panel.add(badButton);
63.
64.         panel.add(combo);
65.         add(panel);
66.         pack();
67.     }
68. }
69.
```

**Listing 14-14** SwingThreadTest.java (continued)

```
70. /**
71. * This runnable modifies a combo box by randomly adding and removing numbers. This can result
72. * in errors because the combo box methods are not synchronized and both the worker thread
73. * and the event dispatch thread access the combo box.
74. */
75. class BadWorkerRunnable implements Runnable
76. {
77.     public BadWorkerRunnable(JComboBox aCombo)
78.     {
79.         combo = aCombo;
80.         generator = new Random();
81.     }
82.
83.     public void run()
84.     {
85.         try
86.         {
87.             while (true)
88.             {
89.                 int i = Math.abs(generator.nextInt());
90.                 if (i % 2 == 0) combo.insertItemAt(i, 0);
91.                 else if (combo.getItemCount() > 0) combo.removeItemAt(i % combo.getItemCount());
92.                 Thread.sleep(1);
93.             }
94.         }
95.         catch (InterruptedException e)
96.         {
97.         }
98.     }
99.
100.     private JComboBox combo;
101.     private Random generator;
102. }
103.
104. /**
105. * This runnable modifies a combo box by randomly adding and removing numbers. In order to
106. * ensure that the combo box is not corrupted, the editing operations are forwarded to the
107. * event dispatch thread.
108. */
109. class GoodWorkerRunnable implements Runnable
110. {
111.     public GoodWorkerRunnable(JComboBox aCombo)
112.     {
113.         combo = aCombo;
114.         generator = new Random();
115.     }
116.
117.     public void run()
118.     {
```

**Listing 14-14** SwingThreadTest.java (continued)

```
119.     try
120.     {
121.         while (true)
122.         {
123.             EventQueue.invokeLater(new Runnable()
124.             {
125.                 public void run()
126.                 {
127.                     int i = Math.abs(generator.nextInt());
128.                     if (i % 2 == 0) combo.insertItemAt(i, 0);
129.                     else if (combo.getItemCount() > 0) combo.removeItemAt(i
130.                         % combo.getItemCount());
131.                 }
132.             });
133.             Thread.sleep(1);
134.         }
135.     }
136.     catch (InterruptedException e)
137.     {
138.     }
139. }
140.
141. private JComboBox combo;
142. private Random generator;
143. }
```

**API** java.awt.EventQueue 1.1

- static void invokeLater(Runnable runnable) **1.2**  
causes the run method of the runnable object to be executed in the event dispatch thread after pending events have been processed.
- static void invokeAndWait(Runnable runnable) **1.2**  
causes the run method of the runnable object to be executed in the event dispatch thread after pending events have been processed. This call blocks until the run method has terminated.
- static boolean isDispatchThread() **1.2**  
returns true if the thread executing this method is the event dispatch thread.

**Using the Swing Worker**

When a user issues a command for which processing takes a long time, you will want to fire up a new thread to do the work. As you saw in the preceding section, that thread should use the `EventQueue.invokeLater` method to update the user interface.

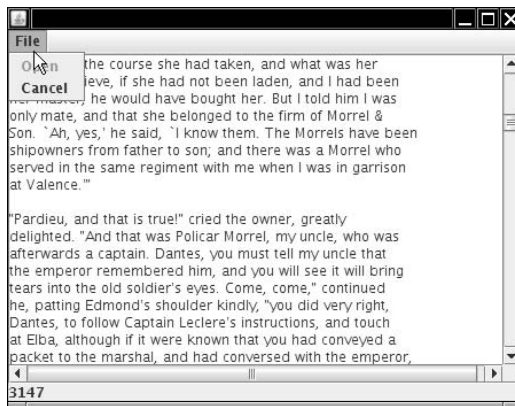
Several authors have produced convenience classes to ease this programming task, and one of these classes has made its way into Java SE 6. In this section, we describe that `SwingWorker` class.

The program in Listing 14–15 has commands for loading a text file and for canceling the file loading process. You should try the program with a long file, such as the full text of *The Count of Monte Cristo*, supplied in the `gutenberg` directory of the book's companion code. The file is loaded in a separate thread. While the file is read, the `Open` menu item is disabled and the `Cancel` item is enabled (see Figure 14–10). After each line is read, a line counter in the status bar is updated. After the reading process is complete, the `Open` menu item is reenabled, the `Cancel` item is disabled, and the status line text is set to `Done`.

This example shows the typical UI activities of a background task:

- After each work unit, update the UI to show progress.
- After the work is finished, make a final change to the UI.

The `SwingWorker` class makes it easy to implement such a task. You override the `doInBackground` method to do the time-consuming work and occasionally call `publish` to communicate work progress. This method is executed in a worker thread. The `publish` method causes a process method to execute in the event dispatch thread to deal with the progress data. When the work is complete, the `done` method is called in the event dispatch thread so that you can finish updating the UI.



**Figure 14–10** Loading a file in a separate thread

Whenever you want to do some work in the worker thread, construct a new worker. (Each worker object is meant to be used only once.) Then call the `execute` method. You will typically call `execute` on the event dispatch thread, but that is not a requirement.

It is assumed that a worker produces a result of some kind; therefore, `SwingWorker<T, V>` implements `Future<T>`. This result can be obtained by the `get` method of the `Future` interface. Since the `get` method blocks until the result is available, you don't want to call it immediately after calling `execute`. It is a good idea to call it only when you know that the work has been completed. Typically, you call `get` from the `done` method. (There is no requirement to call `get`. Sometimes, processing the progress data is all you need.)

Both the intermediate progress data and the final result can have arbitrary types. The `SwingWorker` class has these types as type parameters. A `SwingWorker<T, V>` produces a result of type `T` and progress data of type `V`.



To cancel the work in progress, use the `cancel` method of the `Future` interface. When the work is canceled, the `get` method throws a `CancellationException`.

As already mentioned, the worker thread's call to `publish` will cause calls to `process` on the event dispatch thread. For efficiency, the results of several calls to `publish` may be batched up in a single call to `process`. The `process` method receives a `List` containing all intermediate results.

Let us put this mechanism to work for reading in a text file. As it turns out, a `JTextArea` is quite slow. Appending lines from a long text file (such as all lines in *The Count of Monte Cristo*) takes considerable time.

To show the user that progress is being made, we want to display the number of lines read in a status line. Thus, the progress data consist of the current line number and the current line of text. We package these into a trivial inner class:

```
private class ProgressData
{
    public int number;
    public String line;
}
```

The final result is the text that has been read into a `StringBuilder`. Thus, we need a `SwingWorker<StringBuilder, ProgressData>`.

In the `doInBackground` method, we read a file, a line at a time. After each line, we call `publish` to publish the line number and the text of the current line.

```
@Override public StringBuilder doInBackground() throws IOException, InterruptedException
{
    int lineNumber = 0;
    Scanner in = new Scanner(new FileInputStream(file));
    while (in.hasNextLine())
    {
        String line = in.nextLine();
        lineNumber++;
        text.append(line);
        text.append("\n");
        ProgressData data = new ProgressData();
        data.number = lineNumber;
        data.line = line;
        publish(data);
        Thread.sleep(1); // to test cancellation; no need to do this in your programs
    }
    return text;
}
```

We also sleep for a millisecond after every line so that you can test cancellation without getting stressed out, but you wouldn't want to slow down your own programs by sleeping. If you comment out this line, you will find that *The Count of Monte Cristo* loads quite quickly, with only a few batched user interface updates.



**NOTE:** You can make this program behave quite smoothly by updating the text area from the worker thread, but this is not possible for most Swing components. We show you the general approach in which all component updates occur in the event dispatch thread.

In the process method, we ignore all line numbers but the last one, and we concatenate all lines for a single update of the text area.

```
@Override public void process(List<ProgressData> data)
{
    if (isCancelled()) return;
    StringBuilder b = new StringBuilder();
    statusLine.setText("" + data.get(data.size() - 1).number);
    for (ProgressData d : data) { b.append(d.line); b.append("\n"); }
    textArea.append(b.toString());
}
```

In the done method, the text area is updated with the complete text, and the Cancel menu item is disabled.

Note how the worker is started in the event listener for the Open menu item.

This simple technique allows you to execute time-consuming tasks while keeping the user interface responsive.

#### Listing 14–15 SwingWorkerTest.java

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.io.*;
4. import java.util.*;
5. import java.util.List;
6. import java.util.concurrent.*;
7.
8. import javax.swing.*;
9.
10. /**
11.  * This program demonstrates a worker thread that runs a potentially time-consuming task.
12.  * @version 1.1 2007-05-18
13.  * @author Cay Horstmann
14.  */
15. public class SwingWorkerTest
16. {
17.     public static void main(String[] args) throws Exception
18.     {
19.         EventQueue.invokeLater(new Runnable()
20.         {
21.             public void run()
22.             {
23.                 JFrame frame = new SwingWorkerFrame();
24.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25.                 frame.setVisible(true);
26.             }
27.         });
28.     }
29. }
30.
```

**Listing 14-15** SwingWorkerTest.java (continued)

```
31. /**
32.  * This frame has a text area to show the contents of a text file, a menu to open a file and
33.  * cancel the opening process, and a status line to show the file loading progress.
34.  */
35. class SwingWorkerFrame extends JFrame
36. {
37.     public SwingWorkerFrame()
38.     {
39.         chooser = new JFileChooser();
40.         chooser.setCurrentDirectory(new File("."));
41.
42.         textArea = new JTextArea();
43.         add(new JScrollPane(textArea));
44.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
45.
46.         statusLine = new JLabel(" ");
47.         add(statusLine, BorderLayout.SOUTH);
48.
49.         JMenuBar menuBar = new JMenuBar();
50.         setJMenuBar(menuBar);
51.
52.         JMenu menu = new JMenu("File");
53.         menuBar.add(menu);
54.
55.         JMenuItem openItem = new JMenuItem("Open");
56.         menu.add(openItem);
57.         openItem.addActionListener(new ActionListener()
58.         {
59.             public void actionPerformed(ActionEvent event)
60.             {
61.                 // show file chooser dialog
62.                 int result = chooser.showOpenDialog(null);
63.
64.                 // if file selected, set it as icon of the label
65.                 if (result == JFileChooser.APPROVE_OPTION)
66.                 {
67.                     textArea.setText("");
68.                     openItem.setEnabled(false);
69.                     textReader = new TextReader(chooser.getSelectedFile());
70.                     textReader.execute();
71.                     cancelItem.setEnabled(true);
72.                 }
73.             }
74.         });
75.
76.         JMenuItem cancelItem = new JMenuItem("Cancel");
77.         menu.add(cancelItem);
78.         cancelItem.setEnabled(false);
79.         cancelItem.addActionListener(new ActionListener()
```

**Listing 14–15** SwingWorkerTest.java (continued)

```
80.     {
81.         public void actionPerformed(ActionEvent event)
82.         {
83.             textReader.cancel(true);
84.         }
85.     });
86. }
87.
88. private class ProgressData
89. {
90.     public int number;
91.     public String line;
92. }
93.
94. private class TextReader extends SwingWorker<StringBuilder, ProgressData>
95. {
96.     public TextReader(File file)
97.     {
98.         this.file = file;
99.     }
100.
101.     // the following method executes in the worker thread; it doesn't touch Swing components
102.
103.     @Override
104.     public StringBuilder doInBackground() throws IOException, InterruptedException
105.     {
106.         int lineNumber = 0;
107.         Scanner in = new Scanner(new FileInputStream(file));
108.         while (in.hasNextLine())
109.         {
110.             String line = in.nextLine();
111.             lineNumber++;
112.             text.append(line);
113.             text.append("\n");
114.             ProgressData data = new ProgressData();
115.             data.number = lineNumber;
116.             data.line = line;
117.             publish(data);
118.             Thread.sleep(1); // to test cancellation; no need to do this in your programs
119.         }
120.         return text;
121.     }
122.
123.     // the following methods execute in the event dispatch thread
124.
125.     @Override
126.     public void process(List<ProgressData> data)
127.     {
128.         if (isCancelled()) return;
```

**Listing 14–15** SwingWorkerTest.java (continued)

```
129.     StringBuilder b = new StringBuilder();
130.     statusLine.setText("" + data.get(data.size() - 1).number);
131.     for (ProgressData d : data)
132.     {
133.         b.append(d.line);
134.         b.append("\n");
135.     }
136.     textArea.append(b.toString());
137. }
138.
139. @Override
140. public void done()
141. {
142.     try
143.     {
144.         StringBuilder result = get();
145.         textArea.setText(result.toString());
146.         statusLine.setText("Done");
147.     }
148.     catch (InterruptedException ex)
149.     {
150.     }
151.     catch (CancellationException ex)
152.     {
153.         textArea.setText("");
154.         statusLine.setText("Cancelled");
155.     }
156.     catch (ExecutionException ex)
157.     {
158.         statusLine.setText("" + ex.getCause());
159.     }
160.
161.     cancelItem.setEnabled(false);
162.     openItem.setEnabled(true);
163. }
164.
165. private File file;
166. private StringBuilder text = new StringBuilder();
167. };
168.
169. private JFileChooser chooser;
170. private JTextArea textArea;
171. private JLabel statusLine;
172. private JMenuItem openItem;
173. private JMenuItem cancelItem;
174. private SwingWorker<StringBuilder, ProgressData> textReader;
175.
176. public static final int DEFAULT_WIDTH = 450;
177. public static final int DEFAULT_HEIGHT = 350;
178. }
```

**API** `javax.swing.SwingWorker<T, V>` 6

- `abstract T doInBackground()`  
override this method to carry out the background task and to return the result of the work.
- `void process(List<V> data)`  
override this method to process intermediate progress data in the event dispatch thread.
- `void publish(V... data)`  
forwards intermediate progress data to the event dispatch thread. Call this method from `doInBackground`.
- `void execute()`  
schedules this worker for execution on a worker thread.
- `SwingWorker.StateValue getState()`  
gets the state of this worker, one of `PENDING`, `STARTED`, or `DONE`.

**The Single-Thread Rule**

Every Java application starts with a `main` method that runs in the main thread. In a Swing program, the main thread is short-lived. It schedules the construction of the user interface in the event dispatch thread and then exits. After the user interface construction, the event dispatch thread processes event notifications, such as calls to `actionPerformed` or `paintComponent`. Other threads, such as the thread that posts events into the event queue, are running behind the scenes, but those threads are invisible to the application programmer.

Earlier in the chapter, we introduced the single-thread rule: “Do not touch Swing components in any thread other than the event dispatch thread.” In this section, we investigate that rule further.

There are a few exceptions to the single-thread rule.

- You can safely add and remove event listeners in any thread. Of course, the listener methods will be invoked in the event dispatch thread.
- A small number of Swing methods are thread safe. They are specially marked in the API documentation with the sentence “*This method is thread safe, although most Swing methods are not.*” The most useful among these thread-safe methods are

```

JTextComponent.setText
JTextArea.insert
JTextArea.append
JTextArea.replaceRange
JComponent.repaint
JComponent.revalidate

```



**NOTE:** We used the `repaint` method many times in this book, but the `revalidate` method is less common. Its purpose is to force a layout of a component after the contents have changed. The traditional AWT has a `validate` method to force the layout of a component. For Swing components, you should simply call `revalidate` instead. (However, to force the layout of a `JFrame`, you still need to call `validate`—a `JFrame` is a `Component` but not a `JComponent`.)

Historically, the single-thread rule was more permissive. Any thread was allowed to construct components, set their properties, and add them into containers, as long as none of the components had been *realized*. A component is realized if it can receive paint or validation events. This is the case as soon as the `setVisible(true)` or `pack(!)` methods have been invoked on the component, or if the component has been added to a container that has been realized.

That version of the single-thread rule was convenient. It allowed you to create the GUI of in the `main` method and then call `setVisible(true)` on the top-level frame of the application. There was no bothersome scheduling of a `Runnable` on the event dispatch thread.

Unfortunately, some component implementors did not pay attention to the subtleties of the original single-thread rule. They launched activities on the event dispatch thread without ever bothering to check whether the component was realized. For example, if you call `setSelectionStart` or `setSelectionEnd` on a `JTextComponent`, a caret movement is scheduled in the event dispatch thread, even if the component is not visible.

It might well have been possible to detect and fix these problems, but the Swing designers took the easy way out. They decreed that it is never safe to access components from any thread other than the event dispatch thread. Therefore, you need to construct the user interface in the event dispatch thread, using the call to `EventQueue.invokeLater` that you have seen in all our sample programs.

Of course, there are plenty of programs that are not so careful and live by the old version of the single-thread rule, initializing the user interface on the main thread. Those programs incur the slight risk that some of the user interface initialization causes actions on the event dispatch thread that conflict with actions on the main thread. As we said in Chapter 7, you don't want to be one of the unlucky few who run into trouble and waste time debugging an intermittent threading bug. Therefore, you should simply follow the strict single-thread rule.

You have now reached the end of Volume I of *Core Java*. This volume covered the fundamentals of the Java programming language and the parts of the standard library that you need for most programming projects. We hope that you enjoyed your tour through the Java fundamentals and that you found useful information along the way. For advanced topics, such as networking, advanced AWT/Swing, security, and internationalization, please turn to Volume II.





---

# *Appendix*

## JAVA KEYWORDS

<b>Keyword</b>	<b>Meaning</b>	<b>See Chapter</b>
abstract	an abstract class or method	5
assert	used to locate internal program errors	11
boolean	the Boolean type	3
break	breaks out of a switch or loop	3
byte	the 8-bit integer type	3
case	a case of a switch	3
catch	the clause of a try block catching an exception	11
char	the Unicode character type	3
class	defines a class type	4
const	not used	
continue	continues at the end of a loop	3
default	the default clause of a switch	3
do	the top of a do/while loop	3

<b>Keyword</b>	<b>Meaning</b>	<b>See Chapter</b>
double	the double-precision floating-number type	3
else	the else clause of an if statement	3
enum	an enumerated type	3
extends	defines the parent class of a class	4
final	a constant, or a class or method that cannot be overridden	5
finally	the part of a try block that is always executed	11
float	the single-precision floating-point type	3
for	a loop type	3
goto	not used	
if	a conditional statement	3
implements	defines the interface(s) that a class implements	6
import	imports a package	4
instanceof	tests if an object is an instance of a class	5
int	the 32-bit integer type	3
interface	an abstract type with methods that a class can implement	6
long	the 64-bit long integer type	3
native	a method implemented by the host system	11 (Vol. II)
new	allocates a new object or array	3
null	a null reference	3
package	a package of classes	4
private	a feature that is accessible only by methods of this class	4
protected	a feature that is accessible only by methods of this class, its children, and other classes in the same package	5
public	a feature that is accessible by methods of all classes	4

<b>Keyword</b>	<b>Meaning</b>	<b>See Chapter</b>
return	returns from a method	3
short	the 16-bit integer type	3
static	a feature that is unique to its class, not to objects of its class	3
strictfp	Use strict rules for floating-point computations	2
super	the superclass object or constructor	5
switch	a selection statement	3
synchronized	a method or code block that is atomic to a thread	14
this	the implicit argument of a method, or a constructor of this class	4
throw	throws an exception	11
throws	the exceptions that a method can throw	11
transient	marks data that should not be persistent	1 (Vol. II)
try	a block of code that traps exceptions	11
void	denotes a method that returns no value	3
volatile	ensures that a field is coherently accessed by multiple threads	14
while	a loop	3