



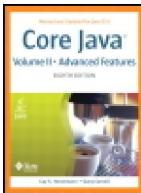
2015. java学习交流群

8918 1289

每天有免费的Java学习课堂

——学习Java就是这么简单

——为Java而燃烧——



Core Java™ Volume II—Advanced Features, Eighth Edition
by Cay S. Horstmann; Gary Cornell

Publisher: Prentice Hall

Pub Date: April 08, 2008

Print ISBN-10: 0-13-235479-9

Print ISBN-13: 978-0-13-235479-0

eText ISBN-10: 0-13-714448-2

eText ISBN-13: 978-0-13-714448-8

Pages: 1056

[Table of Contents](#) | [Index](#)

Overview

The revised edition of the classic Core Java™, Volume II—Advanced Features, covers advanced user-interface programming and the enterprise features of the Java SE 6 platform. Like Volume I (which covers the core language and library features), this volume has been updated for Java SE 6 and new coverage is highlighted throughout. All sample programs have been carefully crafted to illustrate the latest programming techniques, displaying best-practices solutions to the types of real-world problems professional developers encounter.

Volume II includes new sections on the StAX API, JDBC 4, compiler API, scripting framework, splash screen and tray APIs, and many other Java SE 6 enhancements. In this book, the authors focus on the more advanced features of the Java language, including complete coverage of

- Streams and Files
- Networking
- Database programming
- XML
- JNDI and LDAP
- Internationalization
- Advanced GUI components
- Java 2D and advanced AWT
- JavaBeans
- Security
- RMI and Web services
- Collections
- Annotations
- Native methods

For thorough coverage of Java fundamentals—including interfaces and inner classes, GUI programming with Swing, exception handling, generics, collections, and concurrency—look for the eighth edition of Core Java™, Volume I—Fundamentals (ISBN: 978-0-13-235476-9).



Core Java™ Volume II—Advanced Features, Eighth Edition by Cay S. Horstmann; Gary Cornell

Publisher: Prentice Hall

Pub Date: April 08, 2008

Print ISBN-10: 0-13-235479-9

Print ISBN-13: 978-0-13-235479-0

eText ISBN-10: 0-13-714448-2

eText ISBN-13: 978-0-13-714448-8

Pages: 1056

[Table of Contents](#) | [Index](#)

[Copyright](#)

[Preface](#)

[Acknowledgments](#)

[Chapter 1. Streams and Files](#)

[Streams](#)

[Text Input and Output](#)

[Reading and Writing Binary Data](#)

[ZIP Archives](#)

[Object Streams and Serialization](#)

[File Management](#)

[New I/O](#)

[Regular Expressions](#)

[Chapter 2. XML](#)

[Introducing XML](#)

[Parsing an XML Document](#)

[Validating XML Documents](#)

[Locating Information with XPath](#)

[Using Namespaces](#)

[Streaming Parsers](#)

[Generating XML Documents](#)

[XSL Transformations](#)

[Chapter 3. Networking](#)

[Connecting to a Server](#)

[Implementing Servers](#)

[Interruptible Sockets](#)

[Sending E-Mail](#)

[Making URL Connections](#)

[Chapter 4. Database Programming](#)

[The Design of JDBC](#)

[The Structured Query Language](#)

[JDBC Configuration](#)

[Executing SQL Statements](#)

[Query Execution](#)

[Scrollable and Updatable Result Sets](#)

[Row Sets](#)

[Metadata](#)

[Transactions](#)

[Connection Management in Web and Enterprise Applications](#)

[Introduction to LDAP](#)

[Chapter 5. Internationalization](#)

[Locales](#)

[Number Formats](#)

[Date and Time](#)

[Collation](#)[Message Formatting](#)[Text Files and Character Sets](#)[Resource Bundles](#)[A Complete Example](#)

[Chapter 6. Advanced Swing](#)

[Lists](#)[Tables](#)[Trees](#)[Text Components](#)[Progress Indicators](#)[Component Organizers](#)

[Chapter 7. Advanced AWT](#)

[The Rendering Pipeline](#)[Shapes](#)[Areas](#)[Strokes](#)[Paint](#)[Coordinate Transformations](#)[Clipping](#)[Transparency and Composition](#)[Rendering Hints](#)[Readers and Writers for Images](#)[Image Manipulation](#)[Printing](#)[The Clipboard](#)[Drag and Drop](#)[Platform Integration](#)

[Chapter 8. JavaBeans Components](#)

[Why Beans?](#)[The Bean-Writing Process](#)[Using Beans to Build an Application](#)[Naming Patterns for Bean Properties and Events](#)[Bean Property Types](#)[BeanInfo Classes](#)[Property Editors](#)[Customizers](#)[JavaBeans Persistence](#)

[Chapter 9. Security](#)

[Class Loaders](#)[Bytecode Verification](#)[Security Managers and Permissions](#)[User Authentication](#)[Digital Signatures](#)[Code Signing](#)[Encryption](#)

[Chapter 10. Distributed Objects](#)

[The Roles of Client and Server](#)[Remote Method Calls](#)[The RMI Programming Model](#)[Parameters and Return Values in Remote Methods](#)[Remote Object Activation](#)[Web Services and JAX-WS](#)

[Chapter 11. Scripting, Compiling, and Annotation Processing](#)

[Scripting for the Java Platform](#)

[Using Annotations](#)[Annotation Syntax](#)[Standard Annotations](#)[Source-Level Annotation Processing](#)[Bytecode Engineering](#)

[Chapter 12. Native Methods](#)

[Calling a C Function from a Java Program](#)[Numeric Parameters and Return Values](#)[String Parameters](#)[Accessing Fields](#)[Encoding Signatures](#)[Calling Java Methods](#)[Accessing Array Elements](#)[Handling Errors](#)[Using the Invocation API](#)[A Complete Example: Accessing the Windows Registry](#)[Index](#)



Copyright

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 U.S. 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 TYPO-GRAFICAL 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 PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

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

Visit us on the Web: informit.com/ph

Library of Congress Cataloging-in-Publication Data

Horstmann, Cay S., 1959-

Core Java. Volume 1, Fundamentals / Cay S. Horstmann, Gary Cornell. —
8th ed.

p. cm.

Includes index.

ISBN 978-0-13-235476-9 (pbk. : alk. paper) 1. Java (Computer program language) I. Cornell, Gary. II. Title. III. Title: Fundamentals. IV. Title: Core Java fundamentals.

QA76.73.J38H6753 2008

005.13'3—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 likewise. For information regarding permissions, write to: Pearson Education, Inc., Rights and Contracts Department, 501 Boylston Street, Suite 900, Boston, MA 02116, Fax: 617-671-3447.

ISBN-13: 978-0-13-235479-0

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

First printing, April 2008





Preface

To the Reader

The book you have in your hands is the second volume of the eighth edition of Core Java™, fully updated for Java SE 6. The first volume covers the essential features of the language; this volume covers the advanced topics that a programmer will need to know for professional software development. Thus, as with the first volume and the previous editions of this book, we are still targeting programmers who want to put Java technology to work on real projects.

Please note: If you are an experienced developer who is comfortable with advanced language features such as inner classes and generics, you need not have read the first volume in order to benefit from this volume. While we do refer to sections of the previous volume when appropriate (and, of course, hope you will buy or have bought Volume I), you can find the needed background material in any comprehensive introductory book about the Java platform.

Finally, when any book is being written, errors and inaccuracies are inevitable. We would very much like to hear about them should you find any in this book. Of course, we would prefer to hear about them only once. For this reason, we have put up a web site at <http://horstmann.com/corejava> with an FAQ, bug fixes, and workarounds. Strategically placed at the end of the bug report web page (to encourage you to read the previous reports) is a form that you can use to report bugs or problems and to send suggestions for improvements to future editions.

About This Book

The chapters in this book are, for the most part, independent of each other. You should be able to delve into whatever topic interests you the most and read the chapters in any order.

The topic of [Chapter 1](#) is input and output handling. In Java, all I/O is handled through so-called streams. Streams let you deal, in a uniform manner, with communications among various sources of data, such as files, network connections, or memory blocks. We include detailed coverage of the reader and writer classes, which make it easy to deal with Unicode. We show you what goes on under the hood when you use the object serialization mechanism, which makes saving and loading objects easy and convenient. Finally, we cover the "new I/O" classes (which were new when they were added to Java SE 1.4) that support efficient file operations, and the regular expression library.

[Chapter 2](#) covers XML. We show you how to parse XML files, how to generate XML, and how to use XSL transformations. As a useful example, we show you how to specify the layout of a Swing form in XML. This chapter has been updated to include the XPath API, which makes "finding needles in XML haystacks" much easier.

[Chapter 3](#) covers the networking API. Java makes it phenomenally easy to do complex network programming. We show you how to make network connections to servers, how to implement your own servers, and how to make HTTP connections.

[Chapter 4](#) covers database programming. The main focus is on JDBC, the Java database connectivity API that lets Java programs connect to relational databases. We show you how to write useful programs to handle realistic database chores, using a core subset of the JDBC API. (A complete treatment of the JDBC API would require a book almost as long as this one.) We finish the chapter with a brief introduction into hierarchical databases and discuss JNDI (the Java Naming and Directory Interface) and LDAP (the Lightweight Directory Access Protocol).

[Chapter 5](#) discusses a feature that we believe can only grow in importance—internationalization. The Java programming language is one of the few languages designed from the start to handle Unicode, but the internationalization support in the Java platform goes much further. As a result, you can internationalize Java applications so that they not only cross platforms but cross country boundaries as well. For example, we show you how to write a retirement calculator applet that uses either English, German, or Chinese languages—depending on the locale of the browser.

[Chapter 6](#) contains all the Swing material that didn't make it into Volume I, especially the important but complex tree and table components. We show the basic uses of editor panes, the Java implementation of a "multiple document" interface, progress indicators that you use in multithreaded programs, and "desktop integration features" such as splash screens and support for the system tray. Again, we focus on the most useful constructs that you are likely to encounter in practical programming because an encyclopedic coverage of the entire Swing library would fill several volumes and would only be of interest to dedicated taxonomists.

[Chapter 7](#) covers the Java 2D API, which you can use to create realistic drawings and special effects. The chapter also covers some advanced features of the AWT (Abstract Windowing Toolkit) that seemed too specialized for coverage in Volume I but are, nonetheless, techniques that should be part of every programmer's toolkit. These features include printing and the APIs for cut-and-paste and drag-and-drop.

[Chapter 8](#) shows you what you need to know about the component API for the Java platform—JavaBeans. We show you how to write your own beans that other programmers can manipulate in integrated builder environments. We conclude this chapter by showing you how you can use JavaBeans persistence to store your own data in a format that—unlike object serialization—is suitable for long-term storage.

[Chapter 9](#) takes up the Java security model. The Java platform was designed from the ground up to be secure, and this chapter takes you under the hood to see how this design is implemented. We show you how to write your own class loaders and security managers for special-purpose applications. Then, we take up the security API that allows for such important features as message and code signing, authorization and authentication, and encryption. We conclude with examples that use the AES and RSA encryption algorithms.

[Chapter 10](#) covers distributed objects. We cover RMI (Remote Method Invocation) in detail. This API lets you work with Java objects that are distributed over multiple machines. We then briefly discuss web services and show you an example in which a Java program communicates with the Amazon Web Service.

[Chapter 11](#) discusses three techniques for processing code. The scripting and compiler APIs, introduced in Java SE 6, allow your program to call code in scripting languages such as JavaScript or Groovy, and to compile Java code. Annotations allow you to add arbitrary information (sometimes called metadata) to a Java program. We show you how annotation processors can harvest these annotations at the source or class file level, and how annotations can be used to influence the behavior of classes at runtime. Annotations are only useful with tools, and we hope that our discussion will help you select useful annotation processing tools for your needs.

[Chapter 12](#) takes up native methods, which let you call methods written for a specific machine such as the Microsoft Windows API. Obviously, this feature is controversial: Use native methods, and the cross-platform nature of the Java platform vanishes. Nonetheless, every serious programmer writing Java applications for specific platforms needs to know these techniques. At times, you need to turn to the operating system's API for your target platform when you interact with a device or service that is not supported by the Java platform. We illustrate this by showing you how to access the registry API in Windows from a Java program.

As always, all chapters have been completely revised for the latest version of Java. Outdated material has been removed, and the new APIs of Java SE 6 are covered in detail.

Conventions

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

Note



Notes are tagged with a checkmark button that looks like this.

Tip



Helpful tips are tagged with this exclamation point button.

Caution



Notes that warn of pitfalls or dangerous situations are tagged with an x button.

C++ Note



There are a number of C++ notes that explain the difference between the Java programming language and C++. You can skip them if you aren't interested in C++.



Application Programming Interface

The Java platform 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. These descriptions are a bit more informal but occasionally a little more informative than those in the official on-line API documentation.

Programs whose source code is included in the companion code for this book are listed as examples; for instance,

Listing 11.1. `ScriptTest.java`

You can download the companion code from <http://horstmann.com/corejava>.





Acknowledgments

Writing a book is always a monumental effort, and rewriting doesn't seem to be much easier, especially with such a rapid rate of 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), Lance Anderson (Sun Microsystems), 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), Robert Evans (Senior Staff, The Johns Hopkins University Applied Physics Lab), David Geary (Sabreware), 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 Kimmet (Preview Systems), Chris Laffra, Charlie Lai (Sun Microsystems), Angelika Langer, Doug Langston, Hang Lau (McGill University), Mark Lawrence, Doug Lea (SUNY Oswego), Gregory Longshore, Bob Lynch (Lynch Associates), Philip Milne (consultant), Mark Morrissey (The Oregon Graduate Institute), Mahesh Neelakanta (Florida Atlantic University), Hao Pham, Paul Phlion, Blake Ragsdell, Ylber Ramadani (Ryerson University), 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), Richard Slywczak (NASA/Glenn Research Center), 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, 2008





Chapter 1. Streams and Files

- [STREAMS](#)
- [TEXT INPUT AND OUTPUT](#)
- [READING AND WRITING BINARY DATA](#)
- [ZIP ARCHIVES](#)
- [OBJECT STREAMS AND SERIALIZATION](#)
- [FILE MANAGEMENT](#)
- [NEW I/O](#)
- [REGULAR EXPRESSIONS](#)

In this chapter, we cover the Java application programming interfaces (APIs) for input and output. You will learn how to access files and directories and how to read and write data in binary and text format. This chapter also shows you the object serialization mechanism that lets you store objects as easily as you can store text or numeric data. Next, we turn to several improvements that were made in the "new I/O" package `java.nio`, introduced in Java SE 1.4. We finish the chapter with a discussion of regular expressions, even though they are not actually related to streams and files. We couldn't find a better place to handle that topic, and apparently neither could the Java team—the regular expression API specification was attached to the specification request for the "new I/O" features of Java SE 1.4.

Streams

In the Java API, an object from which we can read a sequence of bytes is called an input stream. An object to which we can write a sequence of bytes is called an output stream. These sources and destinations of byte sequences can be—and often are—files, but they can also be network connections and even blocks of memory. The abstract classes `InputStream` and `OutputStream` form the basis for a hierarchy of input/output (I/O) classes.

Because byte-oriented streams are inconvenient for processing information stored in Unicode (recall that Unicode uses multiple bytes per character), there is a separate hierarchy of classes for processing Unicode characters that inherit from the abstract `Reader` and `Writer` classes. These classes have read and write operations that are based on two-byte Unicode code units rather than on single-byte characters.

Reading and Writing Bytes

The `InputStream` class has an abstract method:

```
abstract int read()
```

This method reads one byte and returns the byte that was read, or -1 if it encounters the end of the input source. The designer of a concrete input stream class overrides this method to provide useful functionality. For example, in the `FileInputStream` class, this method reads one byte from a file. `System.in` is a predefined object of a subclass of `InputStream` that allows you to read information from the keyboard.

The `InputStream` class also has nonabstract methods to read an array of bytes or to skip a number of bytes. These methods call the abstract `read` method, so subclasses need to override only one method.

Similarly, the `OutputStream` class defines the abstract method

```
abstract void write(int b)
```

which writes one byte to an output location.

Both the `read` and `write` methods block until the bytes are actually read or written. This means that if the stream cannot immediately be accessed (usually because of a busy network connection), the current thread blocks. This gives other threads the chance to do useful work while the method is waiting for the stream to again become available.

The `available` method lets you check the number of bytes that are currently available for reading. This means a fragment like the following is unlikely to block:

```
int bytesAvailable = in.available();
if (bytesAvailable > 0)
{
    byte[] data = new byte[bytesAvailable];
    in.read(data);
}
```

When you have finished reading or writing to a stream, close it by calling the `close` method. This call frees up operating system resources that are in limited supply. If an application opens too many streams without closing them, system resources can become depleted. Closing an output stream also flushes the buffer used for the output stream: any characters that were temporarily placed in a buffer so that they could be delivered as a larger packet are sent off. In particular, if you do not close a file, the last packet of bytes might never be delivered. You can also manually flush the output with the `flush` method.

Even if a stream class provides concrete methods to work with the raw `read` and `write` functions, application programmers rarely use them. The data that you are interested in probably contain numbers, strings, and objects, not raw bytes.

Java gives you many stream classes derived from the basic `InputStream` and `OutputStream` classes that let you work with data in the forms that you usually use rather than at the byte level.



java.io.InputStream 1.0

- `abstract int read()`

reads a byte of data and returns the byte read. The `read` method returns a -1 at the end of the stream.

- `int read(byte[] b)`

reads into an array of bytes and returns the actual number of bytes read, or -1 at the end of the stream. The `read` method reads at most `b.length` bytes.

- `int read(byte[] b, int off, int len)`

reads into an array of bytes. The `read` method returns the actual number of bytes read, or -1 at the end of the stream.

Parameters: `b` The array into which the data is read

`off` The offset into `b` where the first bytes should be placed

`len` The maximum number of bytes to read

- `long skip(long n)`

skips `n` bytes in the input stream. Returns the actual number of bytes skipped (which may be less than `n` if the end of the stream was encountered).

- `int available()`

returns the number of bytes available without blocking. (Recall that blocking means that the current thread loses its turn.)

- `void close()`

closes the input stream.

- `void mark(int readlimit)`

puts a marker at the current position in the input stream. (Not all streams support this feature.) If more than `readlimit` bytes have been read from the input stream, then the stream is allowed to forget the marker.

- `void reset()`
returns to the last marker. Subsequent calls to `read` reread the bytes. If there is no current marker, then the stream is not reset.
- `boolean markSupported()`
returns `true` if the stream supports marking.

API

java.io.OutputStream 1.0

- `abstract void write(int n)`
writes a byte of data.
- `void write(byte[] b)`
- `void write(byte[] b, int off, int len)`
writes all bytes or a range of bytes in the array `b`.

Parameters:	<code>b</code>	The array from which to write the data
	<code>off</code>	The offset into <code>b</code> to the first byte that will be written
	<code>len</code>	The number of bytes to write
- `void close()`
flushes and closes the output stream.
- `void flush()`
flushes the output stream; that is, sends any buffered data to its destination.

The Complete Stream Zoo

Unlike C, which gets by just fine with a single type `FILE*`, Java has a whole zoo of more than 60 (!) different stream types (see [Figures 1-1](#) and [1-2](#)).

Figure 1-1. Input and output stream hierarchy

[\[View full size image\]](#)

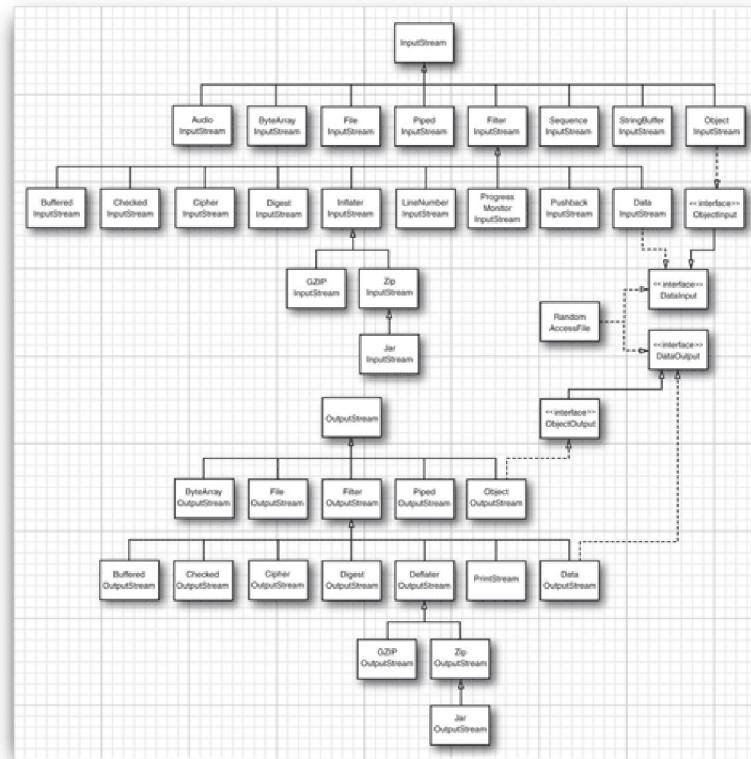
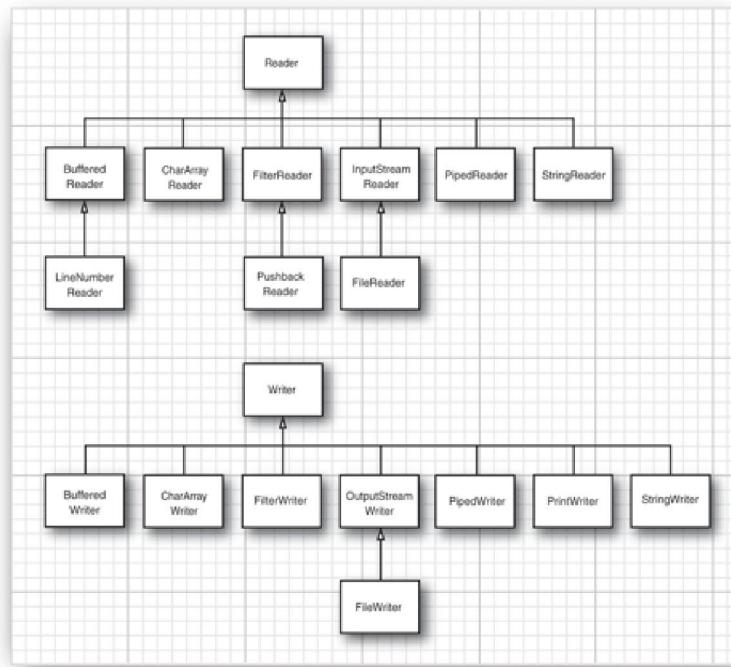


Figure 1-2. Reader and writer hierarchy

[[View full size image](#)]



Let us divide the animals in the stream class zoo by how they are used. There are separate hierarchies for classes that process bytes and characters. As you saw, the `InputStream` and `OutputStream` classes let you read and write individual bytes and arrays of bytes. These classes form the basis of the hierarchy shown in [Figure 1-1](#). To read and write strings and numbers, you need more capable subclasses. For example, `DataInputStream` and `DataOutputStream` let you read and write all the primitive Java types in binary format. Finally, there are streams that do useful stuff; for example, the `ZipInputStream` and `ZipOutputStream` that let you read and write files in the familiar ZIP compression format.

For Unicode text, on the other hand, you use subclasses of the abstract classes `Reader` and `Writer` (see [Figure 1-2](#)). The basic

methods of the `Reader` and `Writer` classes are similar to the ones for `InputStream` and `OutputStream`.

```
abstract int read()
abstract void write(int c)
```

The `read` method returns either a Unicode code unit (as an integer between 0 and 65535) or -1 when you have reached the end of the file. The `write` method is called with a Unicode code unit. (See Volume I, Chapter 3 for a discussion of Unicode code units.)

Java SE 5.0 introduced four additional interfaces: `Closeable`, `Flushable`, `Readable`, and `Appendable` (see [Figure 1-3](#)). The first two interfaces are very simple, with methods

```
void close() throws IOException
```

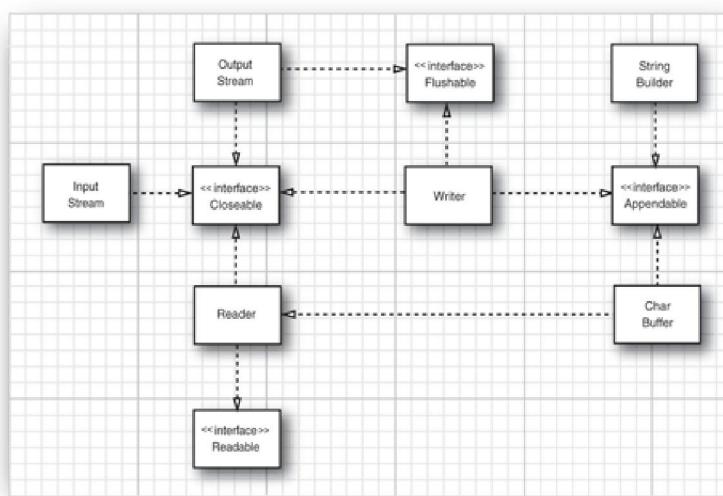
and

```
void flush()
```

respectively. The classes `InputStream`, `OutputStream`, `Reader`, and `Writer` all implement the `Closeable` interface. `OutputStream` and `Writer` implement the `Flushable` interface.

Figure 1-3. The `Closeable`, `Flushable`, `Readable`, and `Appendable` interfaces

[[View full size image](#)]



The `Readable` interface has a single method

```
int read(CharBuffer cb)
```

The `CharBuffer` class has methods for sequential and random read/write access. It represents an in-memory buffer or a memory-mapped file. (See "[The Buffer Data Structure](#)" on page [72](#) for details.)

The `Appendable` interface has two methods for appending single characters and character sequences:

```
Appendable append(char c)
Appendable append(CharSequence s)
```

The `CharSequence` interface describes basic properties of a sequence of `char` values. It is implemented by `String`, `CharBuffer`, `StringBuilder`, and `StringBuffer`.

Of the stream zoo classes, only `Writer` implements `Appendable`.

API**java.io.Closeable 5.0**

- void close()

closes this Closeable. This method may throw an IOException.

API**java.io.Flushable 5.0**

- void flush()

flushes this Flushable.

API**java.lang.Readable 5.0**

- int read(CharBuffer cb)

attempts to read as many char values into cb as it can hold. Returns the number of values read, or -1 if no further values are available from this Readable.

API**java.lang.Appendable 5.0**

- Appendable append(char c)
- Appendable append(CharSequence cs)

appends the given code unit, or all code units in the given sequence, to this Appendable; returns this.

API**java.lang.CharSequence 1.4**

- char charAt(int index)

returns the code unit at the given index.

- int length()

returns the number of code units in this sequence.

- CharSequence subSequence(int startIndex, int endIndex)

returns a CharSequence consisting of the code units stored at index startIndex to endIndex - 1.

- String toString()

returns a string consisting of the code units of this sequence.

Combining Stream Filters

`FileInputStream` and `FileOutputStream` give you input and output streams attached to a disk file. You give the file name or full path name of the file in the constructor. For example,

```
FileInputStream fin = new FileInputStream("employee.dat");
```

looks in the user directory for a file named "employee.dat".

Tip



Because all the classes in `java.io` interpret relative path names as starting with the user's working directory, you may want to know this directory. You can get at this information by a call to `System.getProperty("user.dir")`.

Like the abstract `InputStream` and `OutputStream` classes, these classes support only reading and writing on the byte level. That is, we can only read bytes and byte arrays from the object `fin`.

```
byte b = (byte) fin.read();
```

As you will see in the next section, if we just had a `DataInputStream`, then we could read numeric types:

```
DataInputStream din = . . .;
double s = din.readDouble();
```

But just as the `FileInputStream` has no methods to read numeric types, the `DataInputStream` has no method to get data from a file.

Java uses a clever mechanism to separate two kinds of responsibilities. Some streams (such as the `FileInputStream` and the input stream returned by the `openStream` method of the `URL` class) can retrieve bytes from files and other more exotic locations. Other streams (such as the `DataInputStream` and the `PrintWriter`) can assemble bytes into more useful data types. The Java programmer has to combine the two. For example, to be able to read numbers from a file, first create a `FileInputStream` and then pass it to the constructor of a `DataInputStream`.

```
FileInputStream fin = new FileInputStream("employee.dat");
DataInputStream din = new DataInputStream(fin);
double s = din.readDouble();
```

If you look at [Figure 1-1](#) again, you can see the classes `FilterInputStream` and `FilterOutputStream`. The subclasses of these files are used to add capabilities to raw byte streams.

You can add multiple capabilities by nesting the filters. For example, by default, streams are not buffered. That is, every call to `read` asks the operating system to dole out yet another byte. It is more efficient to request blocks of data instead and put them in a buffer. If you want buffering and the data input methods for a file, you need to use the following rather monstrous sequence of constructors:

```
DataInputStream din = new DataInputStream(
    new BufferedInputStream(
        new FileInputStream("employee.dat")));
```

Notice that we put the `DataInputStream` last in the chain of constructors because we want to use the `DataInputStream` methods, and we want them to use the buffered `read` method.

Sometimes you'll need to keep track of the intermediate streams when chaining them together. For example, when reading input, you often need to peek at the next byte to see if it is the value that you expect. Java provides the `PushbackInputStream` for this purpose.

```
PushbackInputStream pbin = new PushbackInputStream(
    new BufferedInputStream(
        new FileInputStream("employee.dat")));
```

Now you can speculatively read the next byte

```
int b = pbin.read();
```

and throw it back if it isn't what you wanted.

```
if (b != '<') pbin.unread(b);
```

But reading and unread are the only methods that apply to the pushback input stream. If you want to look ahead and also read numbers, then you need both a pushback input stream and a data input stream reference.

```
DataInputStream din = new DataInputStream(
    pbin = new PushbackInputStream(
        new BufferedInputStream(
            new FileInputStream("employee.dat"))));
```

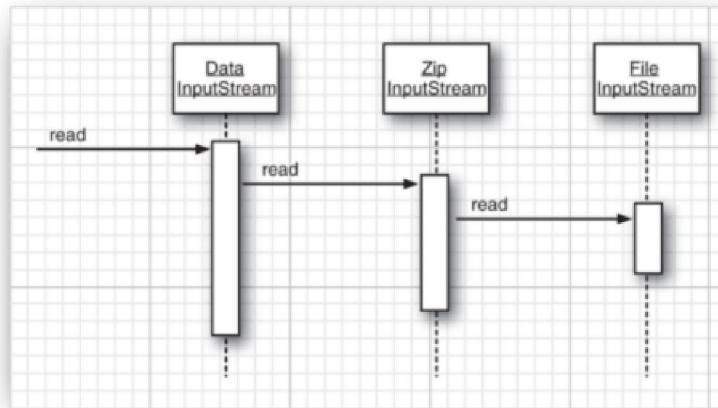
Of course, in the stream libraries of other programming languages, niceties such as buffering and lookahead are automatically taken care of, so it is a bit of a hassle in Java that one has to resort to combining stream filters in these cases. But the ability to mix and match filter classes to construct truly useful sequences of streams does give you an immense amount of flexibility. For example, you can read numbers from a compressed ZIP file by using the following sequence of streams (see [Figure 1-4](#)):

Code View:

```
ZipInputStream zin = new ZipInputStream(new FileInputStream("employee.zip"));
DataInputStream din = new DataInputStream(zin);
```

Figure 1-4. A sequence of filtered streams

[\[View full size image\]](#)



(See "[ZIP Archives](#)" on page [32](#) for more on Java's ability to handle ZIP files.)



java.io.FileInputStream 1.0

- `FileInputStream(String name)`
- `FileInputStream(File file)`

creates a new file input stream, using the file whose path name is specified by the `name` string or the `file` object. (The `File` class is described at the end of this chapter.) Path names that are not absolute are resolved relative to the working directory that was set when the VM started.



java.io.FileOutputStream 1.0

- `FileOutputStream(String name)`
- `FileOutputStream(String name, boolean append)`

- `FileOutputStream(File file)`
- `FileOutputStream(File file, boolean append)`

creates a new file output stream specified by the `name` string or the `file` object. (The `File` class is described at the end of this chapter.) If the `append` parameter is `true`, then data are added at the end of the file. An existing file with the same name will not be deleted. Otherwise, this method deletes any existing file with the same name.

API

java.io.BufferedInputStream 1.0

- `BufferedInputStream(InputStream in)`

creates a buffered stream. A buffered input stream reads characters from a stream without causing a device access every time. When the buffer is empty, a new block of data is read into the buffer.

API

java.io.BufferedOutputStream 1.0

- `BufferedOutputStream(OutputStream out)`

creates a buffered stream. A buffered output stream collects characters to be written without causing a device access every time. When the buffer fills up or when the stream is flushed, the data are written.

API

java.io.PushbackInputStream 1.0

- `PushbackInputStream(InputStream in)`
- `PushbackInputStream(InputStream in, int size)`

constructs a stream with one-byte lookahead or a pushback buffer of specified size.

- `void unread(int b)`

pushes back a byte, which is retrieved again by the next call to read.

Parameters: `b` The byte to be read again



Chapter 1. Streams and Files

- [STREAMS](#)
- [TEXT INPUT AND OUTPUT](#)
- [READING AND WRITING BINARY DATA](#)
- [ZIP ARCHIVES](#)
- [OBJECT STREAMS AND SERIALIZATION](#)

- [FILE MANAGEMENT](#)
- [NEW I/O](#)
- [REGULAR EXPRESSIONS](#)

In this chapter, we cover the Java application programming interfaces (APIs) for input and output. You will learn how to access files and directories and how to read and write data in binary and text format. This chapter also shows you the object serialization mechanism that lets you store objects as easily as you can store text or numeric data. Next, we turn to several improvements that were made in the "new I/O" package `java.nio`, introduced in Java SE 1.4. We finish the chapter with a discussion of regular expressions, even though they are not actually related to streams and files. We couldn't find a better place to handle that topic, and apparently neither could the Java team—the regular expression API specification was attached to the specification request for the "new I/O" features of Java SE 1.4.

Streams

In the Java API, an object from which we can read a sequence of bytes is called an input stream. An object to which we can write a sequence of bytes is called an output stream. These sources and destinations of byte sequences can be—and often are—files, but they can also be network connections and even blocks of memory. The abstract classes `InputStream` and `OutputStream` form the basis for a hierarchy of input/output (I/O) classes.

Because byte-oriented streams are inconvenient for processing information stored in Unicode (recall that Unicode uses multiple bytes per character), there is a separate hierarchy of classes for processing Unicode characters that inherit from the abstract `Reader` and `Writer` classes. These classes have read and write operations that are based on two-byte Unicode code units rather than on single-byte characters.

Reading and Writing Bytes

The `InputStream` class has an abstract method:

```
abstract int read()
```

This method reads one byte and returns the byte that was read, or -1 if it encounters the end of the input source. The designer of a concrete input stream class overrides this method to provide useful functionality. For example, in the `FileInputStream` class, this method reads one byte from a file. `System.in` is a predefined object of a subclass of `InputStream` that allows you to read information from the keyboard.

The `InputStream` class also has nonabstract methods to read an array of bytes or to skip a number of bytes. These methods call the abstract `read` method, so subclasses need to override only one method.

Similarly, the `OutputStream` class defines the abstract method

```
abstract void write(int b)
```

which writes one byte to an output location.

Both the `read` and `write` methods block until the bytes are actually read or written. This means that if the stream cannot immediately be accessed (usually because of a busy network connection), the current thread blocks. This gives other threads the chance to do useful work while the method is waiting for the stream to again become available.

The `available` method lets you check the number of bytes that are currently available for reading. This means a fragment like the following is unlikely to block:

```
int bytesAvailable = in.available();
if (bytesAvailable > 0)
{
    byte[] data = new byte[bytesAvailable];
    in.read(data);
}
```

When you have finished reading or writing to a stream, close it by calling the `close` method. This call frees up operating system resources that are in limited supply. If an application opens too many streams without closing them, system resources can become depleted. Closing an output stream also flushes the buffer used for the output stream: any characters that were temporarily placed in a buffer so that they could be delivered as a larger packet are sent off. In particular, if you do not close a

file, the last packet of bytes might never be delivered. You can also manually flush the output with the `flush` method.

Even if a stream class provides concrete methods to work with the raw `read` and `write` functions, application programmers rarely use them. The data that you are interested in probably contain numbers, strings, and objects, not raw bytes.

Java gives you many stream classes derived from the basic `InputStream` and `OutputStream` classes that let you work with data in the forms that you usually use rather than at the byte level.



java.io.InputStream 1.0

- `abstract int read()`

reads a byte of data and returns the byte read. The `read` method returns a -1 at the end of the stream.

- `int read(byte[] b)`

reads into an array of bytes and returns the actual number of bytes read, or -1 at the end of the stream. The `read` method reads at most `b.length` bytes.

- `int read(byte[] b, int off, int len)`

reads into an array of bytes. The `read` method returns the actual number of bytes read, or -1 at the end of the stream.

Parameters: `b` The array into which the data is read

`off` The offset into `b` where the first bytes should be placed

`len` The maximum number of bytes to read

- `long skip(long n)`

skips `n` bytes in the input stream. Returns the actual number of bytes skipped (which may be less than `n` if the end of the stream was encountered).

- `int available()`

returns the number of bytes available without blocking. (Recall that blocking means that the current thread loses its turn.)

- `void close()`

closes the input stream.

- `void mark(int readlimit)`

puts a marker at the current position in the input stream. (Not all streams support this feature.) If more than `readlimit` bytes have been read from the input stream, then the stream is allowed to forget the marker.

- `void reset()`

returns to the last marker. Subsequent calls to `read` reread the bytes. If there is no current marker, then the stream is not reset.

- `boolean markSupported()`

returns `true` if the stream supports marking.

API**java.io.OutputStream 1.0**

- abstract void write(int n)
writes a byte of data.
- void write(byte[] b)
- void write(byte[] b, int off, int len)
writes all bytes or a range of bytes in the array b.

Parameters:

<code>b</code>	The array from which to write the data
<code>off</code>	The offset into <code>b</code> to the first byte that will be written
<code>len</code>	The number of bytes to write

- void close()
flushes and closes the output stream.
- void flush()
flushes the output stream; that is, sends any buffered data to its destination.

The Complete Stream Zoo

Unlike C, which gets by just fine with a single type `FILE*`, Java has a whole zoo of more than 60 (!) different stream types (see [Figures 1-1](#) and [1-2](#)).

Figure 1-1. Input and output stream hierarchy

[\[View full size image\]](#)

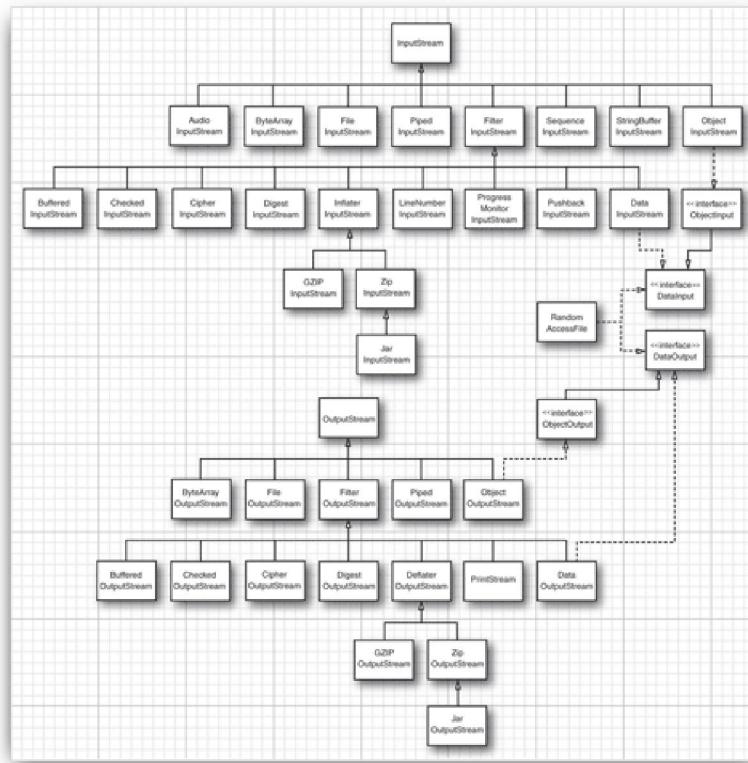
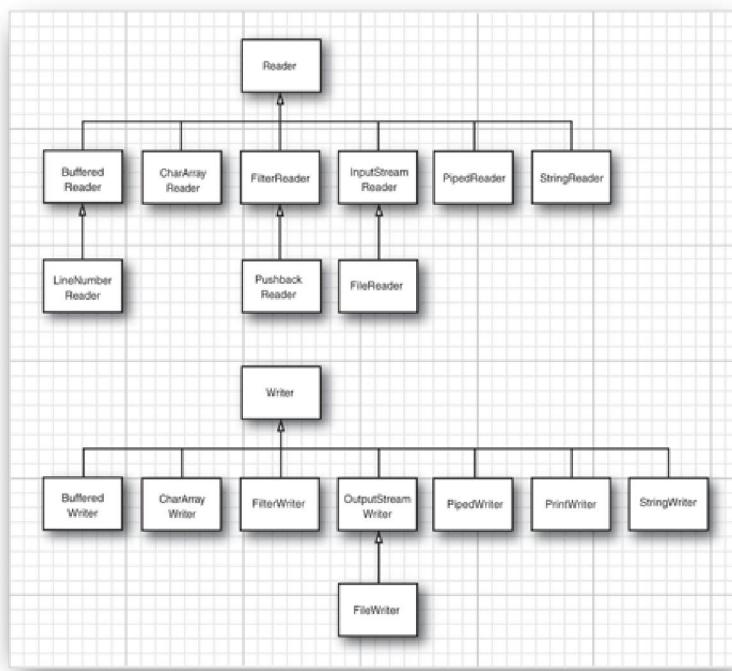


Figure 1-2. Reader and writer hierarchy[\[View full size image\]](#)

Let us divide the animals in the stream class zoo by how they are used. There are separate hierarchies for classes that process bytes and characters. As you saw, the `InputStream` and `OutputStream` classes let you read and write individual bytes and arrays of bytes. These classes form the basis of the hierarchy shown in [Figure 1-1](#). To read and write strings and numbers, you need more capable subclasses. For example, `DataInputStream` and `DataOutputStream` let you read and write all the primitive Java types in binary format. Finally, there are streams that do useful stuff; for example, the `ZipInputStream` and `ZipOutputStream` that let you read and write files in the familiar ZIP compression format.

For Unicode text, on the other hand, you use subclasses of the abstract classes `Reader` and `Writer` (see [Figure 1-2](#)). The basic methods of the `Reader` and `Writer` classes are similar to the ones for `InputStream` and `OutputStream`.

```
abstract int read()
abstract void write(int c)
```

The `read` method returns either a Unicode code unit (as an integer between 0 and 65535) or -1 when you have reached the end of the file. The `write` method is called with a Unicode code unit. (See Volume I, Chapter 3 for a discussion of Unicode code units.)

Java SE 5.0 introduced four additional interfaces: `Closeable`, `Flushable`, `Readable`, and `Appendable` (see [Figure 1-3](#)). The first two interfaces are very simple, with methods

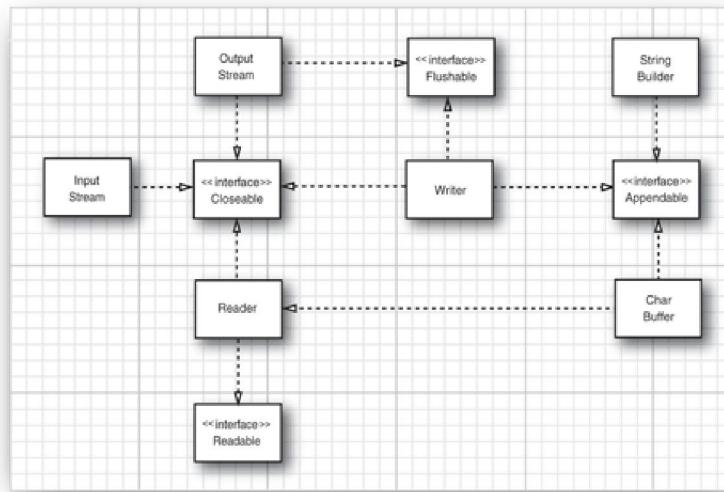
```
void close() throws IOException
```

and

```
void flush()
```

respectively. The classes `InputStream`, `OutputStream`, `Reader`, and `Writer` all implement the `Closeable` interface. `OutputStream` and `Writer` implement the `Flushable` interface.

Figure 1-3. The `Closeable`, `Flushable`, `Readable`, and `Appendable` interfaces[\[View full size image\]](#)



The `Readable` interface has a single method

```
int read(CharBuffer cb)
```

The `CharBuffer` class has methods for sequential and random read/write access. It represents an in-memory buffer or a memory-mapped file. (See "[The Buffer Data Structure](#)" on page [72](#) for details.)

The `Appendable` interface has two methods for appending single characters and character sequences:

```
Appendable append(char c)
Appendable append(CharSequence s)
```

The `CharSequence` interface describes basic properties of a sequence of `char` values. It is implemented by `String`, `CharBuffer`, `StringBuilder`, and `StringBuffer`.

Of the stream zoo classes, only `Writer` implements `Appendable`.



java.io.Closeable 5.0

- `void close()`
closes this `Closeable`. This method may throw an `IOException`.



java.io.Flushable 5.0

- `void flush()`
flushes this `Flushable`.



java.lang.Readable 5.0

- `int read(CharBuffer cb)`
attempts to read as many `char` values into `cb` as it can hold. Returns the number of values read, or -1 if no further values are available from this `Readable`.



java.lang.Appendable 5.0

- Appendable append(char c)
- Appendable append(CharSequence cs)

appends the given code unit, or all code units in the given sequence, to this Appendable; returns this.



java.lang.CharSequence 1.4

- char charAt(int index)
returns the code unit at the given index.
- int length()
returns the number of code units in this sequence.
- CharSequence subSequence(int startIndex, int endIndex)
returns a CharSequence consisting of the code units stored at index startIndex to endIndex - 1.
- String toString()
returns a string consisting of the code units of this sequence.

Combining Stream Filters

`FileInputStream` and `FileOutputStream` give you input and output streams attached to a disk file. You give the file name or full path name of the file in the constructor. For example,

```
FileInputStream fin = new FileInputStream("employee.dat");
```

looks in the user directory for a file named "employee.dat".

Tip



Because all the classes in `java.io` interpret relative path names as starting with the user's working directory, you may want to know this directory. You can get at this information by a call to `System.getProperty("user.dir")`.

Like the abstract `InputStream` and `OutputStream` classes, these classes support only reading and writing on the byte level. That is, we can only read bytes and byte arrays from the object `fin`.

```
byte b = (byte) fin.read();
```

As you will see in the next section, if we just had a `DataInputStream`, then we could read numeric types:

```
DataInputStream din = . . .;
double s = din.readDouble();
```

But just as the `FileInputStream` has no methods to read numeric types, the `DataInputStream` has no method to get data from a file.

Java uses a clever mechanism to separate two kinds of responsibilities. Some streams (such as the `FileInputStream` and the input stream returned by the `openStream` method of the `URL` class) can retrieve bytes from files and other more exotic

locations. Other streams (such as the `DataInputStream` and the `PrintWriter`) can assemble bytes into more useful data types. The Java programmer has to combine the two. For example, to be able to read numbers from a file, first create a `FileInputStream` and then pass it to the constructor of a `DataInputStream`.

```
FileInputStream fin = new FileInputStream("employee.dat");
DataInputStream din = new DataInputStream(fin);
double s = din.readDouble();
```

If you look at [Figure 1-1](#) again, you can see the classes `FilterInputStream` and `FilterOutputStream`. The subclasses of these files are used to add capabilities to raw byte streams.

You can add multiple capabilities by nesting the filters. For example, by default, streams are not buffered. That is, every call to `read` asks the operating system to dole out yet another byte. It is more efficient to request blocks of data instead and put them in a buffer. If you want buffering and the data input methods for a file, you need to use the following rather monstrous sequence of constructors:

```
DataInputStream din = new DataInputStream(
    new BufferedInputStream(
        new FileInputStream("employee.dat")));
```

Notice that we put the `DataInputStream` last in the chain of constructors because we want to use the `DataInputStream` methods, and we want them to use the buffered `read` method.

Sometimes you'll need to keep track of the intermediate streams when chaining them together. For example, when reading input, you often need to peek at the next byte to see if it is the value that you expect. Java provides the `PushbackInputStream` for this purpose.

```
PushbackInputStream pbin = new PushbackInputStream(
    new BufferedInputStream(
        new FileInputStream("employee.dat")));
```

Now you can speculatively read the next byte

```
int b = pbin.read();
```

and throw it back if it isn't what you wanted.

```
if (b != '<') pbin.unread(b);
```

But reading and unreading are the only methods that apply to the pushback input stream. If you want to look ahead and also read numbers, then you need both a pushback input stream and a data input stream reference.

```
DataInputStream din = new DataInputStream(
    pbin = new PushbackInputStream(
        new BufferedInputStream(
            new FileInputStream("employee.dat"))));
```

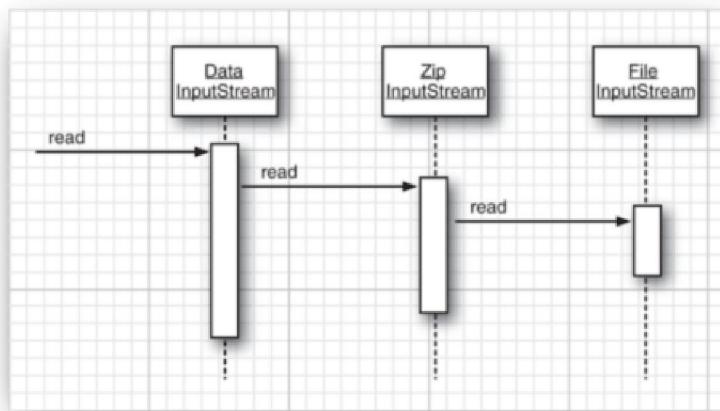
Of course, in the stream libraries of other programming languages, niceties such as buffering and lookahead are automatically taken care of, so it is a bit of a hassle in Java that one has to resort to combining stream filters in these cases. But the ability to mix and match filter classes to construct truly useful sequences of streams does give you an immense amount of flexibility. For example, you can read numbers from a compressed ZIP file by using the following sequence of streams (see [Figure 1-4](#)):

Code View:

```
ZipInputStream zin = new ZipInputStream(new FileInputStream("employee.zip"));
DataInputStream din = new DataInputStream(zin);
```

Figure 1-4. A sequence of filtered streams

[\[View full size image\]](#)



(See "[ZIP Archives](#)" on page [32](#) for more on Java's ability to handle ZIP files.)



java.io.FileInputStream 1.0

- `FileInputStream(String name)`
- `FileInputStream(File file)`

creates a new file input stream, using the file whose path name is specified by the `name` string or the `file` object. (The `File` class is described at the end of this chapter.) Path names that are not absolute are resolved relative to the working directory that was set when the VM started.



java.io.FileOutputStream 1.0

- `FileOutputStream(String name)`
- `FileOutputStream(String name, boolean append)`
- `FileOutputStream(File file)`
- `FileOutputStream(File file, boolean append)`

creates a new file output stream specified by the `name` string or the `file` object. (The `File` class is described at the end of this chapter.) If the `append` parameter is `true`, then data are added at the end of the file. An existing file with the same name will not be deleted. Otherwise, this method deletes any existing file with the same name.



java.io.BufferedInputStream 1.0

- `BufferedInputStream(InputStream in)`

creates a buffered stream. A buffered input stream reads characters from a stream without causing a device access every time. When the buffer is empty, a new block of data is read into the buffer.



java.io.BufferedOutputStream 1.0

- `BufferedOutputStream(OutputStream out)`

creates a buffered stream. A buffered output stream collects characters to be written without causing a device access every time. When the buffer fills up or when the stream is flushed, the data are written.

API

java.io.PushbackInputStream 1.0

- `PushbackInputStream(InputStream in)`
- `PushbackInputStream(InputStream in, int size)`

constructs a stream with one-byte lookahead or a pushback buffer of specified size.

- `void unread(int b)`

pushes back a byte, which is retrieved again by the next call to read.

Parameters: `b` The byte to be read again



Text Input and Output

When saving data, you have the choice between binary and text format. For example, if the integer 1234 is saved in binary, it is written as the sequence of bytes 00 00 04 D2 (in hexadecimal notation). In text format, it is saved as the string "1234". Although binary I/O is fast and efficient, it is not easily readable by humans. We first discuss text I/O and cover binary I/O in the section "[Reading and Writing Binary Data](#)" on page [23](#).

When saving text strings, you need to consider the character encoding. In the UTF-16 encoding, the string "1234" is encoded as 00 31 00 32 00 33 00 34 (in hex). However, many programs expect that text files are encoded in a different encoding. In ISO 8859-1, the encoding most commonly used in the United States and Western Europe, the string would be written as 31 32 33 34, without the zero bytes.

The `OutputStreamWriter` class turns a stream of Unicode characters into a stream of bytes, using a chosen character encoding. Conversely, the `InputStreamReader` class turns an input stream that contains bytes (specifying characters in some character encoding) into a reader that emits Unicode characters.

For example, here is how you make an input reader that reads keystrokes from the console and converts them to Unicode:

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

This input stream reader assumes the default character encoding used by the host system, such as the ISO 8859-1 encoding in Western Europe. You can choose a different encoding by specifying it in the constructor for the `InputStreamReader`, for example,

Code View:

```
InputStreamReader in = new InputStreamReader(new FileInputStream("kremlin.dat"), "ISO8859_5");
```

See "[Character Sets](#)" on page [19](#) for more information on character encodings.

Because it is so common to attach a reader or writer to a file, a pair of convenience classes, `FileReader` and `FileWriter`, is provided for this purpose. For example, the writer definition

```
FileWriter out = new FileWriter("output.txt");
```

is equivalent to

```
FileWriter out = new FileWriter(new FileOutputStream("output.txt"));
```

How to Write Text Output

For text output, you want to use a `PrintWriter`. That class has methods to print strings and numbers in text format. There is even a convenience constructor to link a `PrintWriter` with a `FileWriter`. The statement

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

is equivalent to

```
PrintWriter out = new PrintWriter(new FileWriter("employee.txt"));
```

To write to a print writer, you use the same `print`, `println`, and `printf` methods that you used with `System.out`. You can use these methods to print numbers (`int`, `short`, `long`, `float`, `double`), characters, `boolean` values, strings, and objects.

For example, consider this code:

```
String name = "Harry Hacker";
double salary = 75000;
out.print(name);
out.print(' ');
out.println(salary);
```

This writes the characters

```
Harry Hacker 75000.0
```

to the writer `out`. The characters are then converted to bytes and end up in the file `employee.txt`.

The `println` method adds the correct end-of-line character for the target system ("`\r\n`" on Windows, "`\n`" on UNIX) to the line. This is the string obtained by the call `System.getProperty("line.separator")`.

If the writer is set to autoflush mode, then all characters in the buffer are sent to their destination whenever `println` is called. (Print writers are always buffered.) By default, autoflushing is not enabled. You can enable or disable autoflushing by using the `PrintWriter(Writer out, boolean autoFlush)` constructor:

Code View:

```
PrintWriter out = new PrintWriter(new FileWriter("employee.txt"), true); // autoflush
```

The `print` methods don't throw exceptions. You can call the `checkError` method to see if something went wrong with the stream.

Note



Java veterans might wonder whatever happened to the `PrintStream` class and to `System.out`. In Java 1.0, the `PrintStream` class simply truncated all Unicode characters to ASCII characters by dropping the top byte. Clearly, that was not a clean or portable approach, and it was fixed with the introduction of readers and writers in Java 1.1. For compatibility with existing code, `System.in`, `System.out`, and `System.err` are still streams, not readers and writers. But now the `PrintStream` class internally converts Unicode characters to the default host encoding in the same way as the `PrintWriter` does. Objects of type `PrintStream` act exactly like print writers when you use the `print` and `println` methods, but unlike print writers, they allow you to output raw bytes with the `write(int)` and `write(byte[])` methods.

API

java.io.PrintWriter 1.1

- `PrintWriter(Writer out)`
- `PrintWriter(Writer out, boolean autoFlush)`

creates a new `PrintWriter`.

Parameters: `out` A character-output writer

`autoflush` If `true`, the `println` methods will flush the output buffer (default: `false`)

- `PrintWriter(OutputStream out)`
- `PrintWriter(OutputStream out, boolean autoflush)`

creates a new `PrintWriter` from an existing `OutputStream` by creating the necessary intermediate `OutputStreamWriter`.

- `PrintWriter(String filename)`
- `PrintWriter(File file)`

creates a new `PrintWriter` that writes to the given file by creating the necessary intermediate `FileWriter`.

- `void print(Object obj)`
- prints an object by printing the string resulting from `toString`.

Parameters: `obj` The object to be printed

- `void print(String s)`

prints a Unicode string.

- `void println(String s)`

prints a string followed by a line terminator. Flushes the stream if the stream is in `autoflush` mode.

- `void print(char[] s)`

prints all Unicode characters in the given array.

- `void print(char c)`

prints a Unicode character.

- `void print(int i)`

- `void print(long l)`

- `void print(float f)`

- `void print(double d)`

- `void print(boolean b)`

prints the given value in text format.

- `void printf(String format, Object... args)`

prints the given values, as specified by the format string. See Volume I, Chapter 3 for the specification of the format string.

- `boolean checkError()`

returns `true` if a formatting or output error occurred. Once the stream has encountered an error, it is tainted and all calls to `checkError` return `true`.

How to Read Text Input

As you know:

- To write data in binary format, you use a `DataOutputStream`.
- To write in text format, you use a `PrintWriter`.

Therefore, you might expect that there is an analog to the `DataInputStream` that lets you read data in text format. The closest analog is the `Scanner` class that we used extensively in Volume I. However, before Java SE 5.0, the only game in town for processing text input was the `BufferedReader` class—it has a method, `readLine`, that lets you read a line of text. You need to combine a buffered reader with an input source.

```
BufferedReader in = new BufferedReader(new FileReader("employee.txt"));
```

The `readLine` method returns `null` when no more input is available. A typical input loop, therefore, looks like this:

```
String line;
while ((line = in.readLine()) != null)
{
    do something with line
}
```

However, a `BufferedReader` has no methods for reading numbers. We suggest that you use a `Scanner` for reading text input.

Saving Objects in Text Format

In this section, we walk you through an example program that stores an array of `Employee` records in a text file. Each record is stored in a separate line. Instance fields are separated from each other by delimiters. We use a vertical bar (|) as our delimiter. (A colon (:)) is another popular choice. Part of the fun is that everyone uses a different delimiter.) Naturally, we punt on the issue of what might happen if a | actually occurred in one of the strings we save.

Here is a sample set of records:

```
Harry Hacker|35500|1989|10|1
Carl Cracker|75000|1987|12|15
Tony Tester|38000|1990|3|15
```

Writing records is simple. Because we write to a text file, we use the `PrintWriter` class. We simply write all fields, followed by either a | or, for the last field, a \n. This work is done in the following `writeData` method that we add to our `Employee` class.

```
public void writeData(PrintWriter out) throws IOException
{
    GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(hireDay);
    out.println(name + "|"
        + salary + "|"
        + calendar.get(Calendar.YEAR) + "|"
        + (calendar.get(Calendar.MONTH) + 1) + "|"
        + calendar.get(Calendar.DAY_OF_MONTH));
}
```

To read records, we read in a line at a time and separate the fields. We use a scanner to read each line and then split the line into tokens with the `String.split` method.

```
public void readData(Scanner in)
{
    String line = in.nextLine();
    String[] tokens = line.split("\\|");
    name = tokens[0];
    salary = Double.parseDouble(tokens[1]);
    int y = Integer.parseInt(tokens[2]);
    int m = Integer.parseInt(tokens[3]);
    int d = Integer.parseInt(tokens[4]);
    GregorianCalendar calendar = new GregorianCalendar(y, m - 1, d);
    hireDay = calendar.getTime();
}
```

The parameter of the `split` method is a regular expression describing the separator. We discuss regular expressions in more detail at the end of this chapter. As it happens, the vertical bar character has a special meaning in regular expressions, so it needs to be escaped with a \ character. That character needs to be escaped by another \, yielding the "\\|" expression.

The complete program is in [Listing 1-1](#). The static method

```
void writeData(Employee[] e, PrintWriter out)
```

first writes the length of the array, then writes each record. The static method

```
Employee[] readData(BufferedReader in)
```

first reads in the length of the array, then reads in each record. This turns out to be a bit tricky:

```
int n = in.nextInt();
in.nextLine(); // consume newline
Employee[] employees = new Employee[n];
for (int i = 0; i < n; i++)
{
    employees[i] = new Employee();
    employees[i].readData(in);
}
```

The call to `nextInt` reads the array length but not the trailing newline character. We must consume the newline so that the `readData` method can get the next input line when it calls the `nextLine` method.

Listing 1-1. TextFileTest.java

Code View:

```
1. import java.io.*;
2. import java.util.*;
3.
4. /**
5.  * @version 1.12 2007-06-22
6.  * @author Cay Horstmann
7. */
8. public class TextFileTest
9. {
10.    public static void main(String[] args)
11.    {
12.        Employee[] staff = new Employee[3];
13.
14.        staff[0] = new Employee("Carl Cracker", 75000, 1987, 12, 15);
15.        staff[1] = new Employee("Harry Hacker", 50000, 1989, 10, 1);
16.        staff[2] = new Employee("Tony Tester", 40000, 1990, 3, 15);
17.
18.        try
19.        {
20.            // save all employee records to the file employee.dat
21.            PrintWriter out = new PrintWriter("employee.dat");
22.            writeData(staff, out);
23.            out.close();
24.        }
```

```
24.         // retrieve all records into a new array
25.         Scanner in = new Scanner(new FileReader("employee.dat"));
26.         Employee[] newStaff = readData(in);
27.         in.close();
28.
29.         // print the newly read employee records
30.         for (Employee e : newStaff)
31.             System.out.println(e);
32.     }
33. }
34. catch (IOException exception)
35. {
36.     exception.printStackTrace();
37. }
38. }
39.
40. /**
41. * Writes all employees in an array to a print writer
42. * @param employees an array of employees
43. * @param out a print writer
44. */
45. private static void writeData(Employee[] employees, PrintWriter out) throws IOException
46. {
47.     // write number of employees
48.     out.println(employees.length);
49.
50.     for (Employee e : employees)
51.         e.writeData(out);
52. }
53. /**
54. * Reads an array of employees from a scanner
55. * @param in the scanner
56. * @return the array of employees
57. */
58. private static Employee[] readData(Scanner in)
59. {
60.     // retrieve the array size
61.     int n = in.nextInt();
62.     in.nextLine(); // consume newline
63.
64.     Employee[] employees = new Employee[n];
65.     for (int i = 0; i < n; i++)
66.     {
67.         employees[i] = new Employee();
68.         employees[i].readData(in);
69.     }
70.     return employees;
71. }
72. }
73.
74. class Employee
75. {
76.     public Employee()
77.     {
78.     }
79.
80.     public Employee(String n, double s, int year, int month, int day)
81.     {
82.         name = n;
83.         salary = s;
84.         GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
85.         hireDay = calendar.getTime();
86.     }
87.
88.     public String getName()
89.     {
90.         return name;
91.     }
92.
93.     public double getSalary()
94.     {
95.         return salary;
96.     }
97.
98.     public Date getHireDay()
99.     {
100.        return hireDay;
101.    }
102.
103.    public void raiseSalary(double byPercent)
```

```
104.     {
105.         double raise = salary * byPercent / 100;
106.         salary += raise;
107.     }
108.
109.    public String toString()
110.    {
111.        return getClass().getName() + "[name=" + name + ",salary=" + salary + ",hireDay="
112.            + hireDay + "]";
113.    }
114.
115.    /**
116.     * Writes employee data to a print writer
117.     * @param out the print writer
118.     */
119.    public void writeData(PrintWriter out)
120.    {
121.        GregorianCalendar calendar = new GregorianCalendar();
122.        calendar.setTime(hireDay);
123.        out.println(name + "|" + salary + "|" + calendar.get(Calendar.YEAR) + "|"
124.            + (calendar.get(Calendar.MONTH) + 1) + "|" + calendar.get(Calendar.DAY_OF_MONTH));
125.    }
126.
127.    /**
128.     * Reads employee data from a buffered reader
129.     * @param in the scanner
130.     */
131.    public void readData(Scanner in)
132.    {
133.        String line = in.nextLine();
134.        String[] tokens = line.split("\\|");
135.        name = tokens[0];
136.        salary = Double.parseDouble(tokens[1]);
137.        int y = Integer.parseInt(tokens[2]);
138.        int m = Integer.parseInt(tokens[3]);
139.        int d = Integer.parseInt(tokens[4]);
140.        GregorianCalendar calendar = new GregorianCalendar(y, m - 1, d);
141.        hireDay = calendar.getTime();
142.    }
143.
144.    private String name;
145.    private double salary;
146.    private Date hireDay;
147. }
```

Character Sets

In the past, international character sets have been handled rather unsystematically throughout the Java library. The `java.nio` package—introduced in Java SE 1.4—unifies character set conversion with the introduction of the `Charset` class. (Note that the `s` is lower case.)

A character set maps between sequences of two-byte Unicode code units and byte sequences used in a local character encoding. One of the most popular character encodings is ISO-8859-1, a single-byte encoding of the first 256 Unicode characters. Gaining in importance is ISO-8859-15, which replaces some of the less useful characters of ISO-8859-1 with accented letters used in French and Finnish, and, more important, replaces the "international currency" character ☰ with the Euro symbol (€) in code point `0xA4`. Other examples for character encodings are the variable-byte encodings commonly used for Japanese and Chinese.

The `Charset` class uses the character set names standardized in the IANA Character Set Registry (<http://www.iana.org/assignments/character-sets>). These names differ slightly from those used in previous versions. For example, the "official" name of ISO-8859-1 is now "`ISO-8859-1`" and no longer "`ISO8859_1`", which was the preferred name up to Java SE 1.3.

Note



An excellent reference for the "ISO 8859 alphabet soup" is <http://czyborra.com/charsets/iso8859.html>.

You obtain a `Charset` by calling the static `forName` method with either the official name or one of its aliases:

```
Charset cset = Charset.forName("ISO-8859-1");
```

Character set names are case insensitive.

For compatibility with other naming conventions, each character set can have a number of aliases. For example, ISO-8859-1 has aliases

```
ISO8859-1
ISO_8859_1
ISO8859_1
ISO_8859-1
ISO_8859-1:1987
8859_1
latin1
11
csISOLatin1
iso-ir-100
cp819
IBM819
IBM-819
819
```

The `aliases` method returns a `Set` object of the aliases. Here is the code to iterate through the aliases:

```
Set<String> aliases = cset.aliases();
for (String alias : aliases)
    System.out.println(alias);
```

To find out which character sets are available in a particular implementation, call the static `availableCharsets` method. Use this code to find out the names of all available character sets:

```
Map<String, Charset> charsets = Charset.availableCharsets();
for (String name : charsets.keySet())
    System.out.println(name);
```

[Table 1-1](#) lists the character encodings that every Java implementation is required to have. [Table 1-2](#) lists the encoding schemes that the Java Development Kit (JDK) installs by default. The character sets in [Table 1-3](#) are installed only on operating systems that use non-European languages.

Table 1-1. Required Character Encodings

Charset Standard Name	Legacy Name	Description
US-ASCII	ASCII	American Standard Code for Information Exchange
ISO-8859-1	ISO8859_1	ISO 8859-1, Latin alphabet No. 1
UTF-8	UTF8	Eight-bit Unicode Transformation Format
UTF-16	UTF-16	Sixteen-bit Unicode Transformation Format, byte order specified by an optional initial byte-order mark
UTF-16BE	UnicodeBigUnmarked	Sixteen-bit Unicode Transformation Format, big-endian byte order
UTF-16LE	UnicodeLittleUnmarked	Sixteen-bit Unicode Transformation Format, little-endian byte order

Table 1-2. Basic Character Encodings

Charset Standard Name	Legacy Name	Description
-----------------------	-------------	-------------

ISO8859-2	ISO8859_2	ISO 8859-2, Latin alphabet No. 2
ISO8859-4	ISO8859_4	ISO 8859-4, Latin alphabet No. 4
ISO8859-5	ISO8859_5	ISO 8859-5, Latin/Cyrillic alphabet
ISO8859-7	ISO8859_7	ISO 8859-7, Latin/Greek alphabet
ISO8859-9	ISO8859_9	ISO 8859-9, Latin alphabet No. 5
ISO8859-13	ISO8859_13	ISO 8859-13, Latin alphabet No. 7
ISO8859-15	ISO8859_15	ISO 8859-15, Latin alphabet No. 9
windows-1250	Cp1250	Windows Eastern European
windows-1251	Cp1251	Windows Cyrillic
windows-1252	Cp1252	Windows Latin-1
windows-1253	Cp1253	Windows Greek
windows-1254	Cp1254	Windows Turkish
windows-1257	Cp1257	Windows Baltic

Table 1-3. Extended Character Encodings

Charset	Standard Name	Legacy Name	Description
Big5	Big5	Big5	Big5, Traditional Chinese
Big5-HKSCS	Big5_HKSCS	Big5 with Hong Kong extensions, Traditional Chinese	
EUC-JP	EUC_JP	JIS X 0201, 0208, 0212, EUC encoding, Japanese	
EUC-KR	EUC_KR	KS C 5601, EUC encoding, Korean	
GB18030	GB18030	Simplified Chinese, PRC Standard	
GBK	GBK	GBK, Simplified Chinese	
ISCII91	ISCII91	ISCII91 encoding of Indic scripts	
ISO-2022-JP	ISO2022JP	JIS X 0201, 0208 in ISO 2022 form, Japanese	
ISO-2022-KR	ISO2022KR	ISO 2022 KR, Korean	
ISO8859-3	ISO8859_3	ISO 8859-3, Latin alphabet No. 3	
ISO8859-6	ISO8859_6	ISO 8859-6, Latin/Arabic alphabet	
ISO8859-8	ISO8859_8	ISO 8859-8, Latin/Hebrew alphabet	
Shift_JIS	SJIS	Shift-JIS, Japanese	
TIS-620	TIS620	TIS620, Thai	
windows-1255	Cp1255	Windows Hebrew	
windows-1256	Cp1256	Windows Arabic	
windows-1258	Cp1258	Windows Vietnamese	
windows-31j	MS932	Windows Japanese	
x-EUC-CN	EUC_CN	GB2312, EUC encoding, Simplified Chinese	
x-EUC-JP-LINUX	EUC_JP_LINUX	JIS X 0201, 0208, EUC encoding, Japanese	
x-EUC-TW	EUC_TW	CNS11643 (Plane 1-3), EUC encoding, Traditional Chinese	
x-MS950-HKSCS	MS950_HKSCS	Windows Traditional Chinese with Hong Kong extensions	
x-mswin-936	MS936	Windows Simplified Chinese	
x-windows-949	MS949	Windows Korean	

Local encoding schemes cannot represent all Unicode characters. If a character cannot be represented, it is transformed to a ?.

Once you have a character set, you can use it to convert between Unicode strings and encoded byte sequences. Here is how you encode a Unicode string:

```
String str = . . .;
ByteBuffer buffer = cset.encode(str);
byte[] bytes = buffer.array();
```

Conversely, to decode a byte sequence, you need a byte buffer. Use the static `wrap` method of the `ByteBuffer` array to turn a byte array into a byte buffer. The result of the `decode` method is a `CharBuffer`. Call its `toString` method to get a string.

```
byte[] bytes = . . .;
ByteBuffer bbuf = ByteBuffer.wrap(bytes, offset, length);
CharBuffer cbuf = cset.decode(bbuf);
String str = cbuf.toString();
```

API**java.nio.charset.Charset 1.4**

- `static SortedMap availableCharsets()`

gets all available character sets for this virtual machine. Returns a map whose keys are character set names and whose values are character sets.

- `static Charset forName(String name)`

gets a character set for the given name.

- `Set aliases()`

returns the set of alias names for this character set.

- `ByteBuffer encode(String str)`

encodes the given string into a sequence of bytes.

- `CharBuffer decode(ByteBuffer buffer)`

decodes the given byte sequence. Unrecognized inputs are converted to the Unicode "replacement character" ('\uFFFD').

API**java.nio.ByteBuffer 1.4**

- `byte[] array()`

returns the array of bytes that this buffer manages.

- `static ByteBuffer wrap(byte[] bytes)`

- `static ByteBuffer wrap(byte[] bytes, int offset, int length)`

returns a byte buffer that manages the given array of bytes or the given range.

API**java.nio.CharBuffer**

- `char[] array()`
returns the array of code units that this buffer manages.
- `char charAt(int index)`
returns the code unit at the given index.
- `String toString()`
returns a string consisting of the code units that this buffer manages.



Reading and Writing Binary Data

The `DataOutput` interface defines the following methods for writing a number, character, `boolean` value, or string in binary format:

```
writeChars
writeByte
writeInt
writeShort
writeLong
writeFloat
writeDouble
writeChar
writeBoolean
writeUTF
```

For example, `.writeInt` always writes an integer as a 4-byte binary quantity regardless of the number of digits, and `writeDouble` always writes a `double` as an 8-byte binary quantity. The resulting output is not humanly readable, but the space needed will be the same for each value of a given type and reading it back in will be faster than parsing text.

Note



There are two different methods of storing integers and floating-point numbers in memory, depending on the platform you are using. Suppose, for example, you are working with a 4-byte `int`, say the decimal number 1234, or `4D2` in hexadecimal ($1234 = 4 \times 256 + 13 \times 16 + 2$). This can be stored in such a way that the first of the 4 bytes in memory holds the most significant byte (MSB) of the value: `00 00 04 D2`. This is the so-called big-endian method. Or we can start with the least significant byte (LSB) first: `D2 04 00 00`. This is called, naturally enough, the little-endian method. For example, the SPARC uses big-endian; the Pentium, little-endian. This can lead to problems. When a C or C++ file is saved, the data are saved exactly as the processor stores them. That makes it challenging to move even the simplest data files from one platform to another. In Java, all values are written in the big-endian fashion, regardless of the processor. That makes Java data files platform independent.

The `writeUTF` method writes string data by using a modified version of 8-bit Unicode Transformation Format. Instead of simply using the standard UTF-8 encoding (which is shown in [Table 1-4](#)), character strings are first represented in UTF-16 (see [Table 1-5](#)) and then the result is encoded using the UTF-8 rules. The modified encoding is different for characters with code higher than `0xFFFF`. It is used for backward compatibility with virtual machines that were built when Unicode had not yet grown beyond 16 bits.

Table 1-4. UTF-8 Encoding

Character Range	Encoding
0...7F	<code>0a₆a₅a₄a₃a₂a₁a₀</code>
80...7FF	<code>110a₁₀a₉a₈a₇a₆ 10a₅a₄a₃a₂a₁a₀</code>
800...FFFF	<code>1110a₁₅a₁₄a₁₃a₁₂ 10a₁₁a₁₀a₉a₈a₇a₆ 10a₅a₄a₃a₂a₁a₀</code>
10000...10FFFF	<code>11110a₂₀a₁₉a₁₈ 10a₁₇a₁₆a₁₅a₁₄a₁₃a₁₂ 10a₁₁a₁₀a₉a₈a₇a₆</code>

10a₅a₄a₃a₂a₁a₀

Table 1-5. UTF-16 Encoding

Character Range	Encoding
0...FFFF	a ₁₅ a ₁₄ a ₁₃ a ₁₂ a ₁₁ a ₁₀ a ₉ a ₈ a ₇ a ₆ a ₅ a ₄ a ₃ a ₂ a ₁ a ₀
10000...10FFFF	110110b ₁₉ b ₁₈ b ₁₇ b ₁₆ a ₁₅ a ₁₄ a ₁₃ a ₁₂ a ₁₁ a ₁₀ 110111a ₉ a ₈ a ₇ a ₆ a ₅ a ₄ a ₃ a ₂ a ₁ a ₀ where b ₁₉ b ₁₈ b ₁₇ b ₁₆ = a ₂₀ a ₁₉ a ₁₈ a ₁₇ a ₁₆ -1

Because nobody else uses this modification of UTF-8, you should only use the `writeUTF` method to write strings that are intended for a Java virtual machine; for example, if you write a program that generates bytecodes. Use the `writeChars` method for other purposes.

Note

- See RFC 2279 (<http://ietf.org/rfc/rfc2279.txt>) and RFC 2781 (<http://ietf.org/rfc/rfc2781.txt>) for definitions of UTF-8 and UTF-16.

To read the data back in, use the following methods, defined in the `DataInput` interface:

```
readInt
readShort
readLong
readFloat
readDouble
readChar
readBoolean
readUTF
```

The `DataInputStream` class implements the `DataInput` interface. To read binary data from a file, you combine a `DataInputStream` with a source of bytes such as a `FileInputStream`:

Code View:

```
DataInputStream in = new DataInputStream(new FileInputStream("employee.dat"));
```

Similarly, to write binary data, you use the `DataOutputStream` class that implements the `DataOutput` interface:

Code View:

```
DataOutputStream out = new DataOutputStream(new FileOutputStream("employee.dat"));
```



java.io.DataInput 1.0

- boolean `readBoolean()`
- byte `readByte()`
- char `readChar()`
- double `readDouble()`
- float `readFloat()`

- int readInt()
 - long readLong()
 - short readShort()
- reads in a value of the given type.
- void readFully(byte[] b)
- reads bytes into the array `b`, blocking until all bytes are read.
- Parameters: `b` The buffer into which the data are read
- void readFully(byte[] b, int off, int len)
- reads bytes into the array `b`, blocking until all bytes are read.
- Parameters: `b` The buffer into which the data are read
 `off` The start offset of the data
 `len` The maximum number of bytes to read
- String readUTF()
- reads a string of characters in "modified UTF-8" format.
- int skipBytes(int n)
- skips `n` bytes, blocking until all bytes are skipped.
- Parameters: `n` The number of bytes to be skipped



java.io.DataOutput 1.0

- void writeBoolean(boolean b)
 - void writeByte(int b)
 - void writeChar(int c)
 - void writeDouble(double d)
 - void writeFloat(float f)
 - void writeInt(int i)
 - void writeLong(long l)
 - void writeShort(int s)
- writes a value of the given type.
- void writeChars(String s)
- writes all characters in the string.

- `void writeUTF(String s)`
writes a string of characters in "modified UTF-8" format.

Random-Access Files

The `RandomAccessFile` class lets you find or write data anywhere in a file. Disk files are random access, but streams of data from a network are not. You open a random-access file either for reading only or for both reading and writing. You specify the option by using the string "`r`" (for read access) or "`rw`" (for read/write access) as the second argument in the constructor.

```
RandomAccessFile in = new RandomAccessFile("employee.dat", "r");
RandomAccessFile inOut = new RandomAccessFile("employee.dat", "rw");
```

When you open an existing file as a `RandomAccessFile`, it does not get deleted.

A random-access file has a file pointer that indicates the position of the next byte that will be read or written. The `seek` method sets the file pointer to an arbitrary byte position within the file. The argument to `seek` is a `long` integer between zero and the length of the file in bytes.

The `getFilePointer` method returns the current position of the file pointer.

The `RandomAccessFile` class implements both the `DataInput` and `DataOutput` interfaces. To read and write from a random-access file, you use methods such as `readInt/writeInt` and `readChar/writeChar` that we discussed in the preceding section.

We now walk through an example program that stores employee records in a random access file. Each record will have the same size. This makes it easy to read an arbitrary record. Suppose you want to position the file pointer to the third record. Simply set the file pointer to the appropriate byte position and start reading.

```
long n = 3;
in.seek((n - 1) * RECORD_SIZE);
Employee e = new Employee();
e.readData(in);
```

If you want to modify the record and then save it back into the same location, remember to set the file pointer back to the beginning of the record:

```
in.seek((n - 1) * RECORD_SIZE);
e.writeData(out);
```

To determine the total number of bytes in a file, use the `length` method. The total number of records is the length divided by the size of each record.

```
long nbytes = in.length(); // length in bytes
int nrecords = (int) (nbytes / RECORD_SIZE);
```

Integers and floating-point values have a fixed size in binary format, but we have to work harder for strings. We provide two helper methods to write and read strings of a fixed size.

The `writeFixedString` writes the specified number of code units, starting at the beginning of the string. (If there are too few code units, the method pads the string, using zero values.)

```
public static void writeFixedString(String s, int size, DataOutput out)
    throws IOException
{
    for (int i = 0; i < size; i++)
    {
        char ch = 0;
        if (i < s.length()) ch = s.charAt(i);
        out.writeChar(ch);
    }
}
```

The `readFixedString` method reads characters from the input stream until it has consumed `size` code units or until it encounters a character with a zero value. Then, it skips past the remaining zero values in the input field. For added efficiency, this method uses the `StringBuilder` class to read in a string.

```
public static String readFixedString(int size, DataInput in)
    throws IOException
{
    StringBuilder b = new StringBuilder(size);
    int i = 0;
    boolean more = true;
    while (more && i < size)
    {
        char ch = in.readChar();
        i++;
        if (ch == 0) more = false;
        else b.append(ch);
    }
    in.skipBytes(2 * (size - i));
    return b.toString();
}
```

We placed the `writeFixedString` and `readFixedString` methods inside the `DataIO` helper class.

To write a fixed-size record, we simply write all fields in binary.

```
public void writeData(DataOutput out) throws IOException
{
    DataIO.writeFixedString(name, NAME_SIZE, out);
    out.writeDouble(salary);

    GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(hireDay);
    out.writeInt(calendar.get(Calendar.YEAR));
    out.writeInt(calendar.get(Calendar.MONTH) + 1);
    out.writeInt(calendar.get(Calendar.DAY_OF_MONTH));
}
```

Reading the data back is just as simple.

```
public void readData(DataInput in) throws IOException
{
    name = DataIO.readFixedString(NAME_SIZE, in);
    salary = in.readDouble();
    int y = in.readInt();
    int m = in.readInt();
    int d = in.readInt();
    GregorianCalendar calendar = new GregorianCalendar(y, m - 1, d);
    hireDay = calendar.getTime();
}
```

Let us compute the size of each record. We will use 40 characters for the name strings. Therefore, each record contains 100 bytes:

- 40 characters = 80 bytes for the name
- 1 `double` = 8 bytes for the salary
- 3 `int` = 12 bytes for the date

The program shown in [Listing 1-2](#) writes three records into a data file and then reads them from the file in reverse order. To do this efficiently requires random access—we need to get at the third record first.

Listing 1-2. RandomFileTest.java

Code View:

```
1. import java.io.*;
2. import java.util.*;
3.
4. /**
```

```
5.      * @version 1.11 2004-05-11
6.      * @author Cay Horstmann
7.      */
8.
9. public class RandomFileTest
10. {
11.     public static void main(String[] args)
12.     {
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.         try
20.         {
21.             // save all employee records to the file employee.dat
22.             DataOutputStream out = new DataOutputStream(new FileOutputStream("employee.dat"));
23.             for (Employee e : staff)
24.                 e.writeData(out);
25.             out.close();
26.
27.             // retrieve all records into a new array
28.             RandomAccessFile in = new RandomAccessFile("employee.dat", "r");
29.             // compute the array size
30.             int n = (int) (in.length() / Employee.RECORD_SIZE);
31.             Employee[] newStaff = new Employee[n];
32.
33.             // read employees in reverse order
34.             for (int i = n - 1; i >= 0; i--)
35.             {
36.                 newStaff[i] = new Employee();
37.                 in.seek(i * Employee.RECORD_SIZE);
38.                 newStaff[i].readData(in);
39.             }
40.             in.close();
41.
42.             // print the newly read employee records
43.             for (Employee e : newStaff)
44.                 System.out.println(e);
45.         }
46.         catch (IOException e)
47.         {
48.             e.printStackTrace();
49.         }
50.     }
51. }
52.
53. class Employee
54. {
55.     public Employee() {}
56.
57.     public Employee(String n, double s, int year, int month, int day)
58.     {
59.         name = n;
60.         salary = s;
61.         GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
62.         hireDay = calendar.getTime();
63.     }
64.
65.     public String getName()
66.     {
67.         return name;
68.     }
69.
70.     public double getSalary()
71.     {
72.         return salary;
73.     }
74.
75.     public Date getHireDay()
76.     {
77.         return hireDay;
78.     }
79.
80.     /**
81.      * Raises the salary of this employee.
82.      * @param byPercent the percentage of the raise
83.     */
84.     public void raiseSalary(double byPercent)
```

```
85.     {
86.         double raise = salary * byPercent / 100;
87.         salary += raise;
88.     }
89.
90.    public String toString()
91.    {
92.        return getClass().getName()
93.            + "[name=" + name
94.            + ",salary=" + salary
95.            + ",hireDay=" + hireDay
96.            + "]";
97.    }
98.
99.    /**
100.       Writes employee data to a data output
101.       @param out the data output
102.    */
103.   public void writeData(DataOutput out) throws IOException
104.   {
105.       DataIO.writeFixedString(name, NAME_SIZE, out);
106.       out.writeDouble(salary);
107.
108.       GregorianCalendar calendar = new GregorianCalendar();
109.       calendar.setTime(hireDay);
110.       out.writeInt(calendar.get(Calendar.YEAR));
111.       out.writeInt(calendar.get(Calendar.MONTH) + 1);
112.       out.writeInt(calendar.get(Calendar.DAY_OF_MONTH));
113.   }
114.
115. /**
116.    Reads employee data from a data input
117.    @param in the data input
118. */
119.   public void readData(DataInput in) throws IOException
120.   {
121.       name = DataIO.readFixedString(NAME_SIZE, in);
122.       salary = in.readDouble();
123.       int y = in.readInt();
124.       int m = in.readInt();
125.       int d = in.readInt();
126.       GregorianCalendar calendar = new GregorianCalendar(y, m - 1, d);
127.       hireDay = calendar.getTime();
128.   }
129.
130.   public static final int NAME_SIZE = 40;
131.   public static final int RECORD_SIZE = 2 * NAME_SIZE + 8 + 4 + 4 + 4;
132.
133.   private String name;
134.   private double salary;
135.   private Date hireDay;
136. }
137.
138. class DataIO
139. {
140.     public static String readFixedString(int size, DataInput in)
141.             throws IOException
142.     {
143.         StringBuilder b = new StringBuilder(size);
144.         int i = 0;
145.         boolean more = true;
146.         while (more && i < size)
147.         {
148.             char ch = in.readChar();
149.             i++;
150.             if (ch == 0) more = false;
151.             else b.append(ch);
152.         }
153.         in.skipBytes(2 * (size - i));
154.         return b.toString();
155.     }
156.
157.     public static void writeFixedString(String s, int size, DataOutput out)
158.             throws IOException
159.     {
160.         for (int i = 0; i < size; i++)
161.         {
162.             char ch = 0;
163.             if (i < s.length()) ch = s.charAt(i);
164.             out.writeChar(ch);
```

```
165.     }
166.   }
167. }
```

API

java.io.RandomAccessFile 1.0

- RandomAccessFile(String file, String mode)
- RandomAccessFile(File file, String mode)

Parameters: file The file to be opened
mode "r" for read-only mode, "rw" for read/write mode, "rws" for read/write mode with synchronous disk writes of data and metadata for every update, and "rwd" for read/write mode with synchronous disk writes of data only

- long getFilePointer()
returns the current location of the file pointer.
- void seek(long pos)
sets the file pointer to pos bytes from the beginning of the file.
- long length()
returns the length of the file in bytes.



ZIP Archives

ZIP archives store one or more files in (usually) compressed format. Each ZIP archive has a header with information such as the name of the file and the compression method that was used. In Java, you use a `ZipInputStream` to read a ZIP archive. You need to look at the individual entries in the archive. The `getNextEntry` method returns an object of type `ZipEntry` that describes the entry. The `read` method of the `ZipInputStream` is modified to return -1 at the end of the current entry (instead of just at the end of the ZIP file). You must then call `closeEntry` to read the next entry. Here is a typical code sequence to read through a ZIP file:

```
ZipInputStream zin = new ZipInputStream(new FileInputStream(zipname));
ZipEntry entry;
while ((entry = zin.getNextEntry()) != null)
{
    analyze entry;
    read the contents of zin;
    zin.closeEntry();
}
zin.close();
```

To read the contents of a ZIP entry, you will probably not want to use the raw `read` method; usually, you will use the methods of a more competent stream filter. For example, to read a text file inside a ZIP file, you can use the following loop:

```
Scanner in = new Scanner(zin);
while (in.hasNextLine())
    do something with in.nextLine();
```

Note

- The ZIP input stream throws a `ZipException` when there is an error in reading a ZIP file. Normally this error occurs when the ZIP file has been corrupted.

To write a ZIP file, you use a `ZipOutputStream`. For each entry that you want to place into the ZIP file, you create a `ZipEntry` object. You pass the file name to the `ZipEntry` constructor; it sets the other parameters such as file date and decompression method. You can override these settings if you like. Then, you call the `putNextEntry` method of the `ZipOutputStream` to begin writing a new file. Send the file data to the ZIP stream. When you are done, call `closeEntry`. Repeat for all the files you want to store. Here is a code skeleton:

```
FileOutputStream fout = new FileOutputStream("test.zip");
ZipOutputStream zout = new ZipOutputStream(fout);
for all files
{
    ZipEntry ze = new ZipEntry(filename);
    zout.putNextEntry(ze);
    send data to zout;
    zout.closeEntry();
}
zout.close();
```

Note

- JAR files (which were discussed in Volume I, Chapter 10) are simply ZIP files with another entry, the so-called manifest. You use the `JarInputStream` and `JarOutputStream` classes to read and write the manifest entry.

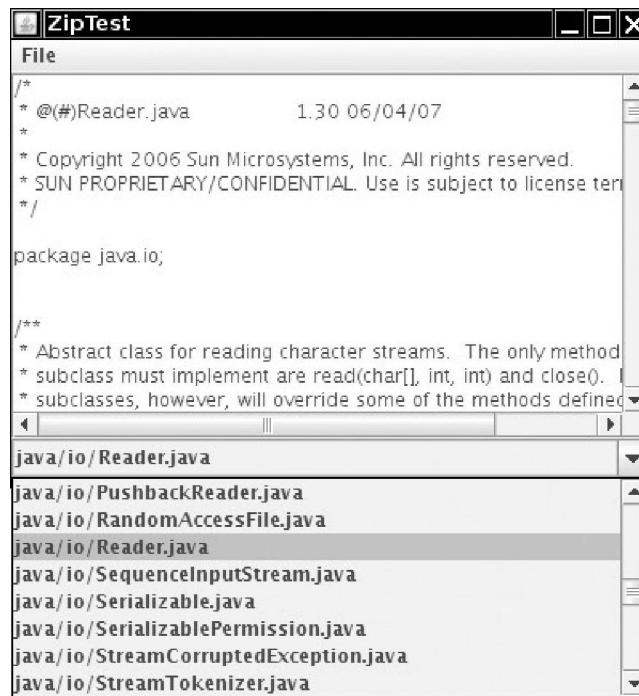
ZIP streams are a good example of the power of the stream abstraction. When you read the data that are stored in compressed form, you don't worry that the data are being decompressed as they are being requested. And the source of the bytes in ZIP formats need not be a file—the ZIP data can come from a network connection. In fact, whenever the class loader of an applet reads a JAR file, it reads and decompresses data from the network.

Note

- The article at <http://www.javaworld.com/javaworld/jw-10-2000/jw-1027-toolbox.html> shows you how to modify a ZIP archive.

The program shown in [Listing 1-3](#) lets you open a ZIP file. It then displays the files stored in the ZIP archive in the combo box at the bottom of the screen. If you select one of the files, the contents of the file are displayed in the text area, as shown in [Figure 1-5](#).

Figure 1-5. The ZipTest program



Listing 1-3. zipTest.java

Code View:

```
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.zip.*;
7. import javax.swing.*;
8.
9. /**
10. * @version 1.32 2007-06-22
11. * @author Cay Horstmann
12. */
13. public class ZipTest
14. {
15.     public static void main(String[] args)
16.     {
17.         EventQueue.invokeLater(new Runnable()
18.         {
19.             public void run()
20.             {
21.                 ZipTestFrame frame = new ZipTestFrame();
22.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23.                 frame.setVisible(true);
24.             }
25.         });
26.     }
27. }
28.
29. /**
30. * A frame with a text area to show the contents of a file inside a ZIP archive, a combo
31. * box to select different files in the archive, and a menu to load a new archive.
32. */
33. class ZipTestFrame extends JFrame
34. {
35.     public ZipTestFrame()
36.     {
37.         setTitle("ZipTest");
38.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
39.
40.         // add the menu and the Open and Exit menu items
41.         JMenuBar menuBar = new JMenuBar();
42.         JMenu menu = new JMenu("File");
43.
44.         JMenuItem openItem = new JMenuItem("Open");
```

```
45.     menu.add(openItem);
46.     openItem.addActionListener(new ActionListener()
47.     {
48.         public void actionPerformed(ActionEvent event)
49.         {
50.             JFileChooser chooser = new JFileChooser();
51.             chooser.setCurrentDirectory(new File("."));
52.             int r = chooser.showOpenDialog(ZipTestFrame.this);
53.             if (r == JFileChooser.APPROVE_OPTION)
54.             {
55.                 zipname = chooser.getSelectedFile().getPath();
56.                 fileCombo.removeAllItems();
57.                 scanZipFile();
58.             }
59.         }
60.     });
61.
62.     JMenuItem exitItem = new JMenuItem("Exit");
63.     menu.add(exitItem);
64.     exitItem.addActionListener(new ActionListener()
65.     {
66.         public void actionPerformed(ActionEvent event)
67.         {
68.             System.exit(0);
69.         }
70.     });
71.
72.     menuBar.add(menu);
73.     setJMenuBar(menuBar);
74.
75.     // add the text area and combo box
76.     fileText = new JTextArea();
77.     fileCombo = new JComboBox();
78.     fileCombo.addActionListener(new ActionListener()
79.     {
80.         public void actionPerformed(ActionEvent event)
81.         {
82.             loadZipFile((String) fileCombo.getSelectedItem());
83.         }
84.     });
85.
86.     add(fileCombo, BorderLayout.SOUTH);
87.     add(new JScrollPane(fileText), BorderLayout.CENTER);
88. }
89.
90. /**
91. * Scans the contents of the ZIP archive and populates the combo box.
92. */
93. public void scanZipFile()
94. {
95.     new SwingWorker<Void, String>()
96.     {
97.         protected Void doInBackground() throws Exception
98.         {
99.             ZipInputStream zin = new ZipInputStream(new FileInputStream(zipname));
100.            ZipEntry entry;
101.            while ((entry = zin.getNextEntry()) != null)
102.            {
103.                publish(entry.getName());
104.                zin.closeEntry();
105.            }
106.            zin.close();
107.            return null;
108.        }
109.
110.        protected void process(List<String> names)
111.        {
112.            for (String name : names)
113.                fileCombo.addItem(name);
114.
115.        }
116.    }.execute();
117. }
118.
119. /**
120. * Loads a file from the ZIP archive into the text area
121. * @param name the name of the file in the archive
122. */
123. public void loadZipFile(final String name)
124. {
```

```
125.     fileCombo.setEnabled(false);
126.     fileText.setText("");
127.     new SwingWorker<Void, Void>()
128.     {
129.         protected Void doInBackground() throws Exception
130.         {
131.             try
132.             {
133.                 ZipInputStream zin = new ZipInputStream(new FileInputStream(zipname));
134.                 ZipEntry entry;
135.
136.                 // find entry with matching name in archive
137.                 while ((entry = zin.getNextEntry()) != null)
138.                 {
139.                     if (entry.getName().equals(name))
140.                     {
141.                         // read entry into text area
142.                         Scanner in = new Scanner(zin);
143.                         while (in.hasNextLine())
144.                         {
145.                             fileText.append(in.nextLine());
146.                             fileText.append("\n");
147.                         }
148.                     }
149.                     zin.closeEntry();
150.                 }
151.                 zin.close();
152.             }
153.             catch (IOException e)
154.             {
155.                 e.printStackTrace();
156.             }
157.             return null;
158.         }
159.
160.         protected void done()
161.         {
162.             fileCombo.setEnabled(true);
163.         }
164.     }.execute();
165. }
166.
167. public static final int DEFAULT_WIDTH = 400;
168. public static final int DEFAULT_HEIGHT = 300;
169. private JComboBox fileCombo;
170. private JTextArea fileText;
171. private String zipname;
172. }
```

API**java.util.zip.ZipInputStream 1.1**

- `ZipInputStream(InputStream in)`

creates a `ZipInputStream` that allows you to inflate data from the given `InputStream`.

- `ZipEntry getNextEntry()`

returns a `ZipEntry` object for the next entry, or `null` if there are no more entries.

- `void closeEntry()`

closes the current open entry in the ZIP file. You can then read the next entry by using `getNextEntry()`.



java.util.zip.ZipOutputStream 1.1

- ZipOutputStream(OutputStream out)

creates a ZipOutputStream that you use to write compressed data to the specified OutputStream.

- void putNextEntry(ZipEntry ze)

writes the information in the given ZipEntry to the stream and positions the stream for the data. The data can then be written to the stream by write().

- void closeEntry()

closes the currently open entry in the ZIP file. Use the putNextEntry method to start the next entry.

- void setLevel(int level)

sets the default compression level of subsequent DEFLATED entries. The default value is Deflater.DEFAULT_COMPRESSION. Throws an IllegalArgumentException if the level is not valid.

Parameters: level A compression level, from 0 (NO_COMPRESSION) to 9 (BEST_COMPRESSION)

- void setMethod(int method)

sets the default compression method for this ZipOutputStream for any entries that do not specify a method.

Parameters: method The compression method, either DEFLATED or STORED



java.util.zip.ZipEntry 1.1

- ZipEntry(String name)

Parameters: name The name of the entry

- long getCrc()

returns the CRC32 checksum value for this ZipEntry.

- String getName()

returns the name of this entry.

- long getSize()

returns the uncompressed size of this entry, or -1 if the uncompressed size is not known.

- boolean isDirectory()

returns true if this entry is a directory.

- `void setMethod(int method)`

Parameters: `method` The compression method for the entry; must be either `DEFLATED` or `STORED`

- `void setSize(long size)`

sets the size of this entry. Only required if the compression method is `STORED`.

Parameters: `size` The uncompressed size of this entry

- `void setCrc(long crc)`

sets the CRC32 checksum of this entry. Use the `CRC32` class to compute this checksum. Only required if the compression method is `STORED`.

Parameters: `crc` The checksum of this entry

API

`java.util.zip.ZipFile 1.1`

- `ZipFile(String name)`
- `ZipFile(File file)`

creates a `ZipFile` for reading from the given string or `File` object.
- `Enumeration entries()`

returns an `Enumeration` object that enumerates the `ZipEntry` objects that describe the entries of the `ZipFile`.
- `ZipEntry getEntry(String name)`

returns the entry corresponding to the given name, or `null` if there is no such entry.

Parameters: `name` The entry name

- `InputStream getInputStream(ZipEntry ze)`

returns an `InputStream` for the given entry.

Parameters: `ze` A `ZipEntry` in the ZIP file

- `String getName()`

returns the path of this ZIP file.



Object Streams and Serialization

Using a fixed-length record format is a good choice if you need to store data of the same type. However, objects that you

create in an object-oriented program are rarely all of the same type. For example, you might have an array called `staff` that is nominally an array of `Employee` records but contains objects that are actually instances of a subclass such as `Manager`.

It is certainly possible to come up with a data format that allows you to store such polymorphic collections, but fortunately, we don't have to. The Java language supports a very general mechanism, called object serialization, that makes it possible to write any object to a stream and read it again later. (You will see later in this chapter where the term "serialization" comes from.)

To save object data, you first need to open an `ObjectOutputStream` object:

Code View:

```
ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("employee.dat"));
```

Now, to save an object, you simply use the `writeObject` method of the `ObjectOutputStream` class as in the following fragment:

```
Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);
Manager boss = new Manager("Carl Cracker", 80000, 1987, 12, 15);
out.writeObject(harry);
out.writeObject(boss);
```

To read the objects back in, first get an `ObjectInputStream` object:

Code View:

```
ObjectInputStream in = new ObjectInputStream(new FileInputStream("employee.dat"));
```

Then, retrieve the objects in the same order in which they were written, using the `readObject` method.

```
Employee e1 = (Employee) in.readObject();
Employee e2 = (Employee) in.readObject();
```

There is, however, one change you need to make to any class that you want to save and restore in an object stream. The class must implement the `Serializable` interface:

```
class Employee implements Serializable { . . . }
```

The `Serializable` interface has no methods, so you don't need to change your classes in any way. In this regard, it is similar to the `Cloneable` interface that we discussed in Volume I, Chapter 6. However, to make a class cloneable, you still had to override the `clone` method of the `Object` class. To make a class serializable, you do not need to do anything else.

Note



You can write and read only objects with the `writeObject/readObject` methods. For primitive type values, you use methods such as `writeInt/readInt` or `writeDouble/readDouble`. (The object stream classes implement the `DataInput/DataOutput` interfaces.)

Behind the scenes, an `ObjectOutputStream` looks at all fields of the objects and saves their contents. For example, when writing an `Employee` object, the name, date, and salary fields are written to the output stream.

However, there is one important situation that we need to consider: What happens when one object is shared by several objects as part of its state?

To illustrate the problem, let us make a slight modification to the `Manager` class. Let's assume that each manager has a secretary:

```
class Manager extends Employee
{
```

```

    . . .
    private Employee secretary;
}

```

Each `Manager` object now contains a reference to the `Employee` object that describes the secretary. Of course, two managers can share the same secretary, as is the case in [Figure 1-6](#) and the following code:

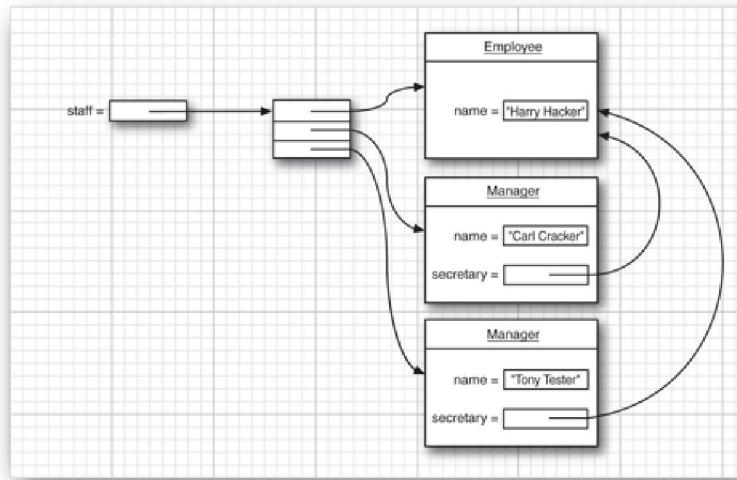
```

harry = new Employee("Harry Hacker", . . .);
Manager carl = new Manager("Carl Cracker", . . .);
carl.setSecretary(harry);
Manager tony = new Manager("Tony Tester", . . .);
tony.setSecretary(harry);

```

Figure 1-6. Two managers can share a mutual employee

[\[View full size image\]](#)



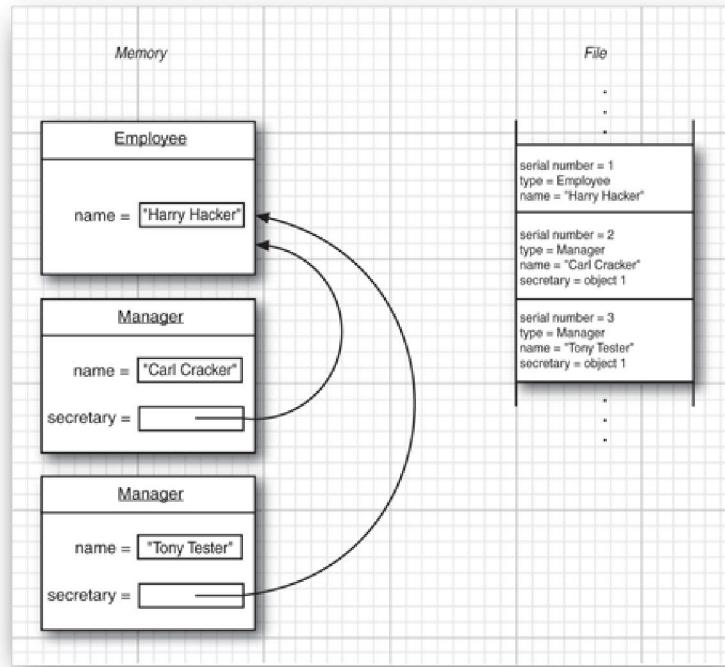
Saving such a network of objects is a challenge. Of course, we cannot save and restore the memory addresses for the secretary objects. When an object is reloaded, it will likely occupy a completely different memory address than it originally did.

Instead, each object is saved with a serial number, hence the name object serialization for this mechanism. Here is the algorithm:

- Associate a serial number with each object reference that you encounter (as shown in [Figure 1-7](#)).

Figure 1-7. An example of object serialization

[\[View full size image\]](#)



- When encountering an object reference for the first time, save the object data to the stream.
- If it has been saved previously, just write "same as previously saved object with serial number x."

When reading back the objects, the procedure is reversed.

- When an object is specified in the stream for the first time, construct it, initialize it with the stream data, and remember the association between the sequence number and the object reference.
- When the tag "same as previously saved object with serial number x," is encountered, retrieve the object reference for the sequence number.

Note



In this chapter, we use serialization to save a collection of objects to a disk file and retrieve it exactly as we stored it. Another very important application is the transmittal of a collection of objects across a network connection to another computer. Just as raw memory addresses are meaningless in a file, they are also meaningless when communicating with a different processor. Because serialization replaces memory addresses with serial numbers, it permits the transport of object collections from one machine to another. We study that use of serialization when discussing remote method invocation in [Chapter 5](#).

[Listing 1-4](#) is a program that saves and reloads a network of `Employee` and `Manager` objects (some of which share the same employee as a secretary). Note that the secretary object is unique after reloading—when `newStaff[1]` gets a raise, that is reflected in the `secretary` fields of the managers.

[Listing 1-4. ObjectStreamTest.java](#)

Code View:

```

1. import java.io.*;
2. import java.util.*;
3.
4. /**
5.  * @version 1.10 17 Aug 1998
6.  * @author Cay Horstmann
7. */
8. class ObjectStreamTest
9. {
10.    public static void main(String[] args)
11.    {
12.        Employee harry = new Employee("Harry Hacker", 50000, 1989, 10, 1);

```

```
13.     Manager carl = new Manager("Carl Cracker", 80000, 1987, 12, 15);
14.     carl.setSecretary(harry);
15.     Manager tony = new Manager("Tony Tester", 40000, 1990, 3, 15);
16.     tony.setSecretary(harry);
17.
18.     Employee[] staff = new Employee[3];
19.
20.     staff[0] = carl;
21.     staff[1] = harry;
22.     staff[2] = tony;
23.
24.     try
25.     {
26.         // save all employee records to the file employee.dat
27.         ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("employee.dat"));
28.         out.writeObject(staff);
29.         out.close();
30.
31.         // retrieve all records into a new array
32.         ObjectInputStream in = new ObjectInputStream(new FileInputStream("employee.dat"));
33.         Employee[] newStaff = (Employee[]) in.readObject();
34.         in.close();
35.
36.         // raise secretary's salary
37.         newStaff[1].raiseSalary(10);
38.
39.         // print the newly read employee records
40.         for (Employee e : newStaff)
41.             System.out.println(e);
42.     }
43.     catch (Exception e)
44.     {
45.         e.printStackTrace();
46.     }
47. }
48. }
49.
50. class Employee implements Serializable
51. {
52.     public Employee()
53.     {
54.     }
55.
56.     public Employee(String n, double s, int year, int month, int day)
57.     {
58.         name = n;
59.         salary = s;
60.         GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
61.         hireDay = calendar.getTime();
62.     }
63.
64.     public String getName()
65.     {
66.         return name;
67.     }
68.
69.     public double getSalary()
70.     {
71.         return salary;
72.     }
73.
74.     public Date getHireDay()
75.     {
76.         return hireDay;
77.     }
78.
79.     public void raiseSalary(double byPercent)
80.     {
81.         double raise = salary * byPercent / 100;
82.         salary += raise;
83.     }
84.
85.     public String toString()
86.     {
87.         return getClass().getName() + "[name=" + name + ",salary=" + salary + ",hireDay="
88.             + hireDay + "]";
89.     }
90.
91.     private String name;
92.     private double salary;
```

```
93.     private Date hireDay;
94. }
95.
96. class Manager extends Employee
97. {
98.     /**
99.      * Constructs a Manager without a secretary
100.     * @param n the employee's name
101.     * @param s the salary
102.     * @param year the hire year
103.     * @param month the hire month
104.     * @param day the hire day
105.     */
106.    public Manager(String n, double s, int year, int month, int day)
107.    {
108.        super(n, s, year, month, day);
109.        secretary = null;
110.    }
111.
112.    /**
113.     * Assigns a secretary to the manager
114.     * @param s the secretary
115.     */
116.    public void setSecretary(Employee s)
117.    {
118.        secretary = s;
119.    }
120.
121.    public String toString()
122.    {
123.        return super.toString() + "[secretary=" + secretary + "]";
124.    }
125.
126.    private Employee secretary;
127. }
```

API**java.io.ObjectOutputStream 1.1**

- `ObjectOutputStream(OutputStream out)`

creates an `ObjectOutputStream` so that you can write objects to the specified `OutputStream`.

- `void writeObject(Object obj)`

writes the specified object to the `ObjectOutputStream`. This method saves the class of the object, the signature of the class, and the values of any nonstatic, nontransient field of the class and its superclasses.

API**java.io.ObjectInputStream 1.1**

- `ObjectInputStream(InputStream in)`

creates an `ObjectInputStream` to read back object information from the specified `InputStream`.

- `Object readObject()`

reads an object from the `ObjectInputStream`. In particular, this method reads back the class of the object, the signature of the class, and the values of the nontransient and nonstatic fields of the class and all its superclasses. It does deserializing to

allow multiple object references to be recovered.

Understanding the Object Serialization File Format

Object serialization saves object data in a particular file format. Of course, you can use the `writeObject/readObject` methods without having to know the exact sequence of bytes that represents objects in a file. Nonetheless, we found studying the data format to be extremely helpful for gaining insight into the object streaming process. Because the details are somewhat technical, feel free to skip this section if you are not interested in the implementation.

Every file begins with the two-byte "magic number"

AC ED

followed by the version number of the object serialization format, which is currently

00 05

(We use hexadecimal numbers throughout this section to denote bytes.) Then, it contains a sequence of objects, in the order that they were saved.

String objects are saved as

74 two-byte length characters

For example, the string "Harry" is saved as

74 00 05 Harry

The Unicode characters of the string are saved in "modified UTF-8" format.

When an object is saved, the class of that object must be saved as well. The class description contains

- The name of the class.
- The serial version unique ID, which is a fingerprint of the data field types and method signatures.
- A set of flags describing the serialization method.
- A description of the data fields.

The fingerprint is obtained by ordering descriptions of the class, superclass, interfaces, field types, and method signatures in a canonical way, and then applying the so-called Secure Hash Algorithm (SHA) to that data.

SHA is a fast algorithm that gives a "fingerprint" to a larger block of information. This fingerprint is always a 20-byte data packet, regardless of the size of the original data. It is created by a clever sequence of bit operations on the data that makes it essentially 100 percent certain that the fingerprint will change if the information is altered in any way. (For more details on SHA, see, for example, *Cryptography and Network Security: Principles and Practice*, by William Stallings [Prentice Hall, 2002].) However, the serialization mechanism uses only the first 8 bytes of the SHA code as a class fingerprint. It is still very likely that the class fingerprint will change if the data fields or methods change.

When reading an object, its fingerprint is compared against the current fingerprint of the class. If they don't match, then the class definition has changed after the object was written, and an exception is generated. Of course, in practice, classes do evolve, and it might be necessary for a program to read in older versions of objects. We discuss this later in the section entitled "[Versioning](#)" on page [54](#).

Here is how a class identifier is stored:

72

2-byte length of class name

```

class name
8-byte fingerprint
1-byte flag
2-byte count of data field descriptors
data field descriptors
78 (end marker)
superclass type (70 if none)

```

The flag byte is composed of three bit masks, defined in `java.io.ObjectStreamConstants`:

```

static final byte SC_WRITE_METHOD = 1;
    // class has writeObject method that writes additional data
static final byte SC_SERIALIZABLE = 2;
    // class implements Serializable interface
static final byte SC_EXTERNALIZABLE = 4;
    // class implements Externalizable interface

```

We discuss the `Externalizable` interface later in this chapter. Externalizable classes supply custom read and write methods that take over the output of their instance fields. The classes that we write implement the `Serializable` interface and will have a flag value of 02. The serializable `java.util.Date` class defines its own `readObject/writeObject` methods and has a flag of 03.

Each data field descriptor has the format:

```

1-byte type code
2-byte length of field name
field name
class name (if field is an object)

```

The type code is one of the following:

B	byte
C	char
D	double
F	float
I	int
J	long
L	object
S	short
Z	boolean
[array

When the type code is `L`, the field name is followed by the field type. Class and field name strings do not start with the string code 74, but field types do. Field types use a slightly different encoding of their names, namely, the format used by native methods.

For example, the salary field of the `Employee` class is encoded as:

```
D 00 06 salary
```

Here is the complete class descriptor of the `Employee` class:

72 00 08 Employee	
E6 D2 86 7D AE AC 18 1B 02	Fingerprint and flags
00 03	Number of instance fields
D 00 06 salary	Instance field type and name
L 00 07 hireDay	Instance field type and name
74 00 10 Ljava/util/Date;	Instance field class name—Date
L 00 04 name	Instance field type and name
74 00 12 Ljava/lang/String;	Instance field class name—String
78	End marker
70	No superclass

These descriptors are fairly long. If the same class descriptor is needed again in the file, an abbreviated form is used:

71 4-byte serial number

The serial number refers to the previous explicit class descriptor. We discuss the numbering scheme later.

An object is stored as

73 class descriptor object data

For example, here is how an `Employee` object is stored:

40 E8 6A 00 00 00 00 00	salary field value—double
73	hireDay field value—new object
71 00 7E 00 08	Existing class <code>java.util.Date</code>
77 08 00 00 00 91 1B 4E B1 80 78	External storage—details later
74 00 0C Harry Hacker	name field value—String

As you can see, the data file contains enough information to restore the `Employee` object.

Arrays are saved in the following format:

75 class descriptor 4-byte number of entries entries

The array class name in the class descriptor is in the same format as that used by native methods (which is slightly different from the class name used by class names in other class descriptors). In this format, class names start with an `L` and end with a semicolon.

For example, an array of three `Employee` objects starts out like this:

75	Array
72 00 0B [LEmployee;	New class, string length, class name <code>Employee</code> []
FC BF 36 11 C5 91 11 C7 02	Fingerprint and flags
00 00	Number of instance fields
78	End marker
70	No superclass
00 00 00 03	Number of array entries

Note that the fingerprint for an array of `Employee` objects is different from a fingerprint of the `Employee` class itself.

All objects (including arrays and strings) and all class descriptors are given serial numbers as they are saved in the output file. The numbers start at 00 7E 00 00.

We already saw that a full class descriptor for any given class occurs only once. Subsequent descriptors refer to it. For example, in our previous example, a repeated reference to the `Date` class was coded as

```
71 00 7E 00 08
```

The same mechanism is used for objects. If a reference to a previously saved object is written, it is saved in exactly the same way; that is, 71 followed by the serial number. It is always clear from the context whether the particular serial reference denotes a class descriptor or an object.

Finally, a null reference is stored as

```
70
```

Here is the commented output of the `ObjectRefTest` program of the preceding section. If you like, run the program, look at a hex dump of its data file `employee.dat`, and compare it with the commented listing. The important lines toward the end of the output show the reference to a previously saved object.

AC ED 00 05	File header
75	Array <code>staff</code> (serial #1)
72 00 0B [LEmployee;	New class, string length, class name <code>Employee</code> [] (serial #0)
FC BF 36 11 C5 91 11 C7 02	Fingerprint and flags
00 00	Number of instance fields
78	End marker
70	No superclass
00 00 00 03	Number of array entries
73	<code>staff[0]</code> —new object (serial #7)
72 00 07 Manager	New class, string length, class name (serial #2)
36 06 AE 13 63 8F 59 B7 02	Fingerprint and flags
00 01	Number of data fields
L 00 09 secretary	Instance field type and name
74 00 0A LEmployee;	Instance field class name— <code>String</code> (serial #3)
78	End marker
72 00 08 Employee	Superclass—new class, string length, class name (serial #4)
E6 D2 86 7D AE AC 18 1B 02	Fingerprint and flags
00 03	Number of instance fields
D 00 06 salary	Instance field type and name
L 00 07 hireDay	Instance field type and name
74 00 10 Ljava/util/Date;	Instance field class name— <code>String</code> (serial #5)
L 00 04 name	Instance field type and name
74 00 12 Ljava/lang/String;	Instance field class name— <code>String</code> (serial #6)
78	End marker
70	No superclass

40 F3 88 00 00 00 00 00	salary field value—double
73	hireDay field value—new object (serial #9)
72 00 0E java.util.Date	New class, string length, class name (serial #8)
68 6A 81 01 4B 59 74 19 03	Fingerprint and flags
00 00	No instance variables
78	End marker
70	No superclass
77 08	External storage, number of bytes
00 00 00 83 E9 39 E0 00	Date
78	End marker
74 00 0C Carl Cracker	name field value—String (serial #10)
73	secretary field value—new object (serial #11)
71 00 7E 00 04	existing class (use serial #4)
40 E8 6A 00 00 00 00 00	salary field value—double
73	hireDay field value—new object (serial #12)
71 00 7E 00 08	Existing class (use serial #8)
77 08	External storage, number of bytes
00 00 00 91 1B 4E B1 80	Date
78	End marker
74 00 0C Harry Hacker	name field value—String (serial #13)
71 00 7E 00 0B	staff[1]—existing object (use serial #11)
73	staff[2]—new object (serial #14)
71 00 7E 00 02	Existing class (use serial #2)
40 E3 88 00 00 00 00 00	salary field value—double
73	hireDay field value—new object (serial #15)
71 00 7E 00 08	Existing class (use serial #8)
77 08	External storage, number of bytes
00 00 00 94 6D 3E EC 00 00	Date
78	End marker
74 00 0B Tony Tester	name field value—String (serial #16)
71 00 7E 00 0B	secretary field value—existing object (use serial #11)

Of course, studying these codes can be about as exciting as reading the average phone book. It is not important to know the exact file format (unless you are trying to create an evil effect by modifying the data), but it is still instructive to know that the object stream contains a detailed description of all the objects that it contains, with sufficient detail to allow reconstruction of both objects and arrays of objects.

What you should remember is this:

- The object stream output contains the types and data fields of all objects.
- Each object is assigned a serial number.
- Repeated occurrences of the same object are stored as references to that serial number.

Modifying the Default Serialization Mechanism

Certain data fields should never be serialized, for example, integer values that store file handles or handles of windows that are only meaningful to native methods. Such information is guaranteed to be useless when you reload an object at a later time or transport it to a different machine. In fact, improper values for such fields can actually cause native methods to crash. Java has an easy mechanism to prevent such fields from ever being serialized. Mark them with the keyword `transient`. You also need to tag fields as `transient` if they belong to nonserializable classes. Transient fields are always skipped when objects are serialized.

The serialization mechanism provides a way for individual classes to add validation or any other desired action to the default read and write behavior. A serializable class can define methods with the signature

```
private void readObject(ObjectInputStream in)
    throws IOException, ClassNotFoundException;
private void writeObject(ObjectOutputStream out)
    throws IOException;
```

Then, the data fields are no longer automatically serialized, and these methods are called instead.

Here is a typical example. A number of classes in the `java.awt.geom` package, such as `Point2D.Double`, are not serializable. Now suppose you want to serialize a class `LabeledPoint` that stores a `String` and a `Point2D.Double`. First, you need to mark the `Point2D.Double` field as `transient` to avoid a `NotSerializableException`.

```
public class LabeledPoint implements Serializable
{
    .
    .
    private String label;
    private transient Point2D.Double point;
}
```

In the `writeObject` method, we first write the object descriptor and the `String` field, `label`, by calling the `defaultWriteObject` method. This is a special method of the `ObjectOutputStream` class that can only be called from within a `writeObject` method of a serializable class. Then we write the point coordinates, using the standard `DataOutput` calls.

```
private void writeObject(ObjectOutputStream out)
    throws IOException
{
    out.defaultWriteObject();
    out.writeDouble(point.getX());
    out.writeDouble(point.getY());
}
```

In the `readObject` method, we reverse the process:

```
private void readObject(ObjectInputStream in)
    throws IOException
{
    in.defaultReadObject();
    double x = in.readDouble();
    double y = in.readDouble();
    point = new Point2D.Double(x, y);
}
```

Another example is the `java.util.Date` class that supplies its own `readObject` and `writeObject` methods. These methods write the date as a number of milliseconds from the epoch (January 1, 1970, midnight UTC). The `Date` class has a complex internal representation that stores both a `Calendar` object and a millisecond count to optimize lookups. The state of the `Calendar` is redundant and does not have to be saved.

The `readObject` and `writeObject` methods only need to save and load their data fields. They should not concern themselves with superclass data or any other class information.

Rather than letting the serialization mechanism save and restore object data, a class can define its own mechanism. To do this, a class must implement the `Externalizable` interface. This in turn requires it to define two methods:

```
public void readExternal(ObjectInputStream in)
    throws IOException, ClassNotFoundException;
public void writeExternal(ObjectOutputStream out)
    throws IOException;
```

Unlike the `readObject` and `writeObject` methods that were described in the preceding section, these methods are fully responsible for saving and restoring the entire object, including the superclass data. The serialization mechanism merely records the class of the object in the stream. When reading an externalizable object, the object stream creates an object with the default constructor and then calls the `readExternal` method. Here is how you can implement these methods for the `Employee` class:

```
public void readExternal(ObjectInput s)
    throws IOException
{
    name = s.readUTF();
    salary = s.readDouble();
    hireDay = new Date(s.readLong());
}

public void writeExternal(ObjectOutput s)
    throws IOException
{
    s.writeUTF(name);
    s.writeDouble(salary);
    s.writeLong(hireDay.getTime());
}
```

Tip



Serialization is somewhat slow because the virtual machine must discover the structure of each object. If you are concerned about performance and if you read and write a large number of objects of a particular class, you should investigate the use of the `Externalizable` interface. The tech tip <http://java.sun.com/developer/TechTips/2000/tt0425.html> demonstrates that in the case of an employee class, using external reading and writing was about 35 to 40 percent faster than the default serialization.

Caution



Unlike the `readObject` and `writeObject` methods, which are private and can only be called by the serialization mechanism, the `readExternal` and `writeExternal` methods are public. In particular, `readExternal` potentially permits modification of the state of an existing object.

Serializing Singletons and Typesafe Enumerations

You have to pay particular attention when serializing and deserializing objects that are assumed to be unique. This commonly happens when you are implementing singletons and typesafe enumerations.

If you use the `enum` construct of Java SE 5.0, then you need not worry about serialization—it just works. However, suppose you maintain legacy code that contains an enumerated type such as

```
public class Orientation
{
    public static final Orientation HORIZONTAL = new Orientation(1);
    public static final Orientation VERTICAL = new Orientation(2);
    private Orientation(int v) { value = v; }
    private int value;
}
```

This idiom was common before enumerations were added to the Java language. Note that the constructor is private. Thus, no objects can be created beyond `Orientation.HORIZONTAL` and `Orientation.VERTICAL`. In particular, you can use the `==` operator to test for object equality:

```
if (orientation == Orientation.HORIZONTAL) . . .
```

There is an important twist that you need to remember when a typesafe enumeration implements the `Serializable` interface. The default serialization mechanism is not appropriate. Suppose we write a value of type `Orientation` and read it in again:

```
Orientation original = Orientation.HORIZONTAL;
ObjectOutputStream out = . . .;
out.writeObject();
out.close();
ObjectInputStream in = . . .;
Orientation saved = (Orientation) in.read();
```

Now the test

```
if (saved == Orientation.HORIZONTAL) . . .
```

will fail. In fact, the `saved` value is a completely new object of the `Orientation` type and not equal to any of the predefined constants. Even though the constructor is private, the serialization mechanism can create new objects!

To solve this problem, you need to define another special serialization method, called `readResolve`. If the `readResolve` method is defined, it is called after the object is deserialized. It must return an object that then becomes the return value of the `readObject` method. In our case, the `readResolve` method will inspect the `value` field and return the appropriate enumerated constant:

```
protected Object readResolve() throws ObjectStreamException
{
    if (value == 1) return Orientation.HORIZONTAL;
    if (value == 2) return Orientation.VERTICAL;
    return null; // this shouldn't happen
}
```

Remember to add a `readResolve` method to all typesafe enumerations in your legacy code and to all classes that follow the singleton design pattern.

Versioning

If you use serialization to save objects, you will need to consider what happens when your program evolves. Can version 1.1 read the old files? Can the users who still use 1.0 read the files that the new version is now producing? Clearly, it would be desirable if object files could cope with the evolution of classes.

At first glance it seems that this would not be possible. When a class definition changes in any way, then its SHA fingerprint also changes, and you know that object streams will refuse to read in objects with different fingerprints. However, a class can indicate that it is compatible with an earlier version of itself. To do this, you must first obtain the fingerprint of the earlier version of the class. You use the stand-alone `serialver` program that is part of the JDK to obtain this number. For example, running

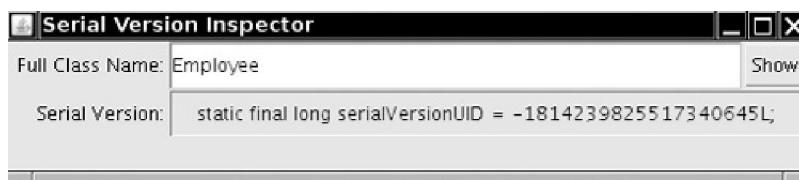
```
serialver Employee
```

prints

```
Employee: static final long serialVersionUID = -1814239825517340645L;
```

If you start the `serialver` program with the `-show` option, then the program brings up a graphical dialog box (see [Figure 1-8](#)).

Figure 1-8. The graphical version of the `serialver` program



All later versions of the class must define the `serialVersionUID` constant to the same fingerprint as the original.

```
class Employee implements Serializable // version 1.1
{
    .
    .
    public static final long serialVersionUID = -1814239825517340645L;
}
```

When a class has a static data member named `serialVersionUID`, it will not compute the fingerprint manually but instead will use that value.

Once that static data member has been placed inside a class, the serialization system is now willing to read in different

versions of objects of that class.

If only the methods of the class change, there is no problem with reading the new object data. However, if data fields change, then you may have problems. For example, the old file object may have more or fewer data fields than the one in the program, or the types of the data fields may be different. In that case, the object stream makes an effort to convert the stream object to the current version of the class.

The object stream compares the data fields of the current version of the class with the data fields of the version in the stream. Of course, the object stream considers only the nontransient and nonstatic data fields. If two fields have matching names but different types, then the object stream makes no effort to convert one type to the other—the objects are incompatible. If the object in the stream has data fields that are not present in the current version, then the object stream ignores the additional data. If the current version has data fields that are not present in the streamed object, the added fields are set to their default (`null` for objects, zero for numbers, and `false` for boolean values).

Here is an example. Suppose we have saved a number of employee records on disk, using the original version (1.0) of the class. Now we change the `Employee` class to version 2.0 by adding a data field called `department`. [Figure 1-9](#) shows what happens when a 1.0 object is read into a program that uses 2.0 objects. The `department` field is set to `null`. [Figure 1-10](#) shows the opposite scenario: A program using 1.0 objects reads a 2.0 object. The additional `department` field is ignored.

Figure 1-9. Reading an object with fewer data fields

[\[View full size image\]](#)

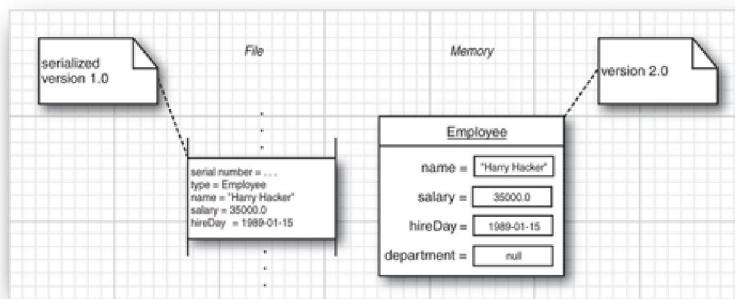
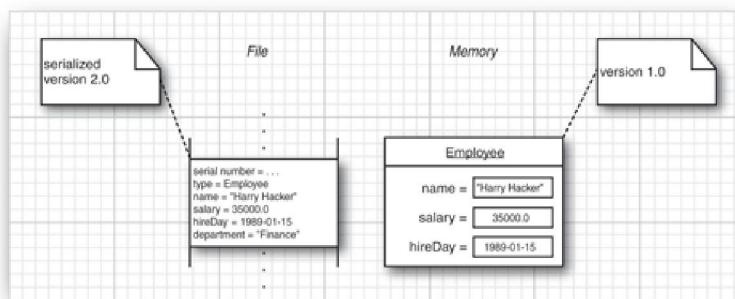


Figure 1-10. Reading an object with more data fields

[\[View full size image\]](#)



Is this process safe? It depends. Dropping a data field seems harmless—the recipient still has all the data that it knew how to manipulate. Setting a data field to `null` might not be so safe. Many classes work hard to initialize all data fields in all constructors to non-`null` values, so that the methods don't have to be prepared to handle `null` data. It is up to the class designer to implement additional code in the `readObject` method to fix version incompatibilities or to make sure the methods are robust enough to handle `null` data.

Using Serialization for Cloning

There is an amusing use for the serialization mechanism: It gives you an easy way to clone an object provided the class is serializable. Simply serialize it to an output stream and then read it back in. The result is a new object that is a deep copy of the existing object. You don't have to write the object to a file—you can use a `ByteArrayOutputStream` to save the data into a byte array.

As [Listing 1-5](#) shows, to get `clone` for free, simply extend the `Serializable` class, and you are done.

You should be aware that this method, although clever, will usually be much slower than a clone method that explicitly constructs a new object and copies or clones the data fields.

Listing 1-5. SerialCloneTest.java

Code View:

```
1. import java.io.*;
2. import java.util.*;
3.
4. public class SerialCloneTest
5. {
6.     public static void main(String[] args)
7.     {
8.         Employee harry = new Employee("Harry Hacker", 35000, 1989, 10, 1);
9.         // clone harry
10.        Employee harry2 = (Employee) harry.clone();
11.
12.        // mutate harry
13.        harry.raiseSalary(10);
14.
15.        // now harry and the clone are different
16.        System.out.println(harry);
17.        System.out.println(harry2);
18.    }
19. }
20.
21. /**
22.  * A class whose clone method uses serialization.
23. */
24. class SerialCloneable implements Cloneable, Serializable
25. {
26.     public Object clone()
27.     {
28.         try
29.         {
30.             // save the object to a byte array
31.             ByteArrayOutputStream bout = new ByteArrayOutputStream();
32.             ObjectOutputStream out = new ObjectOutputStream(bout);
33.             out.writeObject(this);
34.             out.close();
35.
36.             // read a clone of the object from the byte array
37.             ByteArrayInputStream bin = new ByteArrayInputStream(bout.toByteArray());
38.             ObjectInputStream in = new ObjectInputStream(bin);
39.             Object ret = in.readObject();
40.             in.close();
41.
42.             return ret;
43.         }
44.         catch (Exception e)
45.         {
46.             return null;
47.         }
48.     }
49. }
50.
51. /**
52.  * The familiar Employee class, redefined to extend the
53.  * SerialCloneable class.
54. */
55. class Employee extends SerialCloneable
56. {
57.     public Employee(String n, double s, int year, int month, int day)
58.     {
59.         name = n;
60.         salary = s;
61.         GregorianCalendar calendar = new GregorianCalendar(year, month - 1, day);
62.         hireDay = calendar.getTime();
63.     }
64.
65.     public String getName()
66.     {
67.         return name;
68.     }
69.
70.     public double getSalary()
71.     {
```

```
72.         return salary;
73.     }
74.
75.     public Date getHireDay()
76.     {
77.         return hireDay;
78.     }
79.
80.     public void raiseSalary(double byPercent)
81.     {
82.         double raise = salary * byPercent / 100;
83.         salary += raise;
84.     }
85.
86.     public String toString()
87.     {
88.         return getClass().getName()
89.             + "[name=" + name
90.             + ",salary=" + salary
91.             + ",hireDay=" + hireDay
92.             + "]";
93.     }
94.
95.     private String name;
96.     private double salary;
97.     private Date hireDay;
98. }
```



File Management

You have learned how to read and write data from a file. However, there is more to file management than reading and writing. The `File` class encapsulates the functionality that you will need to work with the file system on the user's machine. For example, you use the `File` class to find out when a file was last modified or to remove or rename the file. In other words, the stream classes are concerned with the contents of the file, whereas the `File` class is concerned with the storage of the file on a disk.

Note



As is so often the case in Java, the `File` class takes the least common denominator approach. For example, under Windows, you can find out (or set) the read-only flag for a file, but while you can find out if it is a hidden file, you can't hide it without using a native method.

The simplest constructor for a `File` object takes a (full) file name. If you don't supply a path name, then Java uses the current directory. For example,

```
File f = new File("test.txt");
```

gives you a file object with this name in the current directory. (The "current directory" is the current directory of the process that executes the virtual machine. If you launched the virtual machine from the command line, it is the directory from which you started the `java` executable.)

Caution



Because the backslash character is the escape character in Java strings, be sure to use `\\" for Windows-style path names ("C:\\Windows\\win.ini"). In Windows, you can also use a single forward slash ("C:/Windows/win.ini") because most Windows file handling system calls will interpret forward slashes as file separators. However, this is not recommended—the behavior of the Windows system functions is subject to change, and on other operating systems, the file separator might be different. Instead, for portable programs, you should use the file separator character for the platform on which your program runs. It is stored in the constant string File.separator.`

A call to this constructor does not create a file with this name if it doesn't exist. Actually, creating a file from a `File` object is done with one of the stream class constructors or the `createNewFile` method in the `File` class. The `createNewFile` method only creates a file if no file with that name exists, and it returns a `boolean` to tell you whether it was successful.

On the other hand, once you have a `File` object, the `exists` method in the `File` class tells you whether a file exists with that name. For example, the following trial program would almost certainly print "false" on anyone's machine, and yet it can print out a path name to this nonexistent file.

```
import java.io.*;

public class Test
{
    public static void main(String args[])
    {
        File f = new File("afilethatprobablydoesntexist");
        System.out.println(f.getAbsolutePath());
        System.out.println(f.exists());
    }
}
```

There are two other constructors for `File` objects:

```
File(String path, String name)
```

which creates a `File` object with the given name in the directory specified by the `path` parameter. (If the `path` parameter is `null`, this constructor creates a `File` object, using the current directory.)

Finally, you can use an existing `File` object in the constructor:

```
File(File dir, String name)
```

where the `File` object represents a directory and, as before, if `dir` is `null`, the constructor creates a `File` object in the current directory.

Somewhat confusingly, a `File` object can represent either a file or a directory (perhaps because the operating system that the Java designers were most familiar with happens to implement directories as files). You use the `isDirectory` and `isFile` methods to tell whether the file object represents a file or a directory. This is surprising—in an object-oriented system, you might have expected a separate `Directory` class, perhaps extending the `File` class.

To make an object representing a directory, you simply supply the directory name in the `File` constructor:

```
File tempDir = new File(File.separator + "temp");
```

If this directory does not yet exist, you can create it with the `mkdir` method:

```
tempDir.mkdir();
```

If a file object represents a directory, use `list()` to get an array of the file names in that directory. The program in [Listing 1-6](#) uses all these methods to print out the directory substructure of whatever path is entered on the command line. (It would be easy enough to change this program into a utility class that returns a list of the subdirectories for further processing.)

Tip

 Always use `File` objects, not strings, when manipulating file or directory names. For example, the `equals` method of the `File` class knows that some file systems are not case significant and that a trailing / in a directory name doesn't matter.

[Listing 1-6. FindDirectories.java](#)

Code View:

```
1. import java.io.*;
```

```
2.
3. /**
4. * @version 1.00 05 Sep 1997
5. * @author Gary Cornell
6. */
7. public class FindDirectories
8. {
9.     public static void main(String[] args)
10.    {
11.        // if no arguments provided, start at the parent directory
12.        if (args.length == 0) args = new String[] { ".." };
13.
14.        try
15.        {
16.            File pathName = new File(args[0]);
17.            String[] fileNames = pathName.list();
18.
19.            // enumerate all files in the directory
20.            for (int i = 0; i < fileNames.length; i++)
21.            {
22.                File f = new File(pathName.getPath(), fileNames[i]);
23.
24.                // if the file is again a directory, call the main method recursively
25.                if (f.isDirectory())
26.                {
27.                    System.out.println(f.getCanonicalPath());
28.                    main(new String[] { f.getPath() });
29.                }
30.            }
31.        }
32.        catch (IOException e)
33.        {
34.            e.printStackTrace();
35.        }
36.    }
37. }
```

Rather than listing all files in a directory, you can use a `FileNameFilter` object as a parameter to the `list` method to narrow down the list. These objects are simply instances of a class that satisfies the `FilenameFilter` interface.

All a class needs to do to implement the `FilenameFilter` interface is define a method called `accept`. Here is an example of a simple `FilenameFilter` class that allows only files with a specified extension:

```
public class ExtensionFilter implements FilenameFilter
{
    public ExtensionFilter(String ext)
    {
        extension = "." + ext;
    }

    public boolean accept(File dir, String name)
    {
        return name.endsWith(extension);
    }

    private String extension;
}
```

When writing portable programs, it is a challenge to specify file names with subdirectories. As we mentioned earlier, it turns out that you can use a forward slash (the UNIX separator) as the directory separator in Windows as well, but other operating systems might not permit this, so we don't recommend using a forward slash.

Caution

 If you do use forward slashes as directory separators in Windows when constructing a `File` object, the `getAbsolutePath` method returns a file name that contains forward slashes, which will look strange to Windows users. Instead, use the `getCanonicalPath` method—it replaces the forward slashes with backslashes.

It is much better to use the information about the current directory separator that the `File` class stores in a static instance field called `separator`. In a Windows environment, this is a backslash (\); in a UNIX environment, it is a forward slash (/). For example:

```
File foo = new File("Documents" + File.separator + "data.txt")
```

Of course, if you use the second alternate version of the `File` constructor

```
File foo = new File("Documents", "data.txt")
```

then the constructor will supply the correct separator.

The API notes that follow give you what we think are the most important remaining methods of the `File` class; their use should be straightforward.



java.io.File 1.0

- `boolean canRead()`
- `boolean canWrite()`
- `boolean canExecute() 6`

indicates whether the file is readable, writable, or executable.

- `boolean setReadable(boolean state, boolean ownerOnly) 6`
- `boolean setWritable(boolean state, boolean ownerOnly) 6`
- `boolean setExecutable(boolean state, boolean ownerOnly) 6`

sets the readable, writable, or executable state of this file. If `ownerOnly` is true, the state is set for the file's owner only. Otherwise, it is set for everyone. The methods return true if setting the state succeeded.

- `static boolean createTempFile(String prefix, String suffix) 1.2`
- `static boolean createTempFile(String prefix, String suffix, File directory) 1.2`

creates a temporary file in the system's default temp directory or the given directory, using the given prefix and suffix to generate the temporary name.

Parameters:	<code>prefix</code>	A prefix string that is at least three characters long
	<code>suffix</code>	An optional suffix. If null, .tmp is used
	<code>directory</code>	The directory in which the file is created. If it is null, the file is created in the current working directory

- `boolean delete()`

tries to delete the file. Returns true if the file was deleted, false otherwise.

- `void deleteOnExit()`

requests that the file be deleted when the virtual machine shuts down.

- `boolean exists()`

returns true if the file or directory exists; false otherwise.

- `String getAbsolutePath()`

returns a string that contains the absolute path name. Tip: Use `getCanonicalPath` instead.
- `File getCanonicalFile() 1.2`

returns a `File` object that contains the canonical path name for the file. In particular, redundant ". ." directories are removed, the correct directory separator is used, and the capitalization preferred by the underlying file system is obtained.
- `String getCanonicalPath() 1.1`

returns a string that contains the canonical path name. In particular, redundant ". ." directories are removed, the correct directory separator is used, and the capitalization preferred by the underlying file system is obtained.
- `String getName()`

returns a string that contains the file name of the `File` object (does not include path information).
- `String getParent()`

returns a string that contains the name of the parent of this `File` object. If this `File` object is a file, then the parent is the directory containing it. If it is a directory, then the parent is the parent directory or `null` if there is no parent directory.
- `File getParentFile() 1.2`

returns a `File` object for the parent of this `File` directory. See `getParent` for a definition of "parent."
- `String getPath()`

returns a string that contains the path name of the file.
- `boolean isDirectory()`

returns `true` if the `File` represents a directory; `false` otherwise.
- `boolean isFile()`

returns `true` if the `File` object represents a file as opposed to a directory or a device.
- `boolean isHidden() 1.2`

returns `true` if the `File` object represents a hidden file or directory.
- `long lastModified()`

returns the time the file was last modified (counted in milliseconds since Midnight January 1, 1970 GMT), or 0 if the file does not exist. Use the `Date(long)` constructor to convert this value to a date.
- `long length()`

returns the length of the file in bytes, or 0 if the file does not exist.
- `String[] list()`

returns an array of strings that contain the names of the files and directories contained by this `File` object, or `null` if this `File` was not representing a directory.

- `String[] list(FilenameFilter filter)`

returns an array of the names of the files and directories contained by this `File` that satisfy the filter, or `null` if none exist.
- `File[] listFiles() 1.2`

returns an array of `File` objects corresponding to the files and directories contained by this `File` object, or `null` if this `File` was not representing a directory.
- `File[] listFiles(FilenameFilter filter) 1.2`

returns an array of `File` objects for the files and directories contained by this `File` that satisfy the filter, or `null` if none exist.
- `static File[] listRoots() 1.2`

returns an array of `File` objects corresponding to all the available file roots. (For example, on a Windows system, you get the `File` objects representing the installed drives, both local drives and mapped network drives. On a UNIX system, you simply get "/".)
- `boolean createNewFile() 1.2`

atomically makes a new file whose name is given by the `File` object if no file with that name exists. That is, the checking for the file name and the creation are not interrupted by other file system activity. Returns `true` if the method created the file.
- `boolean mkdir()`

makes a subdirectory whose name is given by the `File` object. Returns `true` if the directory was successfully created; `false` otherwise.
- `boolean mkdirs()`

unlike `mkdir`, creates the parent directories if necessary. Returns `false` if any of the necessary directories could not be created.
- `boolean renameTo(File newName)`

returns `true` if the name was changed; `false` otherwise.
- `boolean setLastModified(long time) 1.2`

sets the last modified time of the file. Returns `true` if successful, `false` otherwise. `time` is a long integer representing the number of milliseconds since Midnight January 1, 1970, GMT. Use the `getTime` method of the `Date` class to calculate this value.
- `boolean setReadOnly() 1.2`

sets the file to be read-only. Returns `true` if successful, `false` otherwise.
- `URL toURL() 1.2`

converts the `File` object to a file `URL`.
- `long getTotalSpace() 6`
- `long getFreeSpace() 6`
- `long getUsableSpace() 6`

gets the total size, number of unallocated bytes, and number of available bytes on the partition described by this `File` object. If this `File` object does not describe a

partition, the methods return 0.

API

java.io.FilenameFilter 1.0

- boolean accept(File dir, String name)

should be defined to return `true` if the file matches the filter criterion.

Parameters: `dir` A `File` object representing the directory that contains the file
 `name` The name of the file



New I/O

Java SE 1.4 introduced a number of features for improved input/output processing, collectively called the "new I/O," in the `java.nio` package. (Of course, the "new" moniker is somewhat regrettable because, a few years down the road, the package wasn't new any longer.)

The package includes support for the following features:

- Character set encoders and decoders
- Nonblocking I/O
- Memory-mapped files
- File locking

We already covered character encoding and decoding in the section "[Character Sets](#)" on page [19](#). Nonblocking I/O is discussed in [Chapter 3](#) because it is particularly important when communicating across a network. In the following sections, we examine memory-mapped files and file locking in detail.

Memory-Mapped Files

Most operating systems can take advantage of the virtual memory implementation to "map" a file, or a region of a file, into memory. Then the file can be accessed as if it were an in-memory array, which is much faster than the traditional file operations.

At the end of this section, you can find a program that computes the CRC32 checksum of a file, using traditional file input and a memory-mapped file. On one machine, we got the timing data shown in [Table 1-6](#) when computing the checksum of the 37-Mbyte file `rt.jar` in the `jre/lib` directory of the JDK.

Table 1-6. Timing Data for File Operations

Method	Time
Plain Input Stream	110 seconds
Buffered Input Stream	9.9 seconds
Random Access File	162 seconds
Memory Mapped file	7.2 seconds

As you can see, on this particular machine, memory mapping is a bit faster than using buffered sequential input and dramatically faster than using a `RandomAccessFile`.

Of course, the exact values will differ greatly from one machine to another, but it is obvious that the performance gain can be substantial if you need to use random access. For sequential reading of files of moderate size, on the other hand, there is no reason to use memory mapping.

The `java.nio` package makes memory mapping quite simple. Here is what you do.

First, get a channel from the file. A channel is an abstraction for disk files that lets you access operating system features such as memory mapping, file locking, and fast data transfers between files. You get a channel by calling the `getChannel` method that has been added to the `FileInputStream`, `FileOutputStream`, and `RandomAccessFile` class.

```
FileInputStream in = new FileInputStream(. . .);
FileChannel channel = in.getChannel();
```

Then you get a `MappedByteBuffer` from the channel by calling the `map` method of the `FileChannel` class. You specify the area of the file that you want to map and a mapping mode. Three modes are supported:

- `FileChannel.MapMode.READ_ONLY`: The resulting buffer is read-only. Any attempt to write to the buffer results in a `ReadOnlyBufferException`.
- `FileChannel.MapMode.READ_WRITE`: The resulting buffer is writable, and the changes will be written back to the file at some time. Note that other programs that have mapped the same file might not see those changes immediately. The exact behavior of simultaneous file mapping by multiple programs is operating-system dependent.
- `FileChannel.MapMode.PRIVATE`: The resulting buffer is writable, but any changes are private to this buffer and are not propagated to the file.

Once you have the buffer, you can read and write data, using the methods of the `ByteBuffer` class and the `Buffer` superclass.

Buffers support both sequential and random data access. A buffer has a position that is advanced by `get` and `put` operations. For example, you can sequentially traverse all bytes in the buffer as

```
while (buffer.hasRemaining())
{
    byte b = buffer.get();
    . .
}
```

Alternatively, you can use random access:

```
for (int i = 0; i < buffer.limit(); i++)
{
    byte b = buffer.get(i);
    . .
}
```

You can also read and write arrays of bytes with the methods

```
get(byte[] bytes)
get(byte[], int offset, int length)
```

Finally, there are methods

```
getInt
getLong
getShort
getChar
getFloat
getDouble
```

to read primitive type values that are stored as binary values in the file. As we already mentioned, Java uses big-endian ordering for binary data. However, if you need to process a file containing binary numbers in little-endian order, simply call

```
buffer.order(ByteOrder.LITTLE_ENDIAN);
```

To find out the current byte order of a buffer, call

```
ByteOrder b = buffer.order()
```

Caution



This pair of methods does not use the `set/get` naming convention.

To write numbers to a buffer, use one of the methods

```
.putInt  
putLong  
putShort  
putChar  
putFloat  
putDouble
```

[Listing 1-7](#) computes the 32-bit cyclic redundancy checksum (CRC32) of a file. That quantity is a checksum that is often used to determine whether a file has been corrupted. Corruption of a file makes it very likely that the checksum has changed. The `java.util.zip` package contains a class `CRC32` that computes the checksum of a sequence of bytes, using the following loop:

```
CRC32 crc = new CRC32();  
while (more bytes)  
    crc.update(next byte)  
long checksum = crc.getValue();
```

Note



For a nice explanation of the CRC algorithm, see <http://www.relisoft.com/Science/CrcMath.html>.

The details of the CRC computation are not important. We just use it as an example of a useful file operation.

Run the program as

```
java NIOTest filename
```

[Listing 1-7. NIOTest.java](#)

Code View:

```
1. import java.io.*;  
2. import java.nio.*;  
3. import java.nio.channels.*;  
4. import java.util.zip.*;  
5.  
6. /**  
7.  * This program computes the CRC checksum of a file. <br>  
8.  * Usage: java NIOTest filename  
9.  * @version 1.01 2004-05-11  
10. * @author Cay Horstmann  
11. */  
12. public class NIOTest  
13. {  
14.     public static long checksumInputStream(String filename) throws IOException  
15.     {  
16.         InputStream in = new FileInputStream(filename);  
17.         CRC32 crc = new CRC32();  
18.  
19.         int c;  
20.         while ((c = in.read()) != -1)  
21.             crc.update(c);  
22.         return crc.getValue();  
23.     }  
24.  
25.     public static long checksumBufferedInputStream(String filename) throws IOException  
26.     {
```

```
27.     InputStream in = new BufferedInputStream(new FileInputStream(filename));
28.     CRC32 crc = new CRC32();
29.
30.     int c;
31.     while ((c = in.read()) != -1)
32.         crc.update(c);
33.     return crc.getValue();
34. }
35.
36. public static long checksumRandomAccessFile(String filename) throws IOException
37. {
38.     RandomAccessFile file = new RandomAccessFile(filename, "r");
39.     long length = file.length();
40.     CRC32 crc = new CRC32();
41.
42.     for (long p = 0; p < length; p++)
43.     {
44.         file.seek(p);
45.         int c = file.readByte();
46.         crc.update(c);
47.     }
48.     return crc.getValue();
49. }
50.
51. public static long checksumMappedFile(String filename) throws IOException
52. {
53.     FileInputStream in = new FileInputStream(filename);
54.     FileChannel channel = in.getChannel();
55.
56.     CRC32 crc = new CRC32();
57.     int length = (int) channel.size();
58.     MappedByteBuffer buffer = channel.map(FileChannel.MapMode.READ_ONLY, 0, length);
59.
60.     for (int p = 0; p < length; p++)
61.     {
62.         int c = buffer.get(p);
63.         crc.update(c);
64.     }
65.     return crc.getValue();
66. }
67.
68. public static void main(String[] args) throws IOException
69. {
70.     System.out.println("Input Stream:");
71.     long start = System.currentTimeMillis();
72.     long crcValue = checksumInputStream(args[0]);
73.     long end = System.currentTimeMillis();
74.     System.out.println(Long.toHexString(crcValue));
75.     System.out.println((end - start) + " milliseconds");
76.
77.     System.out.println("Buffered Input Stream:");
78.     start = System.currentTimeMillis();
79.     crcValue = checksumBufferedInputStream(args[0]);
80.     end = System.currentTimeMillis();
81.     System.out.println(Long.toHexString(crcValue));
82.     System.out.println((end - start) + " milliseconds");
83.
84.     System.out.println("Random Access File:");
85.     start = System.currentTimeMillis();
86.     crcValue = checksumRandomAccessFile(args[0]);
87.     end = System.currentTimeMillis();
88.     System.out.println(Long.toHexString(crcValue));
89.     System.out.println((end - start) + " milliseconds");
90.
91.     System.out.println("Mapped File:");
92.     start = System.currentTimeMillis();
93.     crcValue = checksumMappedFile(args[0]);
94.     end = System.currentTimeMillis();
95.     System.out.println(Long.toHexString(crcValue));
96.     System.out.println((end - start) + " milliseconds");
97. }
98. }
```

API**java.io.FileInputStream 1.0**

- `FileChannel getChannel()` 1.4

returns a channel for accessing this stream.

API**java.io.FileOutputStream 1.0**

- `FileChannel getChannel()` 1.4

returns a channel for accessing this stream.

API**java.io.RandomAccessFile 1.0**

- `FileChannel getChannel()` 1.4

returns a channel for accessing this file.

API**java.nio.channels.FileChannel 1.4**

- `MappedByteBuffer map(FileChannel.MapMode mode, long position, long size)`

maps a region of the file to memory.

Parameters:	<code>mode</code>	One of the constants <code>READ_ONLY</code> , <code>READ_WRITE</code> , or <code>PRIVATE</code> in the <code>FileChannel.MapMode</code> class
	<code>position</code>	The start of the mapped region
	<code>size</code>	The size of the mapped region

API**java.nio.Buffer 1.4**

- `boolean hasRemaining()`

returns `true` if the current buffer position has not yet reached the buffer's limit position.

- `int limit()`

returns the limit position of the buffer; that is, the first position at which no more values are available.

API**java.nio.ByteBuffer 1.4**

- `byte get()`

gets a byte from the current position and advances the current position to the next byte.

- `byte get(int index)`
gets a byte from the specified index.
- `ByteBuffer put(byte b)`
puts a byte to the current position and advances the current position to the next byte.
Returns a reference to this buffer.
- `ByteBuffer put(int index, byte b)`
puts a byte at the specified index. Returns a reference to this buffer.
- `ByteBuffer get(byte[] destination)`
- `ByteBuffer get(byte[] destination, int offset, int length)`
fills a byte array, or a region of a byte array, with bytes from the buffer, and
advances the current position by the number of bytes read. If not enough bytes
remain in the buffer, then no bytes are read, and a `BufferUnderflowException` is
thrown. Returns a reference to this buffer.

Parameters:

<code>destination</code>	The byte array to be filled
<code>offset</code>	The offset of the region to be filled
<code>length</code>	The length of the region to be filled
- `ByteBuffer put(byte[] source)`
- `ByteBuffer put(byte[] source, int offset, int length)`
puts all bytes from a byte array, or the bytes from a region of a byte array, into the
buffer, and advances the current position by the number of bytes read. If not enough
bytes remain in the buffer, then no bytes are written, and a
`BufferOverflowException` is thrown. Returns a reference to this buffer.

Parameters:

<code>source</code>	The byte array to be written
<code>offset</code>	The offset of the region to be written
<code>length</code>	The length of the region to be written
- `Xxx getXxx()`
- `Xxx getXxx(int index)`
- `ByteBuffer putXxx(XXX value)`
- `ByteBuffer putXxx(int index, XXX value)`
gets or puts a binary number. Xxx is one of `Int`, `Long`, `Short`, `Char`, `Float`, or
`Double`.
- `ByteBuffer order(ByteOrder order)`
- `ByteOrder order()`
sets or gets the byte order. The value for `order` is one of the constants `BIG_ENDIAN`
or `LITTLE_ENDIAN` of the `ByteOrder` class.

The Buffer Data Structure

When you use memory mapping, you make a single buffer that spans the entire file, or the area of the file in which you are interested. You can also use buffers to read and write more modest chunks of information.

In this section, we briefly describe the basic operations on `Buffer` objects. A buffer is an array of values of the same type. The `Buffer` class is an abstract class with concrete subclasses `ByteBuffer`, `CharBuffer`, `DoubleBuffer`, `FloatBuffer`, `IntBuffer`, `LongBuffer`, and `ShortBuffer`.

Note

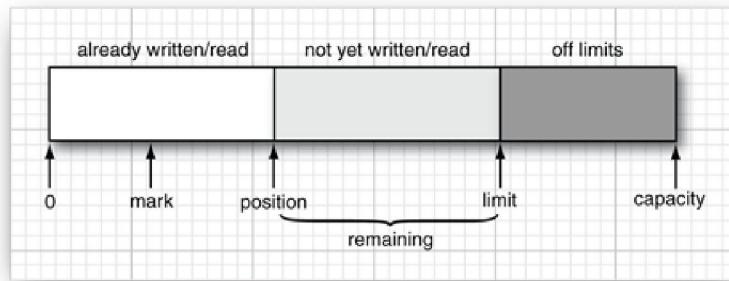


The `StringBuffer` class is not related to these buffers.

In practice, you will most commonly use `ByteBuffer` and `CharBuffer`. As shown in [Figure 1-11](#), a buffer has

- A capacity that never changes.
- A position at which the next value is read or written.
- A limit beyond which reading and writing is meaningless.
- Optionally, a mark for repeating a read or write operation.

Figure 1-11. A buffer



These values fulfill the condition

$$0 \leq \text{mark} \leq \text{position} \leq \text{limit} \leq \text{capacity}$$

The principal purpose for a buffer is a "write, then read" cycle. At the outset, the buffer's position is 0 and the limit is the capacity. Keep calling `put` to add values to the buffer. When you run out of data or you reach the capacity, it is time to switch to reading.

Call `flip` to set the limit to the current position and the position to 0. Now keep calling `get` while the `remaining` method (which returns `limit - position`) is positive. When you have read all values in the buffer, call `clear` to prepare the buffer for the next writing cycle. The `clear` method resets the position to 0 and the limit to the capacity.

If you want to reread the buffer, use `rewind` or `mark/reset`—see the API notes for details.



java.nio.Buffer 1.4

- `Buffer clear()`

prepares this buffer for writing by setting the position to 0 and the limit to the capacity; returns `this`.

- `Buffer flip()`

prepares this buffer for reading by setting the limit to the position and the position to 0; returns `this`.

- `Buffer rewind()`

prepares this buffer for rereading the same values by setting the position to 0 and leaving the limit unchanged; returns `this`.

- `Buffer mark()`

sets the mark of this buffer to the position; returns `this`.

- `Buffer reset()`

sets the position of this buffer to the mark, thus allowing the marked portion to be read or written again; returns `this`.

- `int remaining()`

returns the remaining number of readable or writable values; that is, the difference between limit and position.

- `int position()`

returns the position of this buffer.

- `int capacity()`

returns the capacity of this buffer.



java.nio.CharBuffer 1.4

- `char get()`

- `CharBuffer get(char[] destination)`

- `CharBuffer get(char[] destination, int offset, int length)`

gets one `char` value, or a range of `char` values, starting at the buffer's position and moving the position past the characters that were read. The last two methods return `this`.

- `CharBuffer put(char c)`

- `CharBuffer put(char[] source)`

- `CharBuffer put(char[] source, int offset, int length)`

- `CharBuffer put(String source)`

- `CharBuffer put(CharBuffer source)`

puts one `char` value, or a range of `char` values, starting at the buffer's position and advancing the position past the characters that were written. When reading from a `CharBuffer`, all remaining characters are read. All methods return `this`.

- `CharBuffer read(CharBuffer destination)`

gets `char` values from this buffer and puts them into the destination until the destination's limit is reached. Returns `this`.

File Locking

Consider a situation in which multiple simultaneously executing programs need to modify the same file. Clearly, the programs need to communicate in some way, or the file can easily become damaged.

File locks control access to a file or a range of bytes within a file. However, file locking varies greatly among operating

systems, which explains why file locking capabilities were absent from prior versions of the JDK.

File locking is not all that common in application programs. Many applications use a database for data storage, and the database has mechanisms for resolving concurrent access. If you store information in flat files and are worried about concurrent access, you might find it simpler to start using a database rather than designing complex file locking schemes.

Still, there are situations in which file locking is essential. Suppose your application saves a configuration file with user preferences. If a user invokes two instances of the application, it could happen that both of them want to write the configuration file at the same time. In that situation, the first instance should lock the file. When the second instance finds the file locked, it can decide to wait until the file is unlocked or simply skip the writing process.

To lock a file, call either the `lock` or `tryLock` method of the `FileChannel` class:

```
FileLock lock = channel.lock();
```

or

```
FileLock lock = channel.tryLock();
```

The first call blocks until the lock becomes available. The second call returns immediately, either with the lock or `null` if the lock is not available. The file remains locked until the channel is closed or the `release` method is invoked on the lock.

You can also lock a portion of the file with the call

```
FileLock lock(long start, long size, boolean exclusive)
```

or

```
FileLock tryLock(long start, long size, boolean exclusive)
```

The `exclusive` flag is `true` to lock the file for both reading and writing. It is `false` for a shared lock, which allows multiple processes to read from the file, while preventing any process from acquiring an exclusive lock. Not all operating systems support shared locks. You may get an exclusive lock even if you just asked for a shared one. Call the `isShared` method of the `FileLock` class to find out which kind you have.

Note



If you lock the tail portion of a file and the file subsequently grows beyond the locked portion, the additional area is not locked. To lock all bytes, use a size of `Long.MAX_VALUE`.

Keep in mind that file locking is system dependent. Here are some points to watch for:

- On some systems, file locking is merely advisory. If an application fails to get a lock, it may still write to a file that another application has currently locked.
- On some systems, you cannot simultaneously lock a file and map it into memory.
- File locks are held by the entire Java virtual machine. If two programs are launched by the same virtual machine (such as an applet or application launcher), then they can't each acquire a lock on the same file. The `lock` and `tryLock` methods will throw an `OverlappingFileLockException` if the virtual machine already holds another overlapping lock on the same file.
- On some systems, closing a channel releases all locks on the underlying file held by the Java virtual machine. You should therefore avoid multiple channels on the same locked file.
- Locking files on a networked file system is highly system dependent and should probably be avoided.



java.nio.channels.FileChannel 1.4

- `FileLock lock()`

acquires an exclusive lock on the entire file. This method blocks until the lock is acquired.

- `FileLock tryLock()`

acquires an exclusive lock on the entire file, or returns `null` if the lock cannot be acquired.

- `FileLock lock(long position, long size, boolean shared)`

- `FileLock tryLock(long position, long size, boolean shared)`

acquires a lock on a region of the file. The first method blocks until the lock is acquired, and the second method returns `null` if the lock cannot be acquired.

Parameters:

<code>position</code>	The start of the region to be locked
<code>size</code>	The size of the region to be locked
<code>shared</code>	<code>true</code> for a shared lock, <code>false</code> for an exclusive lock



java.nio.channels.FileLock 1.4

- `void release()`

releases this lock.



Regular Expressions

Regular expressions are used to specify string patterns. You can use regular expressions whenever you need to locate strings that match a particular pattern. For example, one of our sample programs locates all hyperlinks in an HTML file by looking for strings of the pattern ``.

Of course, for specifying a pattern, the `...` notation is not precise enough. You need to specify precisely what sequence of characters is a legal match. You need to use a special syntax whenever you describe a pattern.

Here is a simple example. The regular expression

`[Jj]ava.+`

matches any string of the following form:

- The first letter is a `J` or `j`.
- The next three letters are `ava`.
- The remainder of the string consists of one or more arbitrary characters.

For example, the string "javanese" matches the particular regular expression, but the string "Core Java" does not.

As you can see, you need to know a bit of syntax to understand the meaning of a regular expression. Fortunately, for most purposes, a small number of straightforward constructs are sufficient.

- A character class is a set of character alternatives, enclosed in brackets, such as `[Jj]`, `[0-9]`, `[A-Za-z]`, or `[^0-9]`. Here the `-` denotes a range (all characters whose Unicode value falls between the two bounds), and `^` denotes the complement (all characters except the ones specified).

- There are many predefined character classes such as `\d` (digits) or `\p{sc}` (Unicode currency symbol). See [Tables 1-7](#) and [1-8](#).

Table 1-7. Regular Expression Syntax

Syntax	Explanation
Characters	
<code>c</code>	The character c
<code>\unnnn, \xnn, \on, \onn, \0nnn</code>	The code unit with the given hex or octal value
<code>\t, \n, \r, \f, \a, \e</code>	The control characters tab, newline, return, form feed, alert, and escape
<code>\cc</code>	The control character corresponding to the character c
Character Classes	
<code>[C₁C₂...]</code>	Any of the characters represented by C ₁ , C ₂ , ... The C _i are characters, character ranges (c ₁ -c ₂), or character classes
<code>[^...]</code>	Complement of character class
<code>[. . . & & . . .]</code>	Intersection of two character classes
Predefined Character Classes	
<code>.</code>	Any character except line terminators (or any character if the DOTALL flag is set)
<code>\d</code>	A digit [0-9]
<code>\D</code>	A nondigit [^0-9]
<code>\s</code>	A whitespace character [\t\n\r\f\x0B]
<code>\S</code>	A nonwhitespace character
<code>\w</code>	A word character [a-zA-Z0-9_]
<code>\W</code>	A nonword character
<code>\p{name}</code>	A named character class—see Table 1-8
<code>\P{name}</code>	The complement of a named character class
Boundary Matchers	
<code>^ \$</code>	Beginning, end of input (or beginning, end of line in multiline mode)
<code>\b</code>	A word boundary
<code>\B</code>	A nonword boundary
<code>\A</code>	Beginning of input
<code>\z</code>	End of input
<code>\Z</code>	End of input except final line terminator
<code>\G</code>	End of previous match
Quantifiers	
<code>X?</code>	Optional X
<code>X*</code>	X, 0 or more times
<code>X+</code>	X, 1 or more times
<code>X{n} X{n,} X{n,m}</code>	X n times, at least n times, between n and m times
Quantifier Suffixes	
<code>?</code>	Turn default (greedy) match into reluctant match
<code>+</code>	Turn default (greedy) match into possessive match

Set Operations

XY	Any string from X, followed by any string from Y
X Y	Any string from X or Y
Grouping	
(X)	Capture the string matching X as a group
\n	The match of the nth group
Escapes	
\c	The character c (must not be an alphabetic character)
\Q . . . \E	Quote . . . verbatim
(? . . .)	Special construct—see API notes of <code>Pattern</code> class

Table 1-8. Predefined Character Class Names

Character Class Name	Explanation
Lower	ASCII lower case [a-z]
Upper	ASCII upper case [A-Z]
Alpha	ASCII alphabetic [A-Za-z]
Digit	ASCII digits [0-9]
Alnum	ASCII alphabetic or digit [A-Za-z0-9]
XDigit	Hex digits [0-9A-Fa-f]
Print or Graph	Printable ASCII character [\x21-\x7E]
Punct	ASCII nonalpha or digit [\p{Print} && \P{Alnum}]
ASCII	All ASCII [\x00-\x7F]
Cntrl	ASCII Control character [\x00-\x1F]
Blank	Space or tab [\t]
Space	Whitespace [\t\n\r\f\0x0B]
javaLowerCase	Lower case, as determined by <code>Character.isLowerCase()</code>
javaUpperCase	Upper case, as determined by <code>Character.isUpperCase()</code>
javaWhitespace	Whitespace, as determined by <code>Character.isWhitespace()</code>
javaMirrored	Mirrored, as determined by <code>Character.isMirrored()</code>
InBlock	Block is the name of a Unicode character block, with spaces removed, such as <code>BasicLatin</code> or <code>Mongolian</code> . See http://www.unicode.org for a list of block names.
Category or InCategory	Category is the name of a Unicode character category such as L (letter) or Sc (currency symbol). See http://www.unicode.org for a list of category names.

- Most characters match themselves, such as the `a` characters in the preceding example.
- The `.` symbol matches any character (except possibly line terminators, depending on flag settings).
- Use `\` as an escape character, for example `\.` matches a period and `\\\` matches a backslash.
- `^` and `$` match the beginning and end of a line, respectively.
- If X and Y are regular expressions, then XY means "any match for X followed by a match for Y". X | Y means "any match for X or Y".

- You can apply quantifiers X+ (1 or more), X* (0 or more), and X? (0 or 1) to an expression X.
- By default, a quantifier matches the largest possible repetition that makes the overall match succeed. You can modify that behavior with suffixes ? (reluctant or stingy match—match the smallest repetition count) and + (possessive or greedy match—match the largest count even if that makes the overall match fail).

For example, the string `cab` matches `[a-z]*ab` but not `[a-z]*+ab`. In the first case, the expression `[a-z]*` only matches the character `c`, so that the characters `ab` match the remainder of the pattern. But the greedy version `[a-z]*+` matches the characters `cab`, leaving the remainder of the pattern unmatched.

- You can use groups to define subexpressions. Enclose the groups in `()`; for example, `([+-]?) ([0-9]+)`. You can then ask the pattern matcher to return the match of each group or to refer back to a group with `\n`, where `n` is the group number (starting with `\1`).

For example, here is a somewhat complex but potentially useful regular expression—it describes decimal or hexadecimal integers:

```
[+-]?[0-9]+\|0[XX][0-9A-Fa-f]+
```

Unfortunately, the expression syntax is not completely standardized between the various programs and libraries that use regular expressions. Although there is consensus on the basic constructs, there are many maddening differences in the details. The Java regular expression classes use a syntax that is similar to, but not quite the same as, the one used in the Perl language. [Table 1-7](#) shows all constructs of the Java syntax. For more information on the regular expression syntax, consult the API documentation for the `Pattern` class or the book *Mastering Regular Expressions* by Jeffrey E. F. Friedl (O'Reilly and Associates, 1997).

The simplest use for a regular expression is to test whether a particular string matches it. Here is how you program that test in Java. First construct a `Pattern` object from the string denoting the regular expression. Then get a `Matcher` object from the pattern, and call its `matches` method:

```
Pattern pattern = Pattern.compile(patternString);
Matcher matcher = pattern.matcher(input);
if (matcher.matches()) . . .
```

The input of the matcher is an object of any class that implements the `CharSequence` interface, such as a `String`, `StringBuilder`, or `CharBuffer`.

When compiling the pattern, you can set one or more flags, for example,

```
Pattern pattern = Pattern.compile(patternString,
    Pattern.CASE_INSENSITIVE + Pattern.UNICODE_CASE);
```

The following six flags are supported:

- `CASE_INSENSITIVE`: Match characters independently of the letter case. By default, this flag takes only US ASCII characters into account.
- `UNICODE_CASE`: When used in combination with `CASE_INSENSITIVE`, use Unicode letter case for matching.
- `MULTILINE`: `^` and `$` match the beginning and end of a line, not the entire input.
- `UNIX_LINES`: Only '`\n`' is recognized as a line terminator when matching `^` and `$` in multiline mode.
- `DOTALL`: When using this flag, the `.` symbol matches all characters, including line terminators.
- `CANON_EQ`: Takes canonical equivalence of Unicode characters into account. For example, `u` followed by `''` (diaeresis) matches `ü`.

If the regular expression contains groups, then the `Matcher` object can reveal the group boundaries. The methods

```
int start(int groupIndex)
int end(int groupIndex)
```

yield the starting index and the past-the-end index of a particular group.

You can simply extract the matched string by calling

```
String group(int groupIndex)
```

Group 0 is the entire input; the group index for the first actual group is 1. Call the `groupCount` method to get the total group count.

Nested groups are ordered by the opening parentheses. For example, given the pattern

```
((1?[0-9]):([0-5][0-9]))[ap]m
```

and the input

11:59am

the matcher reports the following groups

Group Index	Start	End	String
0	0	7	11:59am
1	0	5	11:59
2	0	2	11
3	3	5	59

[Listing 1-8](#) prompts for a pattern, then for strings to match. It prints out whether or not the input matches the pattern. If the input matches and the pattern contains groups, then the program prints the group boundaries as parentheses, such as

```
((11):(59))am
```

Listing 1-8. RegexTest.java

Code View:

```
1. import java.util.*;
2. import java.util.regex.*;
3.
4. /**
5. * This program tests regular expression matching.
6. * Enter a pattern and strings to match, or hit Cancel
7. * to exit. If the pattern contains groups, the group
8. * boundaries are displayed in the match.
9. * @version 1.01 2004-05-11
10. * @author Cay Horstmann
11. */
12. public class RegExTest
13. {
14.     public static void main(String[] args)
15.     {
16.         Scanner in = new Scanner(System.in);
17.         System.out.println("Enter pattern: ");
18.         String patternString = in.nextLine();
19.
20.         Pattern pattern = null;
21.         try
22.         {
23.             pattern = Pattern.compile(patternString);
24.         }
25.         catch (PatternSyntaxException e)
26.         {
27.             System.out.println("Pattern syntax error");
28.             System.exit(1);
29.         }
}
```

```

30.
31.     while (true)
32.     {
33.         System.out.println("Enter string to match: ");
34.         String input = in.nextLine();
35.         if (input == null || input.equals("")) return;
36.         Matcher matcher = pattern.matcher(input);
37.         if (matcher.matches())
38.         {
39.             System.out.println("Match");
40.             int g = matcher.groupCount();
41.             if (g > 0)
42.             {
43.                 for (int i = 0; i < input.length(); i++)
44.                 {
45.                     for (int j = 1; j <= g; j++)
46.                         if (i == matcher.start(j))
47.                             System.out.print('(');
48.                         System.out.print(input.charAt(i));
49.                         for (int j = 1; j <= g; j++)
50.                             if (i + 1 == matcher.end(j))
51.                                 System.out.print(')');
52.                 }
53.                 System.out.println();
54.             }
55.         }
56.         else
57.             System.out.println("No match");
58.     }
59. }
60. }
```

Usually, you don't want to match the entire input against a regular expression, but you want to find one or more matching substrings in the input. Use the `find` method of the `Matcher` class to find the next match. If it returns `true`, use the `start` and `end` methods to find the extent of the match.

```

while (matcher.find())
{
    int start = matcher.start();
    int end = matcher.end();
    String match = input.substring(start, end);
    . .
}
```

[Listing 1-9](#) puts this mechanism to work. It locates all hypertext references in a web page and prints them. To run the program, supply a URL on the command line, such as

```
java HrefMatch http://www.horstmann.com
```

Listing 1-9. `HrefMatch.java`

Code View:

```

1. import java.io.*;
2. import java.net.*;
3. import java.util.regex.*;
4.
5. /**
6.  * This program displays all URLs in a web page by matching a regular expression that describes
7.  * the <a href=...> HTML tag. Start the program as <br>
8.  * java HrefMatch URL
9.  * @version 1.01 2004-06-04
10. * @author Cay Horstmann
11. */
12. public class HrefMatch
13. {
14.     public static void main(String[] args)
```

```

15.     {
16.         try
17.         {
18.             // get URL string from command line or use default
19.             String urlString;
20.             if (args.length > 0) urlString = args[0];
21.             else urlString = "http://java.sun.com";
22.
23.             // open reader for URL
24.             InputStreamReader in = new InputStreamReader(new URL(urlString).openStream());
25.
26.             // read contents into string builder
27.             StringBuilder input = new StringBuilder();
28.             int ch;
29.             while ((ch = in.read()) != -1)
30.                 input.append((char) ch);
31.
32.             // search for all occurrences of pattern
33.             String patternString = "<a\\s+href\\s*=\\s*(\"[^"]*\"|[^\\s>])\\s*>";
34.             Pattern pattern = Pattern.compile(patternString, Pattern.CASE_INSENSITIVE);
35.             Matcher matcher = pattern.matcher(input);
36.
37.             while (matcher.find())
38.             {
39.                 int start = matcher.start();
40.                 int end = matcher.end();
41.                 String match = input.substring(start, end);
42.                 System.out.println(match);
43.             }
44.         }
45.         catch (IOException e)
46.         {
47.             e.printStackTrace();
48.         }
49.         catch (PatternSyntaxException e)
50.         {
51.             e.printStackTrace();
52.         }
53.     }
54. }
```

The `replaceAll` method of the `Matcher` class replaces all occurrences of a regular expression with a replacement string. For example, the following instructions replace all sequences of digits with a # character.

```

Pattern pattern = Pattern.compile("[0-9]+");
Matcher matcher = pattern.matcher(input);
String output = matcher.replaceAll("#");
```

The replacement string can contain references to groups in the pattern: `\$n` is replaced with the nth group. Use `\$` to include a \$ character in the replacement text.

The `replaceFirst` method replaces only the first occurrence of the pattern.

Finally, the `Pattern` class has a `split` method that splits an input into an array of strings, using the regular expression matches as boundaries. For example, the following instructions split the input into tokens, where the delimiters are punctuation marks surrounded by optional whitespace.

```

Pattern pattern = Pattern.compile("\\s*\\p{Punct}\\s*");
String[] tokens = pattern.split(input);
```



java.util.regex.Pattern 1.4

- static Pattern compile(String expression)
- static Pattern compile(String expression, int flags)

compiles the regular expression string into a pattern object for fast processing of matches.

Parameters: `expression` The regular expression
`flags` One or more of the flags `CASE_INSENSITIVE`,
`UNICODE_CASE`, `MULTILINE`, `UNIX_LINES`,
`DOTALL`, and `CANON_EQ`

- `Matcher matcher(CharSequence input)`

returns a `matcher` object that you can use to locate the matches of the pattern in the input.

- `String[] split(CharSequence input)`
- `String[] split(CharSequence input, int limit)`

splits the input string into tokens, where the pattern specifies the form of the delimiters. Returns an array of tokens. The delimiters are not part of the tokens.

Parameters: `input` The string to be split into tokens
`limit` The maximum number of strings to produce. If `limit - 1` matching delimiters have been found, then the last entry of the returned array contains the remaining unsplit input. If `limit` is ≤ 0 , then the entire input is split. If `limit` is 0, then trailing empty strings are not placed in the returned array

API

java.util.regex.Matcher 1.4

- `boolean matches()`

returns `true` if the input matches the pattern.

- `boolean lookingAt()`

returns `true` if the beginning of the input matches the pattern.

- `boolean find()`

- `boolean find(int start)`

attempts to find the next match and return `true` if another match is found.

Parameters: `start` The index at which to start searching

- `int start()`

- `int end()`

returns the start or past-the-end position of the current match.

- `String group()`

returns the current match.

- `int groupCount()`

returns the number of groups in the input pattern.

- `int start(int groupIndex)`

- `int end(int groupIndex)`

returns the start or past-the-end position of a given group in the current match.

Parameters: `groupIndex` The group index (starting with 1), or 0 to indicate the entire match

- `String group(int groupIndex)`

returns the string matching a given group.

Parameters: `groupIndex` The group index (starting with 1), or 0 to indicate the entire match

- `String replaceAll(String replacement)`

- `String replaceFirst(String replacement)`

returns a string obtained from the matcher input by replacing all matches, or the first match, with the replacement string.

Parameters: `replacement` The replacement string. It can contain references to a pattern group as `\$n`. Use `\$` to include a `$` symbol

- `Matcher reset()`

- `Matcher reset(CharSequence input)`

resets the matcher state. The second method makes the matcher work on a different input. Both methods return `this`.

You have now seen how to carry out input and output operations in Java, and you had an overview of the regular expression package that was a part of the "new I/O" specification. In the next chapter, we turn to the processing of XML data.





Chapter 2. XML

- [INTRODUCING XML](#)
- [PARSING AN XML DOCUMENT](#)
- [VALIDATING XML DOCUMENTS](#)
- [LOCATING INFORMATION WITH XPATH](#)
- [USING NAMESPACES](#)
- [STREAMING PARSERS](#)
- [GENERATING XML DOCUMENTS](#)
- [XSL TRANSFORMATIONS](#)

The preface of the book Essential XML by Don Box et al. (Addison-Wesley Professional 2000) states only half-jokingly: "The Extensible Markup Language (XML) has replaced Java, Design Patterns, and Object Technology as the software industry's solution to world hunger." Indeed, as you will see in this chapter, XML is a very useful technology for describing structured information. XML tools make it easy to process and transform that information. However, XML is not a silver bullet. You need domain-specific standards and code libraries to use it effectively. Moreover, far from making Java technology obsolete, XML works very well with Java. Since the late 1990s, IBM, Apache, and others have been instrumental in producing high-quality Java libraries for XML processing. Starting with Java SE 1.4, Sun has integrated the most important libraries into the Java platform.

This chapter introduces XML and covers the XML features of the Java library. As always, we point out along the way when the hype surrounding XML is justified and when you have to take it with a grain of salt and solve your problems the old-fashioned way, through good design and code.

Introducing XML

In Chapter 10 of Volume I, you have seen the use of property files to describe the configuration of a program. A property file contains a set of name/value pairs, such as

```
fontname=Times Roman  
fontsize=12  
windowsize=400 200  
color=0 50 100
```

You can use the `Properties` class to read in such a file with a single method call. That's a nice feature, but it doesn't really go far enough. In many cases, the information that you want to describe has more structure than the property file format can comfortably handle. Consider the `fontname/fontsize` entries in the example. It would be more object oriented to have a single entry:

```
font=Times Roman 12
```

But then parsing the font description gets ugly—you have to figure out when the font name ends and when the font size starts.

Property files have a single flat hierarchy. You can often see programmers work around that limitation with key names such as

```
title.fontname=Helvetica  
title.fontsize=36  
body.fontname=Times Roman  
body.fontsize=12
```

Another shortcoming of the property file format is caused by the requirement that keys be unique. To store a sequence of values, you need another workaround, such as

```
menu.item.1=Times Roman  
menu.item.2=Helvetica  
menu.item.3=Goudy Old Style
```

The XML format solves these problems because it can express hierarchical structures and thus is more flexible than the flat table structure of a property file.

An XML file for describing a program configuration might look like this:

Code View:

```
<configuration>
  <title>
    <font>
      <name>Helvetica</name>
      <size>36</size>
    </font>
  </title>
  <body>
    <font>
      <name>Times Roman</name>
      <size>12</size>
    </font>
  </body>
  <window>
    <width>400</width>
    <height>200</height>
  </window>
  <color>
    <red>0</red>
    <green>50</green>
    <blue>100</blue>
  </color>
  <menu>
    <item>Times Roman</item>
    <item>Helvetica</item>
    <item>Goudy Old Style</item>
  </menu>
</configuration>
```

The XML format allows you to express the structure hierarchy and repeated elements without contortions.

As you can see, the format of an XML file is straightforward. It looks similar to an HTML file. There is a good reason—both the XML and HTML formats are descendants of the venerable Standard Generalized Markup Language (SGML).

SGML has been around since the 1970s for describing the structure of complex documents. It has been used with success in some industries that require ongoing maintenance of massive documentation, in particular, the aircraft industry. However, SGML is quite complex, so it has never caught on in a big way. Much of that complexity arises because SGML has two conflicting goals. SGML wants to make sure that documents are formed according to the rules for their document type, but it also wants to make data entry easy by allowing shortcuts that reduce typing. XML was designed as a simplified version of SGML for use on the Internet. As is often true, simpler is better, and XML has enjoyed the immediate and enthusiastic reception that has eluded SGML for so long.

Note



You can find a very nice version of the XML standard, with annotations by Tim Bray, at <http://www.xml.com/axml/axml.html>.

Even though XML and HTML have common roots, there are important differences between the two.

- Unlike HTML, XML is case sensitive. For example, `<H1>` and `<h1>` are different XML tags.
- In HTML, you can omit end tags such as `</p>` or `` tags if it is clear from the context where a paragraph or list item ends. In XML, you can never omit an end tag.
- In XML, elements that have a single tag without a matching end tag must end in a `/`, as in ``. That way, the parser knows not to look for a `` tag.
- In XML, attribute values must be enclosed in quotation marks. In HTML, quotation marks are optional. For example, `<applet code="MyApplet.class" width=300 height=300>` is legal HTML but not legal XML. In XML, you have to use quotation marks: `width="300"`.

- In HTML, you can have attribute names without values, such as `<input type="radio" name="language" value="Java" checked>`. In XML, all attributes must have values, such as `checked="true"` or (ugh) `checked="checked"`.

Note



The current recommendation for web documents by the World Wide Web Consortium (W3C) is the XHTML standard, which tightens up the HTML standard to be XML compliant. You can find a copy of the XHTML standard at <http://www.w3.org/TR/xhtml1/>. XHTML is backward-compatible with current browsers, but not all HTML authoring tools support it. As XHTML becomes more widespread, you can use the XML tools that are described in this chapter to analyze web documents.

The Structure of an XML Document

An XML document should start with a header such as

```
<?xml version="1.0"?>
```

or

```
<?xml version="1.0" encoding="UTF-8"?>
```

Strictly speaking, a header is optional, but it is highly recommended.

Note



Because SGML was created for processing of real documents, XML files are called documents, even though many XML files describe data sets that one would not normally call documents.

The header can be followed by a document type definition (DTD), such as

```
<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

DTDs are an important mechanism to ensure the correctness of a document, but they are not required. We discuss them later in this chapter.

Finally, the body of the XML document contains the root element, which can contain other elements. For example,

```
<?xml version="1.0"?>
<!DOCTYPE configuration . . .>
<configuration>
  <title>
    <font>
      <name>Helvetica</name>
      <size>36</size>
    </font>
  </title>
  .
</configuration>
```

An element can contain child elements, text, or both. In the preceding example, the `font` element has two child elements, `name` and `size`. The `name` element contains the text "Helvetica".

Tip



It is best if you structure your XML documents such that an element contains either child elements or text. In other words, you should avoid situations such as

```
<font>
  Helvetica
  <size>36</size>
</font>
```

This is called mixed contents in the XML specification. As you will see later in this chapter, you can simplify parsing if you avoid mixed contents.

XML elements can contain attributes, such as

```
<size unit="pt">36</size>
```

There is some disagreement among XML designers about when to use elements and when to use attributes. For example, it would seem easier to describe a font as

```
<font name="Helvetica" size="36"/>
```

than

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

However, attributes are much less flexible. Suppose you want to add units to the size value. If you use attributes, then you must add the unit to the attribute value:

```
<font name="Helvetica" size="36 pt"/>
```

Ugh! Now you have to parse the string "36 pt", just the kind of hassle that XML was designed to avoid. Adding an attribute to the `size` element is much cleaner:

```
<font>
  <name>Helvetica</name>
  <size unit="pt">36</size>
</font>
```

A commonly used rule of thumb is that attributes should be used only to modify the interpretation of a value, not to specify values. If you find yourself engaged in metaphysical discussions about whether a particular setting is a modification of the interpretation of a value or not, then just say "no" to attributes and use elements throughout. Many useful XML documents don't use attributes at all.

Note



In HTML, the rule for attribute usage is simple: If it isn't displayed on the web page, it's an attribute. For example, consider the hyperlink

```
<a href="http://java.sun.com">Java Technology</a>
```

The string Java Technology is displayed on the web page, but the URL of the link is not a part of the displayed page. However, the rule isn't all that helpful for most XML files because the data in an XML file aren't normally meant to be viewed by humans.

Elements and text are the "bread and butter" of XML documents. Here are a few other markup instructions that you might encounter:

- Character references have the form `&#decimalValue;` or `&#xhexValue;`. For example, the character é can be denoted with either of the following:

```
&#233;
&#xD9;
```

- Entity references have the form `&name;`. The entity references

```
&lt;
&gt;
```

&
"
'

have predefined meanings: the less than, greater than, ampersand, quotation mark, and apostrophe characters. You can define other entity references in a DTD.

- CDATA sections are delimited by `<! [CDATA[and]]>`. They are a special form of character data. You can use them to include strings that contain characters such as `< > &` without having them interpreted as markup, for example,

```
<! [CDATA[< &gt; are my favorite delimiters]]>
```

CDATA sections cannot contain the string `]]>`. Use this feature with caution! It is too often used as a back door for smuggling legacy data into XML documents.

- Processing instructions are instructions for applications that process XML documents. They are delimited by `<? and ?>`, for example,

```
<?xml-stylesheet href="mystyle.css" type="text/css"?>
```

Every XML document starts with a processing instruction

```
<?xml version="1.0"?>
```

- Comments are delimited by `<!-- and -->`, for example,

```
<!-- This is a comment. -->
```

Comments should not contain the string `--`. Comments should only be information for human readers. They should never contain hidden commands. Use processing instructions for commands.



Chapter 2. XML

- [INTRODUCING XML](#)
- [PARSING AN XML DOCUMENT](#)
- [VALIDATING XML DOCUMENTS](#)
- [LOCATING INFORMATION WITH XPATH](#)
- [USING NAMESPACES](#)
- [STREAMING PARSERS](#)
- [GENERATING XML DOCUMENTS](#)
- [XSL TRANSFORMATIONS](#)

The preface of the book Essential XML by Don Box et al. (Addison-Wesley Professional 2000) states only half-jokingly: "The Extensible Markup Language (XML) has replaced Java, Design Patterns, and Object Technology as the software industry's solution to world hunger." Indeed, as you will see in this chapter, XML is a very useful technology for describing structured information. XML tools make it easy to process and transform that information. However, XML is not a silver bullet. You need domain-specific standards and code libraries to use it effectively. Moreover, far from making Java technology obsolete, XML works very well with Java. Since the late 1990s, IBM, Apache, and others have been instrumental in producing high-quality Java libraries for XML processing. Starting with Java SE 1.4, Sun has integrated the most important libraries into the Java platform.

This chapter introduces XML and covers the XML features of the Java library. As always, we point out along the way when the hype surrounding XML is justified and when you have to take it with a grain of salt and solve your problems the old-fashioned way, through good design and code.

Introducing XML

In Chapter 10 of Volume I, you have seen the use of property files to describe the configuration of a program. A property file contains a set of name/value pairs, such as

```
fontname=Times Roman
fontsize=12
windowsize=400 200
color=0 50 100
```

You can use the `Properties` class to read in such a file with a single method call. That's a nice feature, but it doesn't really go far enough. In many cases, the information that you want to describe has more structure than the property file format can comfortably handle. Consider the `fontname/fontsize` entries in the example. It would be more object oriented to have a single entry:

```
font=Times Roman 12
```

But then parsing the font description gets ugly—you have to figure out when the font name ends and when the font size starts.

Property files have a single flat hierarchy. You can often see programmers work around that limitation with key names such as

```
title.fontname=Helvetica
title.fontsize=36
body.fontname=Times Roman
body.fontsize=12
```

Another shortcoming of the property file format is caused by the requirement that keys be unique. To store a sequence of values, you need another workaround, such as

```
menu.item.1=Times Roman
menu.item.2=Helvetica
menu.item.3=Goudy Old Style
```

The XML format solves these problems because it can express hierarchical structures and thus is more flexible than the flat table structure of a property file.

An XML file for describing a program configuration might look like this:

Code View:

```
<configuration>
  <title>
    <font>
      <name>Helvetica</name>
      <size>36</size>
    </font>
  </title>
  <body>
    <font>
      <name>Times Roman</name>
      <size>12</size>
    </font>
  </body>
  <window>
    <width>400</width>
    <height>200</height>
  </window>
  <color>
    <red>0</red>
    <green>50</green>
    <blue>100</blue>
  </color>
  <menu>
    <item>Times Roman</item>
    <item>Helvetica</item>
    <item>Goudy Old Style</item>
  </menu>
</configuration>
```

The XML format allows you to express the structure hierarchy and repeated elements without contortions.

As you can see, the format of an XML file is straightforward. It looks similar to an HTML file. There is a good reason—both the XML and HTML formats are descendants of the venerable Standard Generalized Markup Language (SGML).

SGML has been around since the 1970s for describing the structure of complex documents. It has been used with success in some industries that require ongoing maintenance of massive documentation, in particular, the aircraft industry. However, SGML is quite complex, so it has never caught on in a big way. Much of that complexity arises because SGML has two conflicting goals. SGML wants to make sure that documents are formed according to the rules for their document type, but it also wants to make data entry easy by allowing shortcuts that reduce typing. XML was designed as a simplified version of SGML for use on the Internet. As is often true, simpler is better, and XML has enjoyed the immediate and enthusiastic reception that has eluded SGML for so long.

Note



You can find a very nice version of the XML standard, with annotations by Tim Bray, at <http://www.xml.com/axml/axml.html>.

Even though XML and HTML have common roots, there are important differences between the two.

- Unlike HTML, XML is case sensitive. For example, `<H1>` and `<h1>` are different XML tags.
- In HTML, you can omit end tags such as `</p>` or `` tags if it is clear from the context where a paragraph or list item ends. In XML, you can never omit an end tag.
- In XML, elements that have a single tag without a matching end tag must end in a `/`, as in ``. That way, the parser knows not to look for a `` tag.
- In XML, attribute values must be enclosed in quotation marks. In HTML, quotation marks are optional. For example, `<applet code="MyApplet.class" width=300 height=300>` is legal HTML but not legal XML. In XML, you have to use quotation marks: `width="300"`.
- In HTML, you can have attribute names without values, such as `<input type="radio" name="language" value="Java" checked>`. In XML, all attributes must have values, such as `checked="true"` or (ugh) `checked="checked"`.

Note



The current recommendation for web documents by the World Wide Web Consortium (W3C) is the XHTML standard, which tightens up the HTML standard to be XML compliant. You can find a copy of the XHTML standard at <http://www.w3.org/TR/xhtml1/>. XHTML is backward-compatible with current browsers, but not all HTML authoring tools support it. As XHTML becomes more widespread, you can use the XML tools that are described in this chapter to analyze web documents.

The Structure of an XML Document

An XML document should start with a header such as

```
<?xml version="1.0"?>
```

or

```
<?xml version="1.0" encoding="UTF-8"?>
```

Strictly speaking, a header is optional, but it is highly recommended.

Note



Because SGML was created for processing of real documents, XML files are called documents, even though many XML files describe data sets that one would not normally call documents.

The header can be followed by a document type definition (DTD), such as

```
<!DOCTYPE web-app PUBLIC  
        "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
```

```
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

DTDs are an important mechanism to ensure the correctness of a document, but they are not required. We discuss them later in this chapter.

Finally, the body of the XML document contains the root element, which can contain other elements. For example,

```
<?xml version="1.0"?>
<!DOCTYPE configuration . . .>
<configuration>
  <title>
    <font>
      <name>Helvetica</name>
      <size>36</size>
    </font>
  </title>
  . .
</configuration>
```

An element can contain child elements, text, or both. In the preceding example, the `font` element has two child elements, `name` and `size`. The `name` element contains the text "Helvetica".

Tip

 It is best if you structure your XML documents such that an element contains either child elements or text. In other words, you should avoid situations such as

```
<font>
  Helvetica
  <size>36</size>
</font>
```

This is called mixed contents in the XML specification. As you will see later in this chapter, you can simplify parsing if you avoid mixed contents.

XML elements can contain attributes, such as

```
<size unit="pt">36</size>
```

There is some disagreement among XML designers about when to use elements and when to use attributes. For example, it would seem easier to describe a font as

```
<font name="Helvetica" size="36"/>
```

than

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

However, attributes are much less flexible. Suppose you want to add units to the size value. If you use attributes, then you must add the unit to the attribute value:

```
<font name="Helvetica" size="36 pt"/>
```

Ugh! Now you have to parse the string "36 pt", just the kind of hassle that XML was designed to avoid. Adding an attribute to the `size` element is much cleaner:

```
<font>
  <name>Helvetica</name>
  <size unit="pt">36</size>
</font>
```

A commonly used rule of thumb is that attributes should be used only to modify the interpretation of a value, not to specify values. If you find yourself engaged in metaphysical discussions about whether a particular setting is a modification of the interpretation of a value or not, then just say "no" to attributes and use elements throughout. Many useful XML documents don't use attributes at all.

Note



In HTML, the rule for attribute usage is simple: If it isn't displayed on the web page, it's an attribute. For example, consider the hyperlink

```
<a href="http://java.sun.com">Java Technology</a>
```

The string Java Technology is displayed on the web page, but the URL of the link is not a part of the displayed page. However, the rule isn't all that helpful for most XML files because the data in an XML file aren't normally meant to be viewed by humans.

Elements and text are the "bread and butter" of XML documents. Here are a few other markup instructions that you might encounter:

- Character references have the form &#decimalValue; or &#xhexValue;. For example, the character é can be denoted with either of the following:

```
&#233;  
&#xD9;
```

- Entity references have the form &name;. The entity references

```
&lt;  
&gt;  
&amp;  
&quot;  
&apos;
```

have predefined meanings: the less than, greater than, ampersand, quotation mark, and apostrophe characters. You can define other entity references in a DTD.

- CDATA sections are delimited by <! [CDATA[and]]>. They are a special form of character data. You can use them to include strings that contain characters such as < > & without having them interpreted as markup, for example,

```
<! [CDATA[< &gt; are my favorite delimiters]]>
```

CDATA sections cannot contain the string]]>. Use this feature with caution! It is too often used as a back door for smuggling legacy data into XML documents.

- Processing instructions are instructions for applications that process XML documents. They are delimited by <? and ?>, for example,

```
<?xmlstylesheet href="mystyle.css" type="text/css"?>
```

Every XML document starts with a processing instruction

```
<?xml version="1.0"?>
```

- Comments are delimited by <!-- and -->, for example,

```
<!-- This is a comment. -->
```

Comments should not contain the string --. Comments should only be information for human readers. They should never contain hidden commands. Use processing instructions for commands.



Parsing an XML Document

To process an XML document, you need to parse it. A parser is a program that reads a file, confirms that the file has the correct

format, breaks it up into the constituent elements, and lets a programmer access those elements. The Java library supplies two kinds of XML parsers:

- Tree parsers such as the Document Object Model (DOM) parser that read an XML document into a tree structure.
- Streaming parsers such as the Simple API for XML (SAX) parser that generate events as they read an XML document.

The DOM parser is easy to use for most purposes, and we explain it first. You would consider a streaming parser if you process very long documents whose tree structures would use up a lot of memory, or if you are just interested in a few elements and you don't care about their context. For more information, see the section "[Streaming Parsers](#)" on page [138](#).

The DOM parser interface is standardized by the World Wide Web Consortium (W3C). The `org.w3c.dom` package contains the definitions of interface types such as `Document` and `Element`. Different suppliers, such as the Apache Organization and IBM, have written DOM parsers whose classes implement these interfaces. The Sun Java API for XML Processing (JAXP) library actually makes it possible to plug in any of these parsers. But Sun also includes its own DOM parser in the Java SDK. We use the Sun parser in this chapter.

To read an XML document, you need a `DocumentBuilder` object, which you get from a `DocumentBuilderFactory`, like this:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
```

You can now read a document from a file:

```
File f = . . .
Document doc = builder.parse(f);
```

Alternatively, you can use a URL:

```
URL u = . . .
Document doc = builder.parse(u);
```

You can even specify an arbitrary input stream:

```
InputStream in = . . .
Document doc = builder.parse(in);
```

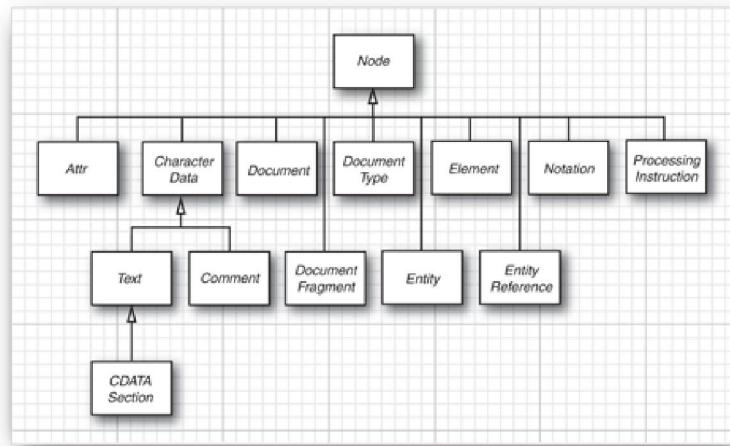
Note

-  If you use an input stream as an input source, then the parser will not be able to locate other files that are referenced relative to the location of the document, such as a DTD in the same directory. You can install an "entity resolver" to overcome that problem.

The `Document` object is an in-memory representation of the tree structure of the XML document. It is composed of objects whose classes implement the `Node` interface and its various subinterfaces. [Figure 2-1](#) shows the inheritance hierarchy of the subinterfaces.

Figure 2-1. The Node interface and its subinterfaces

[\[View full size image\]](#)



You start analyzing the contents of a document by calling the `getDocumentElement` method. It returns the root element.

```
Element root = doc.getDocumentElement();
```

For example, if you are processing a document

```
<?xml version="1.0"?>
<font>
  .
  .
</font>
```

then calling `getDocumentElement` returns the `font` element.

The `getTagName` method returns the tag name of an element. In the preceding example, `root.getTagName()` returns the string "font".

To get the element's children (which may be subelements, text, comments, or other nodes), use the `getChildNodes` method. That method returns a collection of type `NodeList`. That type was invented before the standard Java collections, and it has a different access protocol. The `item` method gets the item with a given index, and the `getLength` method gives the total count of the items. Therefore, you can enumerate all children like this:

```
NodeList children = root.getChildNodes();
for (int i = 0; i < children.getLength(); i++)
{
    Node child = children.item(i);
    .
}
```

Be careful when analyzing the children. Suppose, for example, that you are processing the document

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

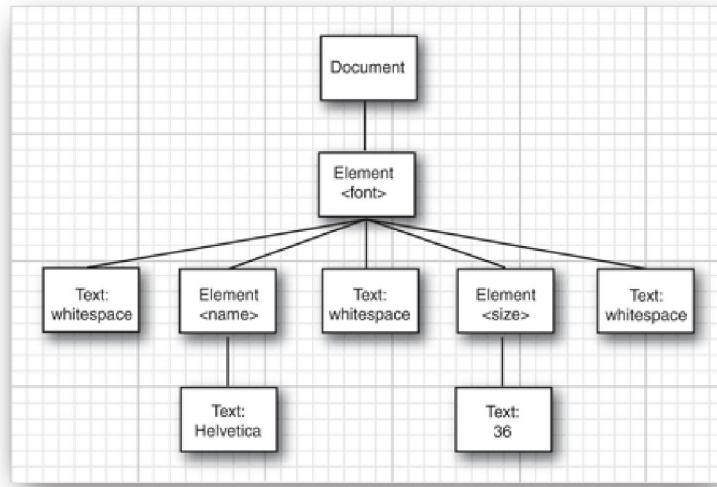
You would expect the `font` element to have two children, but the parser reports five:

- The whitespace between `` and `<name>`
- The `name` element
- The whitespace between `</name>` and `<size>`
- The `size` element
- The whitespace between `</size>` and ``

[Figure 2-2](#) shows the DOM tree.

Figure 2-2. A simple DOM tree

[\[View full size image\]](#)



If you expect only subelements, then you can ignore the whitespace:

```

for (int i = 0; i < children.getLength(); i++)
{
    Node child = children.item(i);
    if (child instanceof Element)
    {
        Element childElement = (Element) child;
        . . .
    }
}
  
```

Now you look at only two elements, with tag names `name` and `size`.

As you see in the next section, you can do even better if your document has a DTD. Then the parser knows which elements don't have text nodes as children, and it can suppress the whitespace for you.

When analyzing the `name` and `size` elements, you want to retrieve the text strings that they contain. Those text strings are themselves contained in child nodes of type `Text`. Because you know that these `Text` nodes are the only children, you can use the `getFirstChild` method without having to traverse another `NodeList`. Then use the `getData` method to retrieve the string stored in the `Text` node.

```

for (int i = 0; i < children.getLength(); i++)
{
    Node child = children.item(i);
    if (child instanceof Element)
    {
        Element childElement = (Element) child;
        Text textNode = (Text) childElement.getFirstChild();
        String text = textNode.getData().trim();
        if (childElement.getTagName().equals("name"))
            name = text;
        else if (childElement.getTagName().equals("size"))
            size = Integer.parseInt(text);
    }
}
  
```

Tip



It is a good idea to call `trim` on the return value of the `getData` method. If the author of an XML file puts the beginning and the ending tag on separate lines, such as

```
<size>
  36
```

```
</size>
```

then the parser includes all line breaks and spaces in the text node data. Calling the `trim` method removes the whitespace surrounding the actual data.

You can also get the last child with the `getLastChild` method, and the next sibling of a node with `getNextSibling`. Therefore, another way of traversing a set of child nodes is

```
for (Node childNode = element.getFirstChild();
    childNode != null;
    childNode = childNode.getNextSibling())
{
    ...
}
```

To enumerate the attributes of a node, call the `getAttributes` method. It returns a `NamedNodeMap` object that contains `Node` objects describing the attributes. You can traverse the nodes in a `NamedNodeMap` in the same way as a `NodeList`. Then call the `getNodeName` and `getNodeValue` methods to get the attribute names and values.

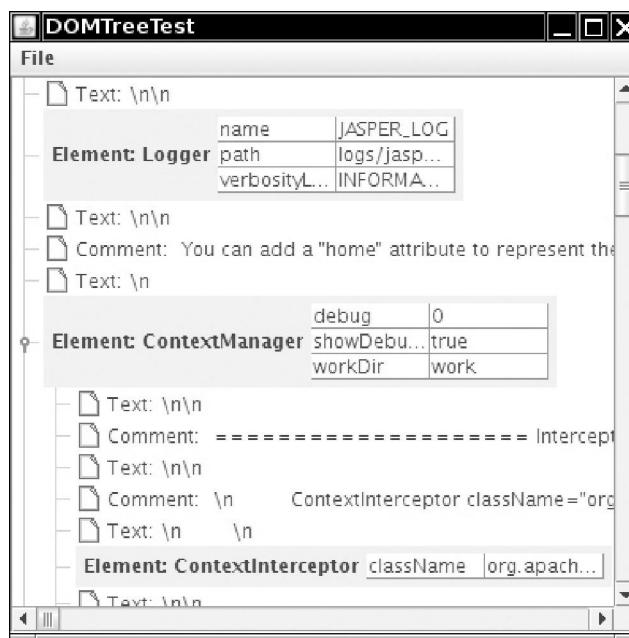
```
NamedNodeMap attributes = element.getAttributes();
for (int i = 0; i < attributes.getLength(); i++)
{
    Node attribute = attributes.item(i);
    String name = attribute.getNodeName();
    String value = attribute.getNodeValue();
    ...
}
```

Alternatively, if you know the name of an attribute, you can retrieve the corresponding value directly:

```
String unit = element.getAttribute("unit");
```

You have now seen how to analyze a DOM tree. The program in [Listing 2-1](#) puts these techniques to work. You can use the File -> Open menu option to read in an XML file. A `DocumentBuilder` object parses the XML file and produces a `Document` object. The program displays the `Document` object as a tree (see [Figure 2-3](#)).

Figure 2-3. A parse tree of an XML document



The tree display shows clearly how child elements are surrounded by text containing whitespace and comments. For greater clarity, the program displays newline and return characters as `\n` and `\r`. (Otherwise, they would show up as hollow boxes, the default symbol for a character that Swing cannot draw in a string.)

In [Chapter 6](#), you will learn the techniques that this program uses to display the tree and the attribute tables. The `DOMTreeModel` class implements the `TreeModel` interface. The `getRoot` method returns the root element of the document. The `getChild` method gets the node list of children and returns the item with the requested index. The tree cell renderer displays the following:

- For elements, the element tag name and a table of all attributes.
- For character data, the interface (Text, Comment, or CDATASection), followed by the data, with newline and return characters replaced by `\n` and `\r`.
- For all other node types, the class name followed by the result of `toString`.

Listing 2-1. `DOMTreeTest.java`

Code View:

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.io.*;
4. import javax.swing.*;
5. import javax.swing.event.*;
6. import javax.swing.table.*;
7. import javax.swing.tree.*;
8. import javax.xml.parsers.*;
9. import org.w3c.dom.*;
10.
11. /**
12.  * This program displays an XML document as a tree.
13.  * @version 1.11 2007-06-24
14.  * @author Cay Horstmann
15. */
16. public class DOMTreeTest
17. {
18.     public static void main(String[] args)
19.     {
20.         EventQueue.invokeLater(new Runnable()
21.         {
22.             public void run()
23.             {
24.                 JFrame frame = new DOMTreeFrame();
25.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26.                 frame.setVisible(true);
27.             }
28.         });
29.     }
30. }
31.
32. /**
33.  * This frame contains a tree that displays the contents of an XML document.
34. */
35. class DOMTreeFrame extends JFrame
36. {
37.     public DOMTreeFrame()
38.     {
39.         setTitle("DOMTreeTest");
40.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
41.
42.         JMenu fileMenu = new JMenu("File");
43.         JMenuItem openItem = new JMenuItem("Open");
44.         openItem.addActionListener(new ActionListener()
45.         {
46.             public void actionPerformed(ActionEvent event)
47.             {
48.                 openFile();
49.             }
50.         });
51.         fileMenu.add(openItem);
52.
53.         JMenuItem exitItem = new JMenuItem("Exit");
54.         exitItem.addActionListener(new ActionListener()
55.         {
56.             public void actionPerformed(ActionEvent event)
57.             {
58.                 System.exit(0);
59.             }
60.         });
61.         fileMenu.add(exitItem);
```

```
62.     JMenuBar menuBar = new JMenuBar();
63.     menuBar.add(fileMenu);
64.     setJMenuBar(menuBar);
65. }
66.
67.
68. /**
69. * Open a file and load the document.
70. */
71. public void openFile()
72. {
73.     JFileChooser chooser = new JFileChooser();
74.     chooser.setCurrentDirectory(new File("."));
75.
76.     chooser.setFileFilter(new javax.swing.filechooser.FileFilter()
77.     {
78.         public boolean accept(File f)
79.         {
80.             return f.isDirectory() || f.getName().toLowerCase().endsWith(".xml");
81.         }
82.
83.         public String getDescription()
84.         {
85.             return "XML files";
86.         }
87.     });
88.     int r = chooser.showOpenDialog(this);
89.     if (r != JFileChooser.APPROVE_OPTION) return;
90.     final File file = chooser.getSelectedFile();
91.
92.     new SwingWorker<Document, Void>()
93.     {
94.         protected Document doInBackground() throws Exception
95.         {
96.             if (builder == null)
97.             {
98.                 DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
99.                 builder = factory.newDocumentBuilder();
100.            }
101.            return builder.parse(file);
102.        }
103.
104.        protected void done()
105.        {
106.            try
107.            {
108.                Document doc = get();
109.                JTree tree = new JTree(new DOMTreeModel(doc));
110.                tree.setCellRenderer(new DOMTreeCellRenderer());
111.
112.                setContentPane(new JScrollPane(tree));
113.                validate();
114.            }
115.            catch (Exception e)
116.            {
117.                JOptionPane.showMessageDialog(DOMTreeFrame.this, e);
118.            }
119.        }
120.    }.execute();
121. }
122.
123. private DocumentBuilder builder;
124. private static final int DEFAULT_WIDTH = 400;
125. private static final int DEFAULT_HEIGHT = 400;
126. }
127.
128. /**
129. * This tree model describes the tree structure of an XML document.
130. */
131. class DOMTreeModel implements TreeModel
132. {
133.     /**
134.      * Constructs a document tree model.
135.      * @param doc the document
136.      */
137.     public DOMTreeModel(Document doc)
138.     {
139.         this.doc = doc;
140.     }
141.
142.     public Object getRoot()
```

```
143.     {
144.         return doc.getDocumentElement();
145.     }
146.
147.     public int getChildCount(Object parent)
148.     {
149.         Node node = (Node) parent;
150.         NodeList list = node.getChildNodes();
151.         return list.getLength();
152.     }
153.
154.     public Object getChild(Object parent, int index)
155.     {
156.         Node node = (Node) parent;
157.         NodeList list = node.getChildNodes();
158.         return list.item(index);
159.     }
160.
161.     public int getIndexOfChild(Object parent, Object child)
162.     {
163.         Node node = (Node) parent;
164.         NodeList list = node.getChildNodes();
165.         for (int i = 0; i < list.getLength(); i++)
166.             if (getChild(node, i) == child) return i;
167.         return -1;
168.     }
169.
170.     public boolean isLeaf(Object node)
171.     {
172.         return getChildCount(node) == 0;
173.     }
174.
175.     public void valueForPathChanged(TreePath path, Object newValue)
176.     {
177.     }
178.
179.     public void addTreeModelListener(TreeModelListener l)
180.     {
181.     }
182.
183.     public void removeTreeModelListener(TreeModelListener l)
184.     {
185.     }
186.
187.     private Document doc;
188. }
189.
190. /**
191. * This class renders an XML node.
192. */
193. class DOMTreeCellRenderer extends DefaultTreeCellRenderer
194. {
195.     public Component getTreeCellRendererComponent(JTree tree, Object value, boolean selected,
196.                                                 boolean expanded, boolean leaf, int row, boolean hasFocus)
197.     {
198.         Node node = (Node) value;
199.         if (node instanceof Element) return elementPanel((Element) node);
200.
201.         super.getTreeCellRendererComponent(tree, value, selected, expanded, leaf, row, hasFocus);
202.         if (node instanceof CharacterData) setText(characterString((CharacterData) node));
203.         else setText(node.getClass() + ":" + node.toString());
204.         return this;
205.     }
206.
207.     public static JPanel elementPanel(Element e)
208.     {
209.         JPanel panel = new JPanel();
210.         panel.add(new JLabel("Element: " + e.getTagName()));
211.         final NamedNodeMap map = e.getAttributes();
212.         panel.add(new JTable(new AbstractTableModel()
213.         {
214.             public int getRowCount()
215.             {
216.                 return map.getLength();
217.             }
218.
219.             public int getColumnCount()
220.             {
221.                 return 2;
222.             }
223.         }
```

```
224.         public Object getValueAt(int r, int c)
225.         {
226.             return c == 0 ? map.item(r).getNodeName() : map.item(r).getNodeValue();
227.         }
228.     });
229.     return panel;
230. }
231.
232. public static String characterString(CharacterData node)
233. {
234.     StringBuilder builder = new StringBuilder(node.getData());
235.     for (int i = 0; i < builder.length(); i++)
236.     {
237.         if (builder.charAt(i) == '\r')
238.         {
239.             builder.replace(i, i + 1, "\\\r");
240.             i++;
241.         }
242.         else if (builder.charAt(i) == '\n')
243.         {
244.             builder.replace(i, i + 1, "\\\n");
245.             i++;
246.         }
247.         else if (builder.charAt(i) == '\t')
248.         {
249.             builder.replace(i, i + 1, "\\\t");
250.             i++;
251.         }
252.     }
253.     if (node instanceof CDATASection) builder.insert(0, "CDATASection: ");
254.     else if (node instanceof Text) builder.insert(0, "Text: ");
255.     else if (node instanceof Comment) builder.insert(0, "Comment: ");
256.
257.     return builder.toString();
258. }
259. }
```



javax.xml.parsers.DocumentBuilderFactory 1.4

- static DocumentBuilderFactory newInstance()

returns an instance of the DocumentBuilderFactory class.

- DocumentBuilder newDocumentBuilder()

returns an instance of the DocumentBuilder class.



javax.xml.parsers.DocumentBuilder 1.4

- Document parse(File f)
- Document parse(String url)
- Document parse(InputStream in)

parses an XML document from the given file, URL, or input stream and returns the parsed document.



org.w3c.dom.Document 1.4

- `Element getDocumentElement()`
returns the root element of the document.



org.w3c.dom.Element 1.4

- `String getTagName()`
returns the name of the element.
- `String getAttribute(String name)`
returns the value of the attribute with the given name, or the empty string if there is no such attribute.



org.w3c.dom.Node 1.4

- `NodeList getChildNodes()`
returns a node list that contains all children of this node.
- `Node getFirstChild()`
- `Node getLastChild()`
gets the first or last child node of this node, or `null` if this node has no children.
- `Node getNextSibling()`
- `Node getPreviousSibling()`
gets the next or previous sibling of this node, or `null` if this node has no siblings.
- `Node getParentNode()`
gets the parent of this node, or `null` if this node is the document node.
- `NamedNodeMap getAttributes()`
returns a node map that contains `Attr` nodes that describe all attributes of this node.
- `String getNodeName()`
returns the name of this node. If the node is an `Attr` node, then the name is the attribute name.
- `String getNodeValue()`
returns the value of this node. If the node is an `Attr` node, then the value is the attribute value.



org.w3c.dom.CharacterData 1.4

- `String getData()`
returns the text stored in this node.

API**org.w3c.dom.NodeList 1.4**

- int getLength()
returns the number of nodes in this list.
- Node item(int index)
returns the node with the given index. The index is between 0 and getLength() - 1.

API**org.w3c.dom.NamedNodeMap 1.4**

- int getLength()
returns the number of nodes in this map.
- Node item(int index)
returns the node with the given index. The index is between 0 and getLength() - 1.



Validating XML Documents

In the preceding section, you saw how to traverse the tree structure of a DOM document. However, if you simply follow that approach, you'll find that you will have quite a bit of tedious programming and error checking. Not only do you have to deal with whitespace between elements, but you also need to check whether the document contains the nodes that you expect. For example, suppose you are reading an element:

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

You get the first child. Oops . . . it is a text node containing whitespace "\n ". You skip text nodes and find the first element node. Then you need to check that its tag name is "name". You need to check that it has one child node of type `Text`. You move on to the next nonwhitespace child and make the same check. What if the author of the document switched the order of the children or added another child element? It is tedious to code all the error checking, and reckless to skip the checks.

Fortunately, one of the major benefits of an XML parser is that it can automatically verify that a document has the correct structure. Then the parsing becomes much simpler. For example, if you know that the `font` fragment has passed validation, then you can simply get the two grandchildren, cast them as `Text` nodes, and get the text data, without any further checking.

To specify the document structure, you can supply a DTD or an XML Schema definition. A DTD or schema contains rules that explain how a document should be formed, by specifying the legal child elements and attributes for each element. For example, a DTD might contain a rule:

```
<!ELEMENT font (name,size)>
```

This rule expresses that a `font` element must always have two children, which are `name` and `size` elements. The XML Schema language expresses the same constraint as

```
<xsd:element name="font">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="size" type="xsd:int"/>
  </xsd:sequence>
</xsd:element>
```

XML Schema can express more sophisticated validation conditions (such as the fact that the `size` element must contain an integer) than can DTDs. Unlike the DTD syntax, the XML Schema syntax uses XML, which is a benefit if you need to process schema files.

The XML Schema language was designed to replace DTDs. However, as we write this chapter, DTDs are still very much alive. XML Schema is very complex and far from universally adopted. In fact, some XML users are so annoyed by the complexity of XML Schema that they use alternative validation languages. The most common choice is Relax NG (<http://www.relaxng.org>).

In the next section, we discuss DTDs in detail. We then briefly cover the basics of XML Schema support. Finally, we show you a complete application that demonstrates how validation simplifies XML programming.

Document Type Definitions

There are several methods for supplying a DTD. You can include a DTD in an XML document like this:

```
<?xml version="1.0"?>
<!DOCTYPE configuration [
    <!ELEMENT configuration . . .>
    more rules
    . .
]>
<configuration>
    . .
</configuration>
```

As you can see, the rules are included inside a `DOCTYPE` declaration, in a block delimited by `[. . .]`. The document type must match the name of the root element, such as `configuration` in our example.

Supplying a DTD inside an XML document is somewhat uncommon because DTDs can grow lengthy. It makes more sense to store the DTD externally. The `SYSTEM` declaration can be used for that purpose. You specify a URL that contains the DTD, for example:

```
<!DOCTYPE configuration SYSTEM "config.dtd">
```

or

```
<!DOCTYPE configuration SYSTEM "http://myserver.com/config.dtd">
```

Caution



If you use a relative URL for the DTD (such as `"config.dtd"`), then give the parser a `File` or `URL` object, not an `InputStream`. If you must parse from an input stream, supply an entity resolver—see the following note.

Finally, the mechanism for identifying "well known" DTDs has its origin in SGML. Here is an example:

```
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
```

If an XML processor knows how to locate the DTD with the public identifier, then it need not go to the URL.

Note



If you use a DOM parser and would like to support a `PUBLIC` identifier, call the `setEntityResolver` method of the `DocumentBuilder` class to install an object of a class that implements the `EntityResolver` interface. That interface has a single method, `resolveEntity`. Here is the outline of a typical implementation:

```
class MyEntityResolver implements EntityResolver
{
    public InputSource resolveEntity(String publicID,
                                    String systemID)
    {
        if (publicID.equals(a known ID))
```

```

        return new InputSource(DTD data);
    else
        return null; // use default behavior
    }
}

```

You can construct the input source from an `InputStream`, a `Reader`, or a string.

Now that you have seen how the parser locates the DTD, let us consider the various kinds of rules.

The `ELEMENT` rule specifies what children an element can have. You specify a regular expression, made up of the components shown in [Table 2-1](#).

Table 2-1. Rules for Element Content

Rule	Meaning
<code>E*</code>	0 or more occurrences of E
<code>E+</code>	1 or more occurrences of E
<code>E?</code>	0 or 1 occurrences of E
<code>E₁ E₂ ... E_n</code>	One of E ₁ , E ₂ , ..., E _n
<code>E₁, E₂, ..., E_n</code>	E ₁ followed by E ₂ , ..., E _n
<code>#PCDATA</code>	Text
<code>(#PCDATA E₁ E₂ ... E_n) *</code>	0 or more occurrences of text and E ₁ , E ₂ , ..., E _n in any order (mixed content)
<code>ANY</code>	Any children allowed
<code>EMPTY</code>	No children allowed

Here are several simple but typical examples. The following rule states that a `menu` element contains 0 or more `item` elements:

```
<!ELEMENT menu (item)*>
```

This set of rules states that a font is described by a name followed by a size, each of which contain text:

```
<!ELEMENT font (name,size)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT size (#PCDATA)>
```

The abbreviation `PCDATA` denotes parsed character data. The data are called "parsed" because the parser interprets the text string, looking for < characters that denote the start of a new tag, or & characters that denote the start of an entity.

An element specification can contain regular expressions that are nested and complex. For example, here is a rule that describes the makeup of a chapter in this book:

```
<!ELEMENT chapter (intro,(heading,(para|image|table|note)+)+)
```

Each chapter starts with an introduction, which is followed by one or more sections consisting of a heading and one or more paragraphs, images, tables, or notes.

However, in one common case you can't define the rules to be as flexible as you might like. Whenever an element can contain text, then there are only two valid cases. Either the element contains nothing but text, such as

```
<!ELEMENT name (#PCDATA)>
```

or the element contains any combination of text and tags in any order, such as

```
<!ELEMENT para (#PCDATA|em|strong|code)*>
```

It is not legal to specify other types of rules that contain #PCDATA. For example, the following rule is illegal:

```
<!ELEMENT captionedImage (image, #PCDATA) >
```

You have to rewrite such a rule, either by introducing another `caption` element or by allowing any combination of `image` tags and text.

This restriction simplifies the job of the XML parser when parsing mixed content (a mixture of tags and text). Because you lose some control when allowing mixed content, it is best to design DTDs such that all elements contain either other elements or nothing but text.

Note



Actually, it isn't quite true that you can specify arbitrary regular expressions of elements in a DTD rule. An XML parser may reject certain complex rule sets that lead to "nondeterministic" parsing. For example, a regular expression $((x, y) \mid (x, z))$ is nondeterministic. When the parser sees x , it doesn't know which of the two alternatives to take. This expression can be rewritten in a deterministic form, as $(x, (y \mid z))$. However, some expressions can't be reformulated, such as $((x, y)^* \mid x?)$. The Sun parser gives no warnings when presented with an ambiguous DTD. It simply picks the first matching alternative when parsing, which causes it to reject some correct inputs. Of course, the parser is well within its rights to do so because the XML standard allows a parser to assume that the DTD is unambiguous.

In practice, this isn't an issue over which you should lose sleep, because most DTDs are so simple that you never run into ambiguity problems.

You also specify rules to describe the legal attributes of elements. The general syntax is

```
<!ATTLIST element attribute type default>
```

[Table 2-2](#) shows the legal attribute types, and [Table 2-3](#) shows the syntax for the defaults.

Table 2-2. Attribute Types

Type	Meaning
CDATA	Any character string
$(A_1 \mid A_2 \mid \dots \mid A_n)$	One of the string attributes $A_1 \mid A_2 \mid \dots \mid A_n$
NMOKEN, NMOKENS	One or more name tokens
ID	A unique ID
IDREF, IDREFS	One or more references to a unique ID
ENTITY, ENTITIES	One or more unparsed entities

Table 2-3. Attribute Defaults

Default	Meaning
#REQUIRED	Attribute is required.
#IMPLIED	Attribute is optional.
A	Attribute is optional; the parser reports it to be A if it is not specified.
#FIXED A	The attribute must either be unspecified or A; in either case, the parser reports it to be A.

Here are two typical attribute specifications:

```
<!ATTLIST font style (plain|bold|italic|bold-italic) "plain">
<!ATTLIST size unit CDATA #IMPLIED>
```

The first specification describes the `style` attribute of a `font` element. There are four legal attribute values, and the default value is `plain`. The second specification expresses that the `unit` attribute of the `size` element can contain any character data sequence.

Note



We generally recommend the use of elements, not attributes, to describe data. Following that recommendation, the font style should be a separate element, such as `<style>plain</style>...`. However, attributes have an undeniable advantage for enumerated types because the parser can verify that the values are legal. For example, if the font style is an attribute, the parser checks that it is one of the four allowed values, and it supplies a default if no value was given.

The handling of a `CDATA` attribute value is subtly different from the processing of `#PCDATA` that you have seen before, and quite unrelated to the `<! [CDATA[...]]>` sections. The attribute value is first normalized; that is, the parser processes character and entity references (such as `é` or `<`) and replaces whitespace with spaces.

An `NMTOKEN` (or name token) is similar to `CDATA`, but most nonalphanumeric characters and internal whitespace are disallowed, and the parser removes leading and trailing whitespace. `NMTOKENS` is a whitespace-separated list of name tokens.

The `ID` construct is quite useful. An `ID` is a name token that must be unique in the document—the parser checks the uniqueness. You will see an application in the next sample program. An `IDREF` is a reference to an ID that exists in the same document—which the parser also checks. `IDREFS` is a whitespace-separated list of ID references.

An `ENTITY` attribute value refers to an "unparsed external entity." That is a holdover from SGML that is rarely used in practice. The annotated XML specification at <http://www.xml.com/axml/axml.html> has an example.

A DTD can also define entities, or abbreviations that are replaced during parsing. You can find a good example for the use of entities in the user interface descriptions for the Mozilla/Netscape 6 browser. Those descriptions are formatted in XML and contain entity definitions such as

```
<!ENTITY back.label "Back">
```

Elsewhere, text can contain an entity reference, for example:

```
<menuitem label="&back.label;\"/>
```

The parser replaces the entity reference with the replacement string. For internationalization of the application, only the string in the entity definition needs to be changed. Other uses of entities are more complex and less commonly used. Look at the XML specification for details.

This concludes the introduction to DTDs. Now that you have seen how to use DTDs, you can configure your parser to take advantage of them. First, tell the document builder factory to turn on validation.

```
factory.setValidating(true);
```

All builders produced by this factory validate their input against a DTD. The most useful benefit of validation is to ignore whitespace in element content. For example, consider the XML fragment

```
<font>
  <name>Helvetica</name>
  <size>36</size>
</font>
```

A nonvalidating parser reports the whitespace between the `font`, `name`, and `size` elements because it has no way of knowing if the children of `font` are

```
(name, size)
(#PCDATA, name, size) *
```

or perhaps

ANY

Once the DTD specifies that the children are `(name, size)`, the parser knows that the whitespace between them is not text. Call

```
factory.setIgnoringElementContentWhitespace(true);
```

and the builder will stop reporting the whitespace in text nodes. That means you can now rely on the fact that a `font` node has two children. You no longer need to program a tedious loop:

```
for (int i = 0; i < children.getLength(); i++)
{
    Node child = children.item(i);
    if (child instanceof Element)
    {
        Element childElement = (Element) child;
        if (childElement.getTagName().equals("name")) . . .
        else if (childElement.getTagName().equals("size")) . . .
    }
}
```

Instead, you can simply access the first and second child:

```
Element nameElement = (Element) children.item(0);
Element sizeElement = (Element) children.item(1);
```

That is why DTDs are so useful. You don't overload your program with rule checking code—the parser has already done that work by the time you get the document.

Tip

 Many programmers who start using XML are uncomfortable with validation and end up analyzing the DOM tree on the fly. If you need to convince colleagues of the benefit of using validated documents, show them the two coding alternatives—it should win them over.

When the parser reports an error, your application will want to do something about it—log it, show it to the user, or throw an exception to abandon the parsing. Therefore, you should install an error handler whenever you use validation. Supply an object that implements the `ErrorHandler` interface. That interface has three methods:

```
void warning(SAXParseException exception)
void error(SAXParseException exception)
void fatalError(SAXParseException exception)
```

You install the error handler with the `setErrorHandler` method of the `DocumentBuilder` class:

```
builder.setErrorHandler(handler);
```



javax.xml.parsers.DocumentBuilder 1.4

- `void setEntityResolver(EntityResolver resolver)`
sets the resolver to locate entities that are referenced in the XML documents to be parsed.
- `void setErrorHandler(ErrorHandler handler)`
sets the handler to report errors and warnings that occur during parsing.



org.xml.sax.EntityResolver 1.4

- `public InputSource resolveEntity(String publicID, String systemID)`

returns an input source that contains the data referenced by the given ID(s), or `null` to indicate that this resolver doesn't know how to resolve the particular name. The `publicID` parameter may be `null` if no public ID was supplied.



org.xml.sax.InputSource 1.4

- `InputSource(InputStream in)`
- `InputSource(Reader in)`
- `InputSource(String systemID)`

constructs an input source from a stream, reader, or system ID (usually a relative or absolute URL).



org.xml.sax.ErrorHandler 1.4

- `void fatalError(SAXParseException exception)`
- `void error(SAXParseException exception)`
- `void warning(SAXParseException exception)`

Override these methods to provide handlers for fatal errors, nonfatal errors, and warnings.



org.xml.sax.SAXParseException 1.4

- `int getLineNumber()`
- `int getColumnNumber()`

returns the line and column number of the end of the processed input that caused the exception.



javax.xml.parsers.DocumentBuilderFactory 1.4

- `boolean isValidating()`
 - `void setValidating(boolean value)`
- gets or sets the `validating` property of the factory. If set to `true`, the parsers that this factory generates validate their input.
- `boolean isIgnoringElementContentWhitespace()`
 - `void setIgnoringElementContentWhitespace(boolean value)`

gets or sets the `ignoringElementContentWhitespace` property of the factory. If set to `true`, the parsers that this factory generates ignore whitespace text between element nodes that don't have mixed content (i.e., a mixture of elements and #PCDATA).

XML Schema

Because XML Schema is quite a bit more complex than the DTD syntax, we cover only the basics. For more information, we

recommend the tutorial at <http://www.w3.org/TR/xmlschema-0>.

To reference a Schema file in a document, add attributes to the root element, for example:

```
<?xml version="1.0"?>
<configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="config.xsd">
    ...
</configuration>
```

This declaration states that the schema file `config.xsd` should be used to validate the document. If your document uses namespaces, the syntax is a bit more complex—see the XML Schema tutorial for details. (The prefix `xsi` is a namespace alias—see the section "[Using Namespaces](#)" on page [136](#) for more information.)

A schema defines a type for each element. The type can be a simple type—a string with formatting restrictions—or a complex type. Some simple types are built into XML Schema, including

```
xsd:string
xsd:int
xsd:boolean
```

Note

 We use the prefix `xsd:` to denote the XML Schema Definition namespace. Some authors use the prefix `xs:` instead.

You can define your own simple types. For example, here is an enumerated type:

```
<xsd:simpleType name="StyleType">
    <xsd:restriction base="xsd:string">
        <xsd:enumeration value="PLAIN" />
        <xsd:enumeration value="BOLD" />
        <xsd:enumeration value="ITALIC" />
        <xsd:enumeration value="BOLD_ITALIC" />
    </xsd:restriction>
</xsd:simpleType>
```

When you define an element, you specify its type:

```
<xsd:element name="name" type="xsd:string"/>
<xsd:element name="size" type="xsd:int"/>
<xsd:element name="style" type="StyleType"/>
```

The type constrains the element content. For example, the elements

```
<size>10</size>
<style>PLAIN</style>
```

will validate correctly, but the elements

```
<size>default</size>
<style>SLANTED</style>
```

will be rejected by the parser.

You can compose types into complex types, for example:

```
<xsd:complexType name="FontType">
    <xsd:sequence>
        <xsd:element ref="name"/>
        <xsd:element ref="size"/>
        <xsd:element ref="style"/>
    </xsd:sequence>
</xsd:complexType>
```

A `FontType` is a sequence of `name`, `size`, and `style` elements. In this type definition, we use the `ref` attribute and refer to definitions that are located elsewhere in the schema. You can also nest definitions, like this:

```
<xsd:complexType name="FontType">
  <xsd:sequence>
    <xsd:element name="name" type="xsd:string"/>
    <xsd:element name="size" type="xsd:int"/>
    <xsd:element name="style" type="StyleType">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="PLAIN" />
          <xsd:enumeration value="BOLD" />
          <xsd:enumeration value="ITALIC" />
          <xsd:enumeration value="BOLD_ITALIC" />
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```

Note the anonymous type definition of the `style` element.

The `xsd:sequence` construct is the equivalent of the concatenation notation in DTDs. The `xsd:choice` construct is the equivalent of the `|` operator. For example,

```
<xsd:complexType name="contactinfo">
  <xsd:choice>
    <xsd:element ref="email"/>
    <xsd:element ref="phone"/>
  </xsd:choice>
</xsd:complexType>
```

This is the equivalent of the DTD type `email|phone`.

To allow repeated elements, you use the `minoccurs` and `maxoccurs` attributes. For example, the equivalent of the DTD type `item*` is

```
<xsd:element name="item" type="..." minoccurs="0" maxoccurs="unbounded">
```

To specify attributes, add `xsd:attribute` elements to `complexType` definitions:

Code View:

```
<xsd:element name="size">
  <xsd:complexType>
    .
    .
    <xsd:attribute name="unit" type="xsd:string" use="optional" default="cm"/>
  </xsd:complexType>
</xsd:element>
```

This is the equivalent of the DTD statement

```
<!ATTLIST size unit CDATA #IMPLIED "cm">
```

You enclose element and type definitions of your schema inside an `xsd:schema` element:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  .
  .
</xsd:schema>
```

Parsing an XML file with a schema is similar to parsing a file with a DTD, but with three differences:

1. You need to turn on support for namespaces, even if you don't use them in your XML files.

```
factory.setNamespaceAware(true);
```

2. You need to prepare the factory for handling schemas, with the following magic incantation:

Code View:

```
final String JAXP_SCHEMA_LANGUAGE = "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
final String W3C_XML_SCHEMA = "http://www.w3.org/2001/XMLSchema";
factory.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
```

3. The parser does not discard element content whitespace. This is a definite annoyance, and there is disagreement whether or not it is an actual bug. See the code in [Listing 2-4](#) on page [122](#) for a workaround.

A Practical Example

In this section, we work through a practical example that shows the use of XML in a realistic setting. Recall from Volume I, Chapter 9 that the `GridBagLayout` is the most useful layout manager for Swing components. However, it is feared not just for its complexity but also for the programming tedium. It would be much more convenient to put the layout instructions into a text file instead of producing large amounts of repetitive code. In this section, you see how to use XML to describe a grid bag layout and how to parse the layout files.

A grid bag is made up of rows and columns, very similar to an HTML table. Similar to an HTML table, we describe it as a sequence of rows, each of which contains cells:

```
<gridbag>
  <row>
    <cell>...</cell>
    <cell>...</cell>
    .
    .
  </row>
  <row>
    <cell>...</cell>
    <cell>...</cell>
    .
    .
  </row>
  .
</gridbag>
```

The `gridbag.dtd` specifies these rules:

```
<!ELEMENT gridbag (row)*>
<!ELEMENT row (cell)*>
```

Some cells can span multiple rows and columns. In the grid bag layout, that is achieved by setting the `gridwidth` and `gridheight` constraints to values larger than 1. We use attributes of the same name:

```
<cell gridwidth="2" gridheight="2">
```

Similarly, we use attributes for the other grid bag constraints `fill`, `anchor`, `gridx`, `gridy`, `weightx`, `weighty`, `ipadx`, and `ipady`. (We don't handle the `insets` constraint because its value is not a simple type, but it would be straightforward to support it.) For example,

```
<cell fill="HORIZONTAL" anchor="NORTH">
```

For most of these attributes, we provide the same defaults as the `GridBagConstraints` default constructor:

```
<!ATTLIST cell gridwidth CDATA "1">
<!ATTLIST cell gridheight CDATA "1">
<!ATTLIST cell fill (NONE|BOTH|HORIZONTAL|VERTICAL) "NONE">
<!ATTLIST cell anchor (CENTER|NORTH|NORTHEAST|EAST
  |SOUTHEAST|SOUTH|SOUTHWEST|WEST|NORTHWEST) "CENTER">
  .
  .
```

The `gridx` and `gridy` values get special treatment because it would be tedious and somewhat error prone to specify them by hand. Supplying them is optional:

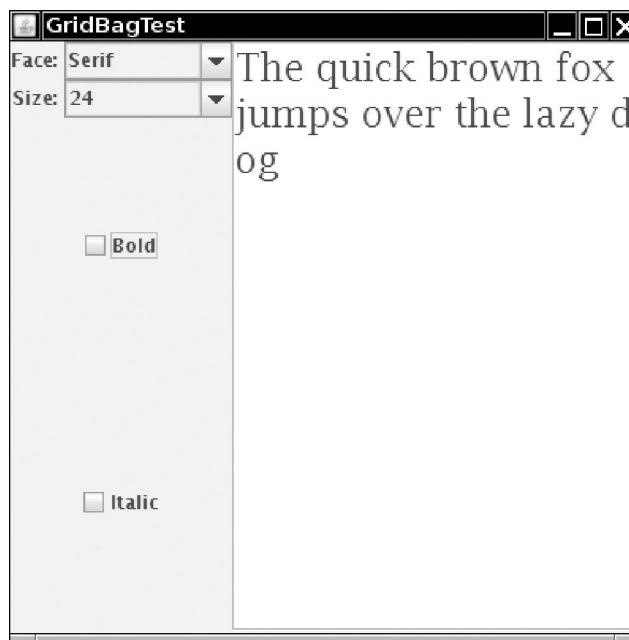
```
<!ATTLIST cell gridx CDATA #IMPLIED>
<!ATTLIST cell gridy CDATA #IMPLIED>
```

If they are not supplied, the program determines them according to the following heuristic: In column 0, the default `gridx` is 0. Otherwise, it is the preceding `gridx` plus the preceding `gridwidth`. The default `gridy` is always the same as the row number. Thus, you don't have to specify `gridx` and `gridy` in the most common cases, in which a component spans multiple rows. However, if a component spans multiple columns, then you must specify `gridx` whenever you skip over that component.

Note

 Grid bag experts might wonder why we don't use the `RELATIVE` and `REMAINDER` mechanism to let the grid bag layout automatically determine the `gridx` and `gridy` positions. We tried, but no amount of fussing would produce the layout of the font dialog example of [Figure 2-4](#). Reading through the `GridBagLayout` source code, it is apparent that the algorithm just won't do the heavy lifting that would be required to recover the absolute positions.

Figure 2-4. A font dialog defined by an XML layout



The program parses the attributes and sets the grid bag constraints. For example, to read the grid width, the program contains a single statement:

```
constraints.gridwidth = Integer.parseInt(e.getAttribute("gridwidth"));
```

The program need not worry about a missing attribute because the parser automatically supplies the default value if no other value was specified in the document.

To test whether a `gridx` or `gridy` attribute was specified, we call the `getAttribute` method and check if it returns the empty string:

```
String value = e.getAttribute("gridy");
if (value.length() == 0) // use default
    constraints.gridx = r;
else
    constraints.gridx = Integer.parseInt(value);
```

We found it convenient to allow arbitrary objects inside cells. That lets us specify noncomponent types such as borders. We only require that the objects belong to a class that follows the JavaBeans convention: to have a default constructor, and to have properties that are given by getter/setter pairs. (We discuss JavaBeans in more detail in [Chapter 8](#).)

A bean is defined by a class name and zero or more properties:

```
<!ELEMENT bean (class, property*)>
<!ELEMENT class (#PCDATA)>
```

A property contains a name and a value.

```
<!ELEMENT property (name, value)>
<!ELEMENT name (#PCDATA)>
```

The value is an integer, boolean, string, or another bean:

```
<!ELEMENT value (int|string|boolean|bean)>
<!ELEMENT int (#PCDATA)>
<!ELEMENT string (#PCDATA)>
<!ELEMENT boolean (#PCDATA)>
```

Here is a typical example, a `JLabel` whose `text` property is set to the string "Face: ".

```
<bean>
  <class>javax.swing.JLabel</class>
  <property>
    <name>text</name>
    <value><string>Face: </string></value>
  </property>
</bean>
```

It seems like a bother to surround a string with the `<string>` tag. Why not just use `#PCDATA` for strings and leave the tags for the other types? Because then we would need to use mixed content and weaken the rule for the `value` element to

```
<!ELEMENT value (#PCDATA|int|boolean|bean)*>
```

However, that rule would allow an arbitrary mixture of text and tags.

The program sets a property by using the `BeanInfo` class. `BeanInfo` enumerates the property descriptors of the bean. We search for the property with the matching name, and then call its setter method with the supplied value.

When our program reads in a user interface description, it has enough information to construct and arrange the user interface components. But, of course, the interface is not alive—no event listeners have been attached. To add event listeners, we have to locate the components. For that reason, we support an optional attribute of type `ID` for each bean:

```
<!ATTLIST bean id ID #IMPLIED>
```

For example, here is a combo box with an ID:

```
<bean id="face">
  <class>javax.swing.JComboBox</class>
</bean>
```

Recall that the parser checks that IDs are unique.

A programmer can attach event handlers like this:

```
gridbag = new GridBagPanel("fontdialog.xml");
setContentPanel(gridbag);
JComboBox face = (JComboBox) gridbag.get("face");
face.addListener(listener);
```

Note



In this example, we only use XML to describe the component layout and leave it to programmers to attach the event handlers in the Java code. You could go a step further and add the code to the XML description. The most promising approach is to use a scripting language such as JavaScript for the code. If you want to add that enhancement, check out the Rhino interpreter at <http://www.mozilla.org/rhino>.

The program in [Listing 2-2](#) shows how to use the `GridBagPanel` class to do all the boring work of setting up the grid bag layout. The layout is defined in [Listing 2-3](#). [Figure 2-4](#) shows the result. The program only initializes the combo boxes (which are too

complex for the bean property-setting mechanism that the `GridBagPane` supports) and attaches event listeners. The `GridBagPane` class in [Listing 2-4](#) parses the XML file, constructs the components, and lays them out. [Listing 2-5](#) shows the DTD.

The program can also process a schema instead of a DTD if you launch it with

```
java GridBagTest fontdialog-schema.xml
```

[Listing 2-6](#) contains the schema.

This example is a typical use of XML. The XML format is robust enough to express complex relationships. The XML parser adds value by taking over the routine job of validity checking and supplying defaults.

Listing 2-2. GridBagTest.java

Code View:

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import javax.swing.*;
4.
5. /**
6. * This program shows how to use an XML file to describe a gridbag layout
7. * @version 1.01 2007-06-25
8. * @author Cay Horstmann
9. */
10. public class GridBagTest
11. {
12.     public static void main(final String[] args)
13.     {
14.         EventQueue.invokeLater(new Runnable()
15.         {
16.             public void run()
17.             {
18.                 String filename = args.length == 0 ? "fontdialog.xml" : args[0];
19.                 JFrame frame = new FontFrame(filename);
20.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
21.                 frame.setVisible(true);
22.             }
23.         });
24.     }
25. }
26.
27. /**
28. * This frame contains a font selection dialog that is described by an XML file.
29. * @param filename the file containing the user interface components for the dialog.
30. */
31. class FontFrame extends JFrame
32. {
33.     public FontFrame(String filename)
34.     {
35.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
36.         setTitle("GridBagTest");
37.
38.         gridbag = new GridBagPane(filename);
39.         add(gridbag);
40.
41.         face = (JComboBox) gridbag.get("face");
42.         size = (JComboBox) gridbag.get("size");
43.         bold = (JCheckBox) gridbag.get("bold");
44.         italic = (JCheckBox) gridbag.get("italic");
45.
46.         face.setModel(new DefaultComboBoxModel(new Object[] { "Serif", "SansSerif",
47.             "Monospaced", "Dialog", "DialogInput" }));
48.
49.         size.setModel(new DefaultComboBoxModel(new Object[] { "8", "10", "12", "15", "18", "24",
50.             "36", "48" }));
51.
52.         ActionListener listener = new ActionListener()
53.         {
54.             public void actionPerformed(ActionEvent event)
55.             {
56.                 setSample();
57.             }
58.         };
59.     }
}
```

```
60.         face.addActionListener(listener);
61.         size.addActionListener(listener);
62.         bold.addActionListener(listener);
63.         italic.addActionListener(listener);
64.         setSample();
65.     }
66.
67. /**
68. * This method sets the text sample to the selected font.
69. */
70. public void setSample()
71. {
72.     String fontFace = (String) face.getSelectedItem();
73.     int fontSize = Integer.parseInt((String) size.getSelectedItem());
74.     JTextArea sample = (JTextArea) gridbag.get("sample");
75.     int fontStyle = (bold.isSelected() ? Font.BOLD : 0)
76.           + (italic.isSelected() ? Font.ITALIC : 0);
77.
78.     sample.setFont(new Font(fontFace, fontStyle, fontSize));
79.     sample.repaint();
80. }
81.
82. private GridBagPane gridbag;
83. private JComboBox face;
84. private JComboBox size;
85. private JCheckBox bold;
86. private JCheckBox italic;
87. private static final int DEFAULT_WIDTH = 400;
88. private static final int DEFAULT_HEIGHT = 400;
89. }
```

Listing 2-3. `fontdialog.xml`

Code View:

```
1. <?xml version="1.0"?>
2. <!DOCTYPE gridbag SYSTEM "gridbag.dtd">
3. <gridbag>
4.   <row>
5.     <cell anchor="EAST">
6.       <bean>
7.         <class>javax.swing.JLabel</class>
8.         <property>
9.           <name>text</name>
10.          <value><string>Face: </string></value>
11.        </property>
12.      </bean>
13.    </cell>
14.    <cell fill="HORIZONTAL" weightx="100">
15.      <bean id="face">
16.        <class>javax.swing.JComboBox</class>
17.      </bean>
18.    </cell>
19.    <cell gridheight="4" fill="BOTH" weightx="100" weighty="100">
20.      <bean id="sample">
21.        <class>javax.swing.JTextArea</class>
22.        <property>
23.          <name>text</name>
24.          <value><string>The quick brown fox jumps over the lazy dog</string></value>
25.        </property>
26.        <property>
27.          <name>editable</name>
28.          <value><boolean>false</boolean></value>
29.        </property>
30.        <property>
31.          <name>lineWrap</name>
32.          <value><boolean>true</boolean></value>
33.        </property>
34.        <property>
35.          <name>border</name>
36.          <value>
37.            <bean>
```

```
38.          <class>javax.swing.border.EtchedBorder</class>
39.          </bean>
40.          </value>
41.          </property>
42.          </bean>
43.        </cell>
44.      </row>
45.    <row>
46.      <cell anchor="EAST">
47.        <bean>
48.          <class>javax.swing.JLabel</class>
49.          <property>
50.            <name>text</name>
51.            <value><string>Size: </string></value>
52.          </property>
53.        </bean>
54.      </cell>
55.      <cell fill="HORIZONTAL" weightx="100">
56.        <bean id="size">
57.          <class>javax.swing.JComboBox</class>
58.        </bean>
59.      </cell>
60.    </row>
61.    <row>
62.      <cell gridwidth="2" weighty="100">
63.        <bean id="bold">
64.          <class>javax.swing.JCheckBox</class>
65.          <property>
66.            <name>text</name>
67.            <value><string>Bold</string></value>
68.          </property>
69.        </bean>
70.      </cell>
71.    </row>
72.    <row>
73.      <cell gridwidth="2" weighty="100">
74.        <bean id="italic">
75.          <class>javax.swing.JCheckBox</class>
76.          <property>
77.            <name>text</name>
78.            <value><string>Italic</string></value>
79.          </property>
80.        </bean>
81.      </cell>
82.    </row>
83.  </gridbag>
```

Listing 2-4. GridBagPane.java

Code View:

```
1. import java.awt.*;
2. import java.beans.*;
3. import java.io.*;
4. import java.lang.reflect.*;
5. import javax.swing.*;
6. import javax.xml.parsers.*;
7. import org.w3c.dom.*;
8.
9. /**
10. * This panel uses an XML file to describe its components and their grid bag layout positions.
11. * @version 1.10 2004-09-04
12. * @author Cay Horstmann
13. */
14. public class GridBagPane extends JPanel
15. {
16.   /**
17.    * Constructs a grid bag pane.
18.    * @param filename the name of the XML file that describes the pane's components and their
19.    * positions
20.    */
21.   public GridBagPane(String filename)
```

```
22. {
23.     setLayout(new GridBagLayout());
24.     constraints = new GridBagConstraints();
25.
26.     try
27.     {
28.         DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
29.         factory.setValidating(true);
30.
31.         if (filename.contains("-schema"))
32.         {
33.             factory.setNamespaceAware(true);
34.             final String JAXP_SCHEMA_LANGUAGE = "http://java.sun.com/xml/jaxp/properties/
35.                                         schemaLanguage";
36.             final String W3C_XML_SCHEMA = "http://www.w3.org/2001/XMLSchema";
37.             factory.setAttribute(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
38.         }
39.
40.         factory.setIgnoringElementContentWhitespace(true);
41.
42.         DocumentBuilder builder = factory.newDocumentBuilder();
43.         Document doc = builder.parse(new File(filename));
44.
45.         if (filename.contains("-schema"))
46.         {
47.             int count = removeElementContentWhitespace(doc.getDocumentElement());
48.             System.out.println(count + " whitespace nodes removed.");
49.         }
50.
51.         parseGridbag(doc.getDocumentElement());
52.     }
53.     catch (Exception e)
54.     {
55.         e.printStackTrace();
56.     }
57. }
58.
59. /**
60. * Removes all (heuristically determined) element content whitespace nodes
61. * @param e the root element
62. * @return the number of whitespace nodes that were removed.
63. */
64. private int removeElementContentWhitespace(Element e)
65. {
66.     NodeList children = e.getChildNodes();
67.     int count = 0;
68.     boolean allTextChildrenAreWhiteSpace = true;
69.     int elements = 0;
70.     for (int i = 0; i < children.getLength() && allTextChildrenAreWhiteSpace; i++)
71.     {
72.         Node child = children.item(i);
73.         if (child instanceof Text && ((Text) child).getData().trim().length() > 0)
74.             allTextChildrenAreWhiteSpace = false;
75.         else if (child instanceof Element)
76.         {
77.             elements++;
78.             count += removeElementContentWhitespace((Element) child);
79.         }
80.     }
81.     if (elements > 0 && allTextChildrenAreWhiteSpace) // heuristics for element content
82.     {
83.         for (int i = children.getLength() - 1; i >= 0; i--)
84.         {
85.             Node child = children.item(i);
86.             if (child instanceof Text)
87.             {
88.                 e.removeChild(child);
89.                 count++;
90.             }
91.         }
92.     }
93.     return count;
94. }
95.
96. /**
97. * Gets a component with a given name
98. * @param name a component name
99. * @return the component with the given name, or null if no component in this grid bag
100. * pane has the given name
101. */
102. public Component get(String name)
```

```
103.     {
104.         Component[] components = getComponents();
105.         for (int i = 0; i < components.length; i++)
106.         {
107.             if (components[i].getName().equals(name)) return components[i];
108.         }
109.         return null;
110.     }
111.
112. /**
113. * Parses a gridbag element.
114. * @param e a gridbag element
115. */
116. private void parseGridbag(Element e)
117. {
118.     NodeList rows = e.getChildNodes();
119.     for (int i = 0; i < rows.getLength(); i++)
120.     {
121.         Element row = (Element) rows.item(i);
122.         NodeList cells = row.getChildNodes();
123.         for (int j = 0; j < cells.getLength(); j++)
124.         {
125.             Element cell = (Element) cells.item(j);
126.             parseCell(cell, i, j);
127.         }
128.     }
129. }
130.
131. /**
132. * Parses a cell element.
133. * @param e a cell element
134. * @param r the row of the cell
135. * @param c the column of the cell
136. */
137. private void parseCell(Element e, int r, int c)
138. {
139.     // get attributes
140.
141.     String value = e.getAttribute("gridx");
142.     if (value.length() == 0) // use default
143.     {
144.         if (c == 0) constraints.gridx = 0;
145.         else constraints.gridx += constraints.gridwidth;
146.     }
147.     else constraints.gridx = Integer.parseInt(value);
148.
149.     value = e.getAttribute("gridy");
150.     if (value.length() == 0) // use default
151.     constraints.gridy = r;
152.     else constraints.gridy = Integer.parseInt(value);
153.
154.     constraints.gridwidth = Integer.parseInt(e.getAttribute("gridwidth"));
155.     constraints.gridheight = Integer.parseInt(e.getAttribute("gridheight"));
156.     constraints.weightx = Integer.parseInt(e.getAttribute("weightx"));
157.     constraints.weighty = Integer.parseInt(e.getAttribute("weighty"));
158.     constraints.ipadx = Integer.parseInt(e.getAttribute("ipadx"));
159.     constraints.ipady = Integer.parseInt(e.getAttribute("ipady"));
160.
161.     // use reflection to get integer values of static fields
162.     Class<GridBagConstraints> cl = GridBagConstraints.class;
163.
164.     try
165.     {
166.         String name = e.getAttribute("fill");
167.         Field f = cl.getField(name);
168.         constraints.fill = f.getInt(cl);
169.
170.         name = e.getAttribute("anchor");
171.         f = cl.getField(name);
172.         constraints.anchor = f.getInt(cl);
173.     }
174.     catch (Exception ex) // the reflection methods can throw various exceptions
175.     {
176.         ex.printStackTrace();
177.     }
178.
179.     Component comp = (Component) parseBean((Element) e.getFirstChild());
180.     add(comp, constraints);
181. }
182.
183. /**
```

```
184.     * Parses a bean element.
185.     * @param e a bean element
186.     */
187.    private Object parseBean(Element e)
188.    {
189.        try
190.        {
191.            NodeList children = e.getChildNodes();
192.            Element classElement = (Element) children.item(0);
193.            String className = ((Text) classElement.getFirstChild()).getData();
194.
195.            Class<?> cl = Class.forName(className);
196.
197.            Object obj = cl.newInstance();
198.
199.            if (obj instanceof Component) ((Component) obj).setName(e.getAttribute("id"));
200.
201.            for (int i = 1; i < children.getLength(); i++)
202.            {
203.                Node propertyElement = children.item(i);
204.                Element nameElement = (Element) propertyElement.getFirstChild();
205.                String propertyName = ((Text) nameElement.getFirstChild()).getData();
206.
207.                Element valueElement = (Element) propertyElement.getLastChild();
208.                Object value = parseValue(valueElement);
209.                BeanInfo beanInfo = Introspector.getBeanInfo(cl);
210.                PropertyDescriptor[] descriptors = beanInfo.getPropertyDescriptors();
211.                boolean done = false;
212.                for (int j = 0; !done && j < descriptors.length; j++)
213.                {
214.                    if (descriptors[j].getName().equals(propertyName))
215.                    {
216.                        descriptors[j].getWriteMethod().invoke(obj, value);
217.                        done = true;
218.                    }
219.                }
220.
221.            }
222.            return obj;
223.        }
224.        catch (Exception ex) // the reflection methods can throw various exceptions
225.        {
226.            ex.printStackTrace();
227.            return null;
228.        }
229.    }
230.
231. /**
232.     * Parses a value element.
233.     * @param e a value element
234.     */
235.    private Object parseValue(Element e)
236.    {
237.        Element child = (Element) e.getFirstChild();
238.        if (child.getTagName().equals("bean")) return parseBean(child);
239.        String text = ((Text) child.getFirstChild()).getData();
240.        if (child.getTagName().equals("int")) return new Integer(text);
241.        else if (child.getTagName().equals("boolean")) return new Boolean(text);
242.        else if (child.getTagName().equals("string")) return text;
243.        else return null;
244.    }
245.
246.    private GridBagConstraints constraints;
247. }
```

Listing 2-5. gridbag.dtd

Code View:

```
1. <!ELEMENT gridbag (row)*>
2. <!ELEMENT row (cell)*>
3. <!ELEMENT cell (bean)>
```

```
4. <!ATTLIST cell gridx CDATA #IMPLIED>
5. <!ATTLIST cell gridy CDATA #IMPLIED>
6. <!ATTLIST cell gridwidth CDATA "1">
7. <!ATTLIST cell gridheight CDATA "1">
8. <!ATTLIST cell weightx CDATA "0">
9. <!ATTLIST cell weighty CDATA "0">
10. <!ATTLIST cell fill (NONE|BOTH|HORIZONTAL|VERTICAL) "NONE">
11. <!ATTLIST cell anchor
12.     (CENTER|NORTH|NORTHEAST|EAST|SOUTHEAST|SOUTH|SOUTHWEST|WEST|NORTHWEST) "CENTER">
13. <!ATTLIST cell ipadx CDATA "0">
14. <!ATTLIST cell ipady CDATA "0">
15.
16. <!ELEMENT bean (class, property*)>
17. <!ATTLIST bean id ID #IMPLIED>
18.
19. <!ELEMENT class (#PCDATA)>
20. <!ELEMENT property (name, value)>
21. <!ELEMENT name (#PCDATA)>
22. <!ELEMENT value (int|string|boolean|bean)>
23. <!ELEMENT int (#PCDATA)>
24. <!ELEMENT string (#PCDATA)>
25. <!ELEMENT boolean (#PCDATA)>
```

Listing 2-6. `gridbag.xsd`

Code View:

```
1. <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
2.
3.     <xsd:element name="gridbag" type="GridBagType"/>
4.
5.     <xsd:element name="bean" type="BeanType"/>
6.
7.     <xsd:complexType name="GridBagType">
8.         <xsd:sequence>
9.             <xsd:element name="row" type="RowType" minOccurs="0" maxOccurs="unbounded"/>
10.            </xsd:sequence>
11.        </xsd:complexType>
12.
13.     <xsd:complexType name="RowType">
14.         <xsd:sequence>
15.             <xsd:element name="cell" type="CellType" minOccurs="0" maxOccurs="unbounded"/>
16.         </xsd:sequence>
17.     </xsd:complexType>
18.
19.     <xsd:complexType name="CellType">
20.         <xsd:sequence>
21.             <xsd:element ref="bean"/>
22.         </xsd:sequence>
23.             <xsd:attribute name="gridx" type="xsd:int" use="optional"/>
24.             <xsd:attribute name="gridy" type="xsd:int" use="optional"/>
25.             <xsd:attribute name="gridwidth" type="xsd:int" use="optional" default="1" />
26.             <xsd:attribute name="gridheight" type="xsd:int" use="optional" default="1" />
27.             <xsd:attribute name="weightx" type="xsd:int" use="optional" default="0" />
28.             <xsd:attribute name="weighty" type="xsd:int" use="optional" default="0" />
29.             <xsd:attribute name="fill" use="optional" default="NONE">
30.                 <xsd:simpleType>
31.                     <xsd:restriction base="xsd:string">
32.                         <xsd:enumeration value="NONE" />
33.                         <xsd:enumeration value="BOTH" />
34.                         <xsd:enumeration value="HORIZONTAL" />
35.                         <xsd:enumeration value="VERTICAL" />
36.                     </xsd:restriction>
37.                 </xsd:simpleType>
38.             </xsd:attribute>
39.             <xsd:attribute name="anchor" use="optional" default="CENTER">
40.                 <xsd:simpleType>
41.                     <xsd:restriction base="xsd:string">
42.                         <xsd:enumeration value="CENTER" />
43.                         <xsd:enumeration value="NORTH" />
44.                         <xsd:enumeration value="NORTHEAST" />
45.                         <xsd:enumeration value="EAST" />
46.                         <xsd:enumeration value="SOUTHEAST" />
47.                         <xsd:enumeration value="SOUTH" />
```

```
48.          <xsd:enumeration value="SOUTHWEST" />
49.          <xsd:enumeration value="WEST" />
50.          <xsd:enumeration value="NORTHWEST" />
51.          </xsd:restriction>
52.        </xsd:simpleType>
53.      </xsd:attribute>
54.      <xsd:attribute name="ipady" type="xsd:int" use="optional" default="0" />
55.      <xsd:attribute name="ipadx" type="xsd:int" use="optional" default="0" />
56.    </xsd:complexType>
57.
58.    <xsd:complexType name="BeanType">
59.      <xsd:sequence>
60.        <xsd:element name="class" type="xsd:string"/>
61.        <xsd:element name="property" type="PropertyType" minOccurs="0" maxOccurs="unbounded"/>
62.      </xsd:sequence>
63.      <xsd:attribute name="id" type="xsd:ID" use="optional" />
64.    </xsd:complexType>
65.
66.    <xsd:complexType name="PropertyType">
67.      <xsd:sequence>
68.        <xsd:element name="name" type="xsd:string"/>
69.        <xsd:element name="value" type="ValueType"/>
70.      </xsd:sequence>
71.    </xsd:complexType>
72.
73.    <xsd:complexType name="ValueType">
74.      <xsd:choice>
75.        <xsd:element ref="bean"/>
76.        <xsd:element name="int" type="xsd:int"/>
77.        <xsd:element name="string" type="xsd:string"/>
78.        <xsd:element name="boolean" type="xsd:boolean"/>
79.      </xsd:choice>
80.    </xsd:complexType>
81.  </xsd:schema>
```



Locating Information with XPath

If you want to locate a specific piece of information in an XML document, then it can be a bit of a hassle to navigate the nodes of the DOM tree. The XPath language makes it simple to access tree nodes. For example, suppose you have this XML document:

```
<configuration>
  . .
  <database>
    <username>dbuser</username>
    <password>secret</password>
  . .
</database>
</configuration>
```

You can get the database user name by evaluating the XPath expression

```
/configuration/database/username
```

That's a lot simpler than the plain DOM approach:

1. Get the document node.
2. Enumerate its children.
3. Locate the `database` element.
4. Get its first child, the `username` element.

5. Get its first child, a `Text` node.

6. Get its data.

An XPath can describe a set of nodes in an XML document. For example, the XPath

```
/gridbag/row
```

describes the set of all `row` elements that are children of the `gridbag` root element. You can select a particular element with the `[]` operator:

```
/gridbag/row[1]
```

is the first row. (The index values start at 1.)

Use the `@` operator to get attribute values. The XPath expression

```
/gridbag/row[1]/cell[1]/@anchor
```

describes the `anchor` attribute of the first cell in the first row. The XPath expression

```
/gridbag/row/cell/@anchor
```

describes all `anchor` attribute nodes of `cell` elements within `row` elements that are children of the `gridbag` root node.

There are a number of useful XPath functions. For example,

```
count(/gridbag/row)
```

returns the number of `row` children of the `gridbag` root. There are many more elaborate XPath expressions—see the specification at <http://www.w3c.org/TR/xpath> or the nifty online tutorial at <http://www.zvon.org/xsl/XPathTutorial/General/examples.html>.

Java SE 5.0 added an API to evaluate XPath expressions. You first create an `XPath` object from an `XPathFactory`:

```
XPathFactory xpfactory = XPathFactory.newInstance();
path = xpfactory.newXPath();
```

You then call the `evaluate` method to evaluate XPath expressions:

```
String username = path.evaluate("/configuration/database/username", doc);
```

You can use the same `XPath` object to evaluate multiple expressions.

This form of the `evaluate` method returns a string result. It is suitable for retrieving text, such as the text of the `username` node in the preceding example. If an XPath expression yields a node set, make a call such as the following:

Code View:

```
NodeList nodes = (NodeList) path.evaluate("/gridbag/row", doc, XPathConstants.NODESET);
```

If the result is a single node, use `XPathConstants.NODE` instead:

Code View:

```
Node node = (Node) path.evaluate("/gridbag/row[1]", doc, XPathConstants.NODE);
```

If the result is a number, use `XPathConstants.NUMBER`:

Code View:

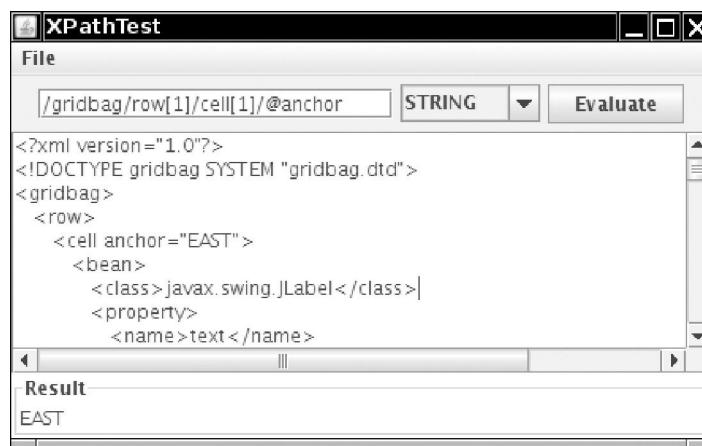
```
int count = ((Number) path.evaluate("count(/gridbag/row)", doc, XPathConstants.NUMBER)).intValue();
```

You don't have to start the search at the document root. You can start the search at any node or node list. For example, if you have a node from a previous evaluation, you can call

```
result = path.evaluate(expression, node);
```

The program in [Listing 2-7](#) demonstrates the evaluation of XPath expressions. Load an XML file and type an expression. Select the expression type and click the Evaluate button. The result of the expression is displayed at the bottom of the frame (see [Figure 2-5](#)).

Figure 2-5. Evaluating XPath expressions



Listing 2-7. `XPathTest.java`

Code View:

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.io.*;
4. import javax.swing.*;
5. import javax.swing.border.*;
6. import javax.xml.namespace.*;
7. import javax.xml.parsers.*;
8. import javax.xml.xpath.*;
9. import org.w3c.dom.*;
10. import org.xml.sax.*;
11.
12. /**
13. * This program evaluates XPath expressions
14. * @version 1.01 2007-06-25
15. * @author Cay Horstmann
16. */
17. public class XPathTest
18. {
19.     public static void main(String[] args)
20.     {
21.         EventQueue.invokeLater(new Runnable()
22.         {
23.             public void run()
24.             {
25.                 JFrame frame = new XPathFrame();
26.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
27.                 frame.setVisible(true);
28.             }
29.         });
30.     }
}
```

```
31. }
32.
33. /**
34. * This frame shows an XML document, a panel to type an XPath expression, and a text field
35. * to display the result.
36. */
37. class XPathFrame extends JFrame
38. {
39.     public XPathFrame()
40.     {
41.         setTitle("XPathTest");
42.
43.         JMenu fileMenu = new JMenu("File");
44.         JMenuItem openItem = new JMenuItem("Open");
45.         openItem.addActionListener(new ActionListener()
46.         {
47.             public void actionPerformed(ActionEvent event)
48.             {
49.                 openFile();
50.             }
51.         });
52.         fileMenu.add(openItem);
53.
54.         JMenuItem exitItem = new JMenuItem("Exit");
55.         exitItem.addActionListener(new ActionListener()
56.         {
57.             public void actionPerformed(ActionEvent event)
58.             {
59.                 System.exit(0);
60.             }
61.         });
62.         fileMenu.add(exitItem);
63.
64.         JMenuBar menuBar = new JMenuBar();
65.         menuBar.add(fileMenu);
66.         setJMenuBar(menuBar);
67.
68.         ActionListener listener = new ActionListener()
69.         {
70.             public void actionPerformed(ActionEvent event)
71.             {
72.                 evaluate();
73.             }
74.         };
75.         expression = new JTextField(20);
76.         expression.addActionListener(listener);
77.         JButton evaluateButton = new JButton("Evaluate");
78.         evaluateButton.addActionListener(listener);
79.
80.         typeCombo = new JComboBox(new Object[] { "STRING", "NODE", "NODESET", "NUMBER",
81.                                     "BOOLEAN" });
82.         typeCombo.setSelectedItem("STRING");
83.
84.         JPanel panel = new JPanel();
85.         panel.add(expression);
86.         panel.add(typeCombo);
87.         panel.add(evaluateButton);
88.         docText = new JTextArea(10, 40);
89.         result = new JTextField();
90.         result.setBorder(new TitledBorder("Result"));
91.
92.         add(panel, BorderLayout.NORTH);
93.         add(new JScrollPane(docText), BorderLayout.CENTER);
94.         add(result, BorderLayout.SOUTH);
95.
96.         try
97.         {
98.             DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
99.             builder = factory.newDocumentBuilder();
100.        }
101.        catch (ParserConfigurationException e)
102.        {
103.            JOptionPane.showMessageDialog(this, e);
104.        }
105.
106.        XPathFactory xpfactory = XPathFactory.newInstance();
107.        path = xpfactory.newXPath();
108.        pack();
109.    }
110.
111.    /**

```

```
112.     * Open a file and load the document.
113.     */
114.     public void openFile()
115.     {
116.         JFileChooser chooser = new JFileChooser();
117.         chooser.setCurrentDirectory(new File("."));
118.
119.         chooser.setFileFilter(new javax.swing.filechooser.FileFilter()
120.             {
121.                 public boolean accept(File f)
122.                 {
123.                     return f.isDirectory() || f.getName().toLowerCase().endsWith(".xml");
124.                 }
125.
126.                 public String getDescription()
127.                 {
128.                     return "XML files";
129.                 }
130.             });
131.         int r = chooser.showOpenDialog(this);
132.         if (r != JFileChooser.APPROVE_OPTION) return;
133.         File f = chooser.getSelectedFile();
134.         try
135.         {
136.             byte[] bytes = new byte[(int) f.length()];
137.             new FileInputStream(f).read(bytes);
138.             docText.setText(new String(bytes));
139.             doc = builder.parse(f);
140.         }
141.         catch (IOException e)
142.         {
143.             JOptionPane.showMessageDialog(this, e);
144.         }
145.         catch (SAXException e)
146.         {
147.             JOptionPane.showMessageDialog(this, e);
148.         }
149.     }
150.
151.     public void evaluate()
152.     {
153.         try
154.         {
155.             String typeName = (String) typeCombo.getSelectedItem();
156.             QName returnType = (QName) XPathConstants.class.getField(typeName).get(null);
157.             Object evalResult = path.evaluate(expression.getText(), doc, returnType);
158.             if (typeName.equals("NODESET"))
159.             {
160.                 NodeList list = (NodeList) evalResult;
161.                 StringBuilder builder = new StringBuilder();
162.                 builder.append("{");
163.                 for (int i = 0; i < list.getLength(); i++)
164.                 {
165.                     if (i > 0) builder.append(", ");
166.                     builder.append("") + list.item(i));
167.                 }
168.                 builder.append("}");
169.                 result.setText("") + builder);
170.             }
171.             else result.setText("") + evalResult);
172.         }
173.         catch (XPathExpressionException e)
174.         {
175.             result.setText("") + e);
176.         }
177.         catch (Exception e) // reflection exception
178.         {
179.             e.printStackTrace();
180.         }
181.     }
182.
183.     private DocumentBuilder builder;
184.     private Document doc;
185.     private XPath path;
186.     private JTextField expression;
187.     private JTextField result;
188.     private JTextArea docText;
189.     private JComboBox typeCombo;
190. }
```

API

javax.xml.xpath.XPathFactory 5.0

- static XPathFactory newInstance()
returns an `XPathFactory` instance for creating `XPath` objects.
- XPath newPath()
constructs an `XPath` object for evaluating `XPath` expressions.

API

javax.xml.xpath.XPath 5.0

- String evaluate(String expression, Object startingPoint)
evaluates an expression, beginning with the given starting point. The starting point can be a node or node list. If the result is a node or node set, then the returned string consists of the data of all text node children.
- Object evaluate(String expression, Object startingPoint, QName resultType)
evaluates an expression, beginning with the given starting point. The starting point can be a node or node list. The `resultType` is one of the constants `STRING`, `NODE`, `NODESET`, `NUMBER`, or `BOOLEAN` in the `XPathConstants` class. The return value is a `String`, `Node`, `NodeList`, `Number`, or `Boolean`.



Using Namespaces

The Java language uses packages to avoid name clashes. Programmers can use the same name for different classes as long as they aren't in the same package. XML has a similar namespace mechanism for element and attribute names.

A namespace is identified by a Uniform Resource Identifier (URI), such as

`http://www.w3.org/2001/XMLSchema`
`uuid:1c759aed-b748-475c-ab68-10679700c4f2`
`urn:com:books-r-us`

The HTTP URL form is the most common. Note that the URL is just used as an identifier string, not as a locator for a document. For example, the namespace identifiers

`http://www.horstmann.com/corejava`
`http://www.horstmann.com/corejava/index.html`

denote different namespaces, even though a web server would serve the same document for both URLs.

There need not be any document at a namespace URL—the XML parser doesn't attempt to find anything at that location. However, as a help to programmers who encounter a possibly unfamiliar namespace, it is customary to place a document explaining the purpose of the namespace at the URL location. For example, if you point your browser to the namespace URL for the XML Schema namespace (<http://www.w3.org/2001/XMLSchema>), you will find a document describing the XML Schema standard.

Why use HTTP URLs for namespace identifiers? It is easy to ensure that they are unique. If you choose a real URL, then the host part's uniqueness is guaranteed by the domain name system. Your organization can then arrange for the uniqueness of the remainder of the URL. This is the same rationale that underlies the use of reversed domain names in Java package names.

Of course, although you want long namespace identifiers for uniqueness, you don't want to deal with long identifiers any more than you have to. In the Java programming language, you use the `import` mechanism to specify the long names of packages, and then use just the short class names. In XML, there is a similar mechanism, like this:

```
<element xmlns="namespaceURI">
    children
</element>
```

The element and its children are now part of the given namespace.

A child can provide its own namespace, for example:

```
<element xmlns="namespaceURI_1">
    <child xmlns="namespaceURI_2">
        grandchildren
    </child>
    more children
</element>
```

Then the first child and the grandchildren are part of the second namespace.

That simple mechanism works well if you need only a single namespace or if the namespaces are naturally nested. Otherwise, you will want to use a second mechanism that has no analog in Java. You can have an alias for a namespace—a short identifier that you choose for a particular document. Here is a typical example:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="gridbag" type="GridBagType"/>
    .
    .
</xsd:schema>
```

The attribute

```
xmlns:alias="namespaceURI"
```

defines a namespace and an alias. In our example, the alias is the string `xsd`. Thus, `xsd:schema` really means "schema in the namespace <http://www.w3.org/2001/XMLSchema>".

Note



Only child elements inherit the namespace of their parent. Attributes without an explicit alias prefix are never part of a namespace. Consider this contrived example:

```
<configuration xmlns="http://www.horstmann.com/corejava"
    xmlns:si="http://www.bipm.fr/enus/3_SI/si.html">
    <size value="210" si:unit="mm"/>
    .
    .
</configuration>
```

In this example, the elements `configuration` and `size` are part of the namespace with URI <http://www.horstmann.com/corejava>. The attribute `si:unit` is part of the namespace with URI http://www.bipm.fr/enus/3_SI/si.html. However, the attribute `value` is not part of any namespace.

You can control how the parser deals with namespaces. By default, the Sun DOM parser is not "namespace aware."

To turn on namespace handling, call the `setNamespaceAware` method of the `DocumentBuilderFactory`:

```
factory.setNamespaceAware(true);
```

Then all builders the factory produces support namespaces. Each node has three properties:

- The qualified name, with an alias prefix, returned by `getnodeName`, `getTagName`, and so on.
- The namespace URI, returned by the `getNamespaceURI` method.

- The local name, without an alias prefix or a namespace, returned by the `getLocalName` method.

Here is an example. Suppose the parser sees the following element:

```
<xsd:schema xmlns:xsl="http://www.w3.org/2001/XMLSchema">
```

It then reports the following:

- Qualified name = `xsd:schema`
- Namespace URI = <http://www.w3.org/2001/XMLSchema>
- Local name = `schema`

Note



If namespace awareness is turned off, then `getNamespaceURI` and `getLocalName` return `null`.

API

org.w3c.dom.Node 1.4

- `String getLocalName()`
returns the local name (without alias prefix), or `null` if the parser is not namespace aware.
- `String getNamespaceURI()`
returns the namespace URI, or `null` if the node is not part of a namespace or if the parser is not namespace aware.

API

javax.xml.parsers.DocumentBuilderFactory 1.4

- `boolean isNamespaceAware()`
- `void setNamespaceAware(boolean value)`
gets or sets the `namespaceAware` property of the factory. If set to `true`, the parsers that this factory generates are namespace aware.



Streaming Parsers

The DOM parser reads an XML document in its entirety into a tree data structure. For most practical applications, DOM works fine. However, it can be inefficient if the document is large and if your processing algorithm is simple enough that you can analyze nodes on the fly, without having to see all of the tree structure. In these cases, you should use a streaming parser.

In the following sections, we discuss the streaming parsers supplied by the Java library: the venerable SAX parser and the more modern StAX parser that was added to Java SE 6. The SAX parser uses event callbacks, and the StAX parser provides an iterator through the parsing events. The latter is usually a bit more convenient.

Using the SAX Parser

The SAX parser reports events as it parses the components of the XML input, but it does not store the document in any way—it is up to the event handlers whether they want to build a data structure. In fact, the DOM parser is built on top of the SAX parser. It builds the DOM tree as it receives the parser events.

Whenever you use a SAX parser, you need a handler that defines the event actions for the various parse events. The `ContentHandler` interface defines several callback methods that the parser executes as it parses the document. Here are the

most important ones:

- `startElement` and `endElement` are called each time a start tag or end tag is encountered.
- `characters` is called whenever character data are encountered.
- `startDocument` and `endDocument` are called once each, at the start and the end of the document.

For example, when parsing the fragment

```
<font>
  <name>Helvetica</name>
  <size units="pt">36</size>
</font>
```

the parser makes the following callbacks:

1. `startElement`, element name: `font`
2. `startElement`, element name: `name`
3. `characters`, content: `Helvetica`
4. `endElement`, element name: `name`
5. `startElement`, element name: `size`, attributes: `units="pt"`
6. `characters`, content: `36`
7. `endElement`, element name: `size`
8. `endElement`, element name: `font`

Your handler needs to override these methods and have them carry out whatever action you want to carry out as you parse the file. The program at the end of this section prints all links `` in an HTML file. It simply overrides the `startElement` method of the handler to check for links with name `a` and an attribute with name `href`. This is potentially useful for implementing a "web crawler," a program that reaches more and more web pages by following links.

Note



Unfortunately, many HTML pages deviate so much from proper XML that the example program will not be able to parse them. As already mentioned, the W3C recommends that web designers use XHTML, an HTML dialect that can be displayed by current web browsers and that is also proper XML. Because the W3C "eats its own dog food," their web pages are written in XHTML. You can use those pages to test the example program. For example, if you run

```
java SAXTest http://www.w3c.org/MarkUp
```

then you will see a list of the URLs of all links on that page.

The sample program is a good example for the use of SAX. We don't care at all in which context the `a` elements occur, and there is no need to store a tree structure.

Here is how you get a SAX parser:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();
```

You can now process a document:

```
parser.parse(source, handler);
```

Here, `source` can be a file, URL string, or input stream. The `handler` belongs to a subclass of `DefaultHandler`. The `DefaultHandler` class defines do-nothing methods for the four interfaces:

```
ContentHandler
DTDHandler
EntityResolver
ErrorHandler
```

The example program defines a handler that overrides the `startElement` method of the `ContentHandler` interface to watch out for `a` elements with an `href` attribute:

Code View:

```
DefaultHandler handler = new
    DefaultHandler()
{
    public void startElement(String namespaceURI, String lname, String qname, Attributes attrs)
        throws SAXException
    {
        if (lname.equalsIgnoreCase("a") && attrs != null)
        {
            for (int i = 0; i < attrs.getLength(); i++)
            {
                String aname = attrs.getLocalName(i);
                if (aname.equalsIgnoreCase("href"))
                    System.out.println(attrs.getValue(i));
            }
        }
    }
};
```

The `startElement` method has three parameters that describe the element name. The `qname` parameter reports the qualified name of the form `alias:localname`. If namespace processing is turned on, then the `namespaceURI` and `lname` parameters describe the namespace and local (unqualified) name.

As with the DOM parser, namespace processing is turned off by default. You activate namespace processing by calling the `setNamespaceAware` method of the factory class:

```
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setNamespaceAware(true);
SAXParser saxParser = factory.newSAXParser();
```

[Listing 2-8](#) contains the code for the web crawler program. Later in this chapter, you will see another interesting use of SAX. An easy way of turning a non-XML data source into XML is to report the SAX events that an XML parser would report. See the section "[XSL Transformations](#)" on page [157](#) for details.

Listing 2-8. SAXTest.java

Code View:

```
1. import java.io.*;
2. import java.net.*;
3. import javax.xml.parsers.*;
4. import org.xml.sax.*;
5. import org.xml.sax.helpers.*;
6.
7. /**
8. * This program demonstrates how to use a SAX parser. The program prints all hyperlinks links
9. * of an XHTML web page. <br>
10. * Usage: java SAXTest url
11. * @version 1.00 2001-09-29
12. * @author Cay Horstmann
13. */
14. public class SAXTest
15. {
16.     public static void main(String[] args) throws Exception
17.     {
18.         String url;
19.         if (args.length == 0)
20.         {
21.             url = "http://www.w3c.org";
22.             System.out.println("Using " + url);
23.         }
24.     }
25. }
```

```
24.     else url = args[0];
25.
26.     DefaultHandler handler = new DefaultHandler()
27.     {
28.         public void startElement(String namespaceURI, String lname, String qname,
29.                                 Attributes attrs)
30.         {
31.             if (lname.equals("a") && attrs != null)
32.             {
33.                 for (int i = 0; i < attrs.getLength(); i++)
34.                 {
35.                     String aname = attrs.getLocalName(i);
36.                     if (aname.equals("href")) System.out.println(attrs.getValue(i));
37.                 }
38.             }
39.         }
40.     };
41.
42.     SAXParserFactory factory = SAXParserFactory.newInstance();
43.     factory.setNamespaceAware(true);
44.     SAXParser saxParser = factory.newSAXParser();
45.     InputStream in = new URL(url).openStream();
46.     saxParser.parse(in, handler);
47. }
48. }
```



javax.xml.parsers.SAXParserFactory 1.4

- static SAXParserFactory newInstance()

returns an instance of the `SAXParserFactory` class.

- SAXParser newSAXParser()

returns an instance of the `SAXParser` class.

- boolean isNamespaceAware()
- void setNamespaceAware(boolean value)

gets or sets the `namespaceAware` property of the factory. If set to `true`, the parsers that this factory generates are namespace aware.
- boolean isValidating()
- void setValidating(boolean value)

gets or sets the `validating` property of the factory. If set to `true`, the parsers that this factory generates validate their input.



javax.xml.parsers.SAXParser 1.4

- void parse(File f, DefaultHandler handler)
 - void parse(String url, DefaultHandler handler)
 - void parse(InputStream in, DefaultHandler handler)
- parses an XML document from the given file, URL, or input stream and reports parse events to the given handler.

API**org.xml.sax.ContentHandler 1.4**

- void startDocument()
- void endDocument()

is called at the start or the end of the document.

- void startElement(String uri, String lname, String qname, Attributes attr)
- void endElement(String uri, String lname, String qname)

is called at the start or the end of an element.

Parameters:	uri	The URI of the namespace (if the parser is namespace aware)
	lname	The local name without alias prefix (if the parser is namespace aware)
	qname	The element name if the parser is not namespace aware, or the qualified name with alias prefix if the parser reports qualified names in addition to local names

- void characters(char[] data, int start, int length)

is called when the parser reports character data.

Parameters:	data	An array of character data
	start	The index of the first character in the data array that is a part of the reported characters
	length	The length of the reported character string

API**org.xml.sax.Attributes 1.4**

- int getLength()

returns the number of attributes stored in this attribute collection.

- String getLocalName(int index)

returns the local name (without alias prefix) of the attribute with the given index, or the empty string if the parser is not namespace aware.

- String getURI(int index)

returns the namespace URI of the attribute with the given index, or the empty string if the node is not part of a namespace or if the parser is not namespace aware.

- String getQName(int index)

returns the qualified name (with alias prefix) of the attribute with the given index, or the empty string if the qualified name is not reported by the parser.

- String getValue(int index)

- String getValue(String qname)

- `String getValue(String uri, String lname)`
returns the attribute value from a given index, qualified name, or namespace URI + local name. Returns `null` if the value doesn't exist.

Using the StAX Parser

The StAX parser is a "pull parser." Instead of installing an event handler, you simply iterate through the events, using this basic loop:

```
InputStream in = url.openStream();
XMLInputFactory factory = XMLInputFactory.newInstance();
XMLStreamReader parser = factory.createXMLStreamReader(in);
while (parser.hasNext())
{
    int event = parser.next();
    Call parser methods to obtain event details
}
```

For example, when parsing the fragment

```
<font>
    <name>Helvetica</name>
    <size units="pt">36</size>
</font>
```

the parser yields the following events:

1. START_ELEMENT, element name: font
2. CHARACTERS, content: white space
3. START_ELEMENT, element name: name
4. CHARACTERS, content: Helvetica
5. END_ELEMENT, element name: name
6. CHARACTERS, content: white space
7. START_ELEMENT, element name: size
8. CHARACTERS, content: 36
9. END_ELEMENT, element name: size
10. CHARACTERS, content: white space
11. END_ELEMENT, element name: font

To analyze the attribute values, call the appropriate methods of the `XMLStreamReader` class. For example,

```
String units = parser.getAttributeValue(null, "units");
```

gets the `units` attribute of the current element.

By default, namespace processing is enabled. You can deactivate it by modifying the factory:

```
XMLInputFactory factory = XMLInputFactory.newInstance();
factory.setProperty(XMLInputFactory.IS_NAMESPACE_AWARE, false);
```

[Listing 2-9](#) contains the code for the web crawler program, implemented with the StAX parser. As you can see, the code is simpler than the equivalent SAX code because you don't have to worry about event handling.

Listing 2-9. StAXTest.java

Code View:

```

1. import java.io.*;
2. import java.net.*;
3. import javax.xml.stream.*;
4.
5. /**
6.  * This program demonstrates how to use a StAX parser. The program prints all hyperlinks links
7.  * of an XHTML web page. <br>
8.  * Usage: java StAXTest url
9.  * @author Cay Horstmann
10. * @version 1.0 2007-06-23
11. */
12. public class StAXTest
13. {
14.     public static void main(String[] args) throws Exception
15.     {
16.         String urlString;
17.         if (args.length == 0)
18.         {
19.             urlString = "http://www.w3c.org";
20.             System.out.println("Using " + urlString);
21.         }
22.         else urlString = args[0];
23.         URL url = new URL(urlString);
24.         InputStream in = url.openStream();
25.         XMLInputFactory factory = XMLInputFactory.newInstance();
26.         XMLStreamReader parser = factory.createXMLStreamReader(in);
27.         while (parser.hasNext())
28.         {
29.             int event = parser.next();
30.             if (event == XMLStreamConstants.START_ELEMENT)
31.             {
32.                 if (parser.getLocalName().equals("a"))
33.                 {
34.                     String href = parser.getAttributeValue(null, "href");
35.                     if (href != null)
36.                         System.out.println(href);
37.                 }
38.             }
39.         }
40.     }
41. }
```

API**javax.xml.stream.XMLInputFactory 6**

- static XMLInputFactory newInstance()

returns an instance of the `XMLInputFactory` class.

- void setProperty(String name, Object value)

sets a property for this factory, or throws an `IllegalArgumentException` if the property is not supported or cannot be set to the given value. The Java SE implementation supports the following `Boolean` valued properties:

`"javax.xml.stream.isValidating"`

When false (the default), the document is not validated. Not required by the specification.

`"javax.xml.stream.isNamespaceAware"`

When true (the default), namespaces are processed. Not required by the specification.

"javax.xml.stream.isCoalescing"	When false (the default), adjacent character data are not coalesced.
"javax.xml.stream.isReplacingEntityReferences"	When true (the default), entity references are replaced and reported as character data.
"javax.xml.stream.isSupportingExternalEntities"	When true (the default), external entities are resolved. The specification gives no default for this property.
"javax.xml.stream.supportDTD"	When true (the default), DTDs are reported as events.

- XMLStreamReader createXMLStreamReader(InputStream in)
- XMLStreamReader createXMLStreamReader(InputStream in, String characterEncoding)
- XMLStreamReader createXMLStreamReader(Reader in)
- XMLStreamReader createXMLStreamReader(Source in)

creates a parser that reads from the given stream, reader, or JAXP source.

API

javax.xml.stream.XMLStreamReader 6

- boolean hasNext()

returns true if there is another parse event.

- int next()

sets the parser state to the next parse event and returns one of the following constants: START_ELEMENT, END_ELEMENT, CHARACTERS, START_DOCUMENT, END_DOCUMENT, CDATA, COMMENT, SPACE (ignorable whitespace), PROCESSING_INSTRUCTION, ENTITY_REFERENCE, DTD.

- boolean isStartElement()
- boolean isEndElement()
- boolean isCharacters()
- boolean isWhiteSpace()

returns true if the current event is a start element, end element, character data, or whitespace.

- QName getName()
- String getLocalName()

gets the name of the element in a START_ELEMENT or END_ELEMENT event.

- `String getText()`
returns the characters of a CHARACTERS, COMMENT, or CDATA event, the replacement value for an ENTITY_REFERENCE, or the internal subset of a DTD.
- `int getAttributeCount()`
- `QName getAttributeName(int index)`
- `String getAttributeLocalName(int index)`
- `String getAttributeValue(int index)`
gets the attribute count and the names and values of the attributes, provided the current event is START_ELEMENT.
- `String getAttributeValue(String namespaceURI, String name)`
gets the value of the attribute with the given name, provided the current event is START_ELEMENT. If namespaceURI is null, the namespace is not checked.



Generating XML Documents

You now know how to write Java programs that read XML. Let us now turn to the opposite process, producing XML output. Of course, you could write an XML file simply by making a sequence of `print` calls, printing the elements, attributes, and text content, but that would not be a good idea. The code is rather tedious, and you can easily make mistakes if you don't pay attention to special symbols (such as " or <) in the attribute values and text content.

A better approach is to build up a DOM tree with the contents of the document and then write out the tree contents. To build a DOM tree, you start out with an empty document. You can get an empty document by calling the `newDocument` method of the `DocumentBuilder` class.

```
Document doc = builder.newDocument();
```

Use the `createElement` method of the `Document` class to construct the elements of your document.

```
Element rootElement = doc.createElement(rootName);
Element childElement = doc.createElement(childName);
```

Use the `createTextNode` method to construct text nodes:

```
Text textNode = doc.createTextNode(textContents);
```

Add the root element to the document, and add the child nodes to their parents:

```
doc.appendChild(rootElement);
rootElement.appendChild(childElement);
childElement.appendChild(textNode);
```

As you build up the DOM tree, you may also need to set element attributes. Simply call the `setAttribute` method of the `Element` class:

```
rootElement.setAttribute(name, value);
```

Somewhat curiously, the DOM API currently has no support for writing a DOM tree to an output stream. To overcome this limitation, we use the Extensible Stylesheet Language Transformations (XSLT) API. For more information about XSLT, turn to the section "[XSL Transformations](#)" on page [157](#). Right now, consider the code that follows a "magic incantation" to produce XML output.

We apply the "do nothing" transformation to the document and capture its output. To include a DOCTYPE node in the output, you also need to set the SYSTEM and PUBLIC identifiers as output properties.

Code View:

```
// construct the "do nothing" transformation
Transformer t = TransformerFactory.newInstance().newTransformer();
// set output properties to get a DOCTYPE node
t.setOutputProperty(OutputKeys.DOCUMENT_TYPE_DECLARATION, systemIdentifier);
t.setOutputProperty(OutputKeys.DOCUMENT_PUBLIC_IDENTIFIER, publicIdentifier);
// set indentation
t.setOutputProperty(OutputKeys.INDENT, "yes");
t.setOutputProperty(OutputKeys.METHOD, "xml");
t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
// apply the "do nothing" transformation and send the output to a file
t.transform(new DOMSource(doc), new StreamResult(new FileOutputStream(file)));
```

Note



The resulting XML file contains no whitespace (that is, no line breaks or indentations). If you like whitespace, set the "OutputKeys.INDENT" property to the string "yes".

[Listing 2-10](#) on page [150](#) is a typical program that produces XML output. The program draws a modernist painting—a random set of colored rectangles (see [Figure 2-6](#)). To save a masterpiece, we use the Scalable Vector Graphics (SVG) format. SVG is an XML format to describe complex graphics in a device-independent fashion. You can find more information about SVG at <http://www.w3c.org/Graphics/SVG>. To view SVG files, download the Apache Batik viewer ([Figure 2-7](#)) from <http://xmlgraphics.apache.org/batik>.

Figure 2-6. Generating modern art

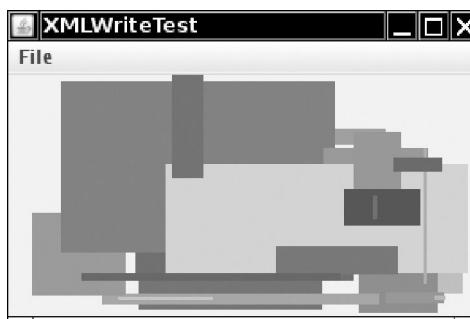
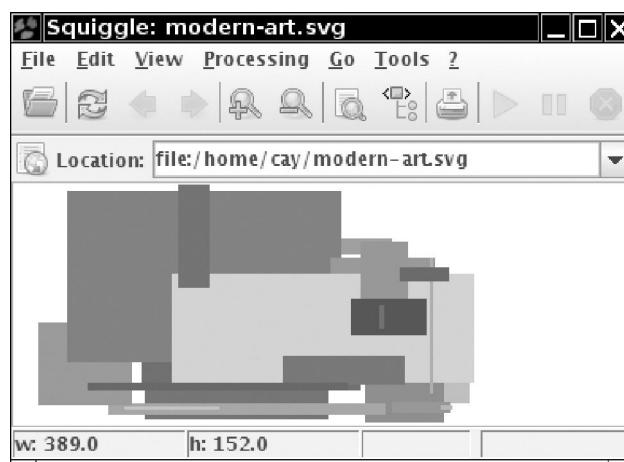


Figure 2-7. The Apache Batik SVG viewer



We don't go into details about SVG. If you are interested in SVG, we suggest you start with the tutorial on the Adobe site. For our purposes, we just need to know how to express a set of colored rectangles. Here is a sample:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 20000802//EN"
  "http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd">
<svg width="300" height="150">
<rect x="231" y="61" width="9" height="12" fill="#6e4a13"/>
<rect x="107" y="106" width="56" height="5" fill="#c406be"/>
. . .
</svg>
```

As you can see, each rectangle is described as a `rect` node. The position, width, height, and fill color are attributes. The fill color is an RGB value in hexadecimal.

Note



SVG uses attributes heavily. In fact, some attributes are quite complex. For example, here is a path element:

```
<path d="M 100 100 L 300 100 L 200 300 z">
```

The `M` denotes a "moveto" command, `L` is "lineto," and `z` is "closepath" (!). Apparently, the designers of this data format didn't have much confidence in using XML for structured data. In your own XML formats, you might want to use elements instead of complex attributes.



javax.xml.parsers.DocumentBuilder 1.4

- `Document newDocument()`

returns an empty document.



org.w3c.dom.Document 1.4

- `Element createElement(String name)`

returns an element with the given name.

- `Text createTextNode(String data)`

returns a text node with the given data.



org.w3c.dom.Node 1.4

- `Node appendChild(Node child)`

appends a node to the list of children of this node. Returns the appended node.



org.w3c.dom.Element 1.4

- `void setAttribute(String name, String value)`

sets the attribute with the given name to the given value.

- `void setAttributeNS(String uri, String qname, String value)`

sets the attribute with the given namespace URI and qualified name to the given value.

Parameters:	uri	The URI of the namespace, or <code>null</code>
	qname	The qualified name. If it has an alias prefix, then <code>uri</code> must not be <code>null</code>
	value	The attribute value

API

javax.xml.transform.TransformerFactory 1.4

- static `TransformerFactory newInstance()`
returns an instance of the `TransformerFactory` class.
- `Transformer newTransformer()`
returns an instance of the `Transformer` class that carries out an identity or "do nothing" transformation.

API

javax.xml.transform.Transformer 1.4

- `void setOutputProperty(String name, String value)`
sets an output property. See <http://www.w3.org/TR/xslt#output> for a listing of the standard output properties. The most useful ones are shown here:

<code>doctype-public</code>	The public ID to be used in the DOCTYPE declaration
<code>doctype-system</code>	The system ID to be used in the DOCTYPE declaration
<code>indent</code>	"yes" or "no"
<code>method</code>	"xml", "html", "text", or a custom string
- `void transform(Source from, Result to)`
transforms an XML document.

API

javax.xml.transform.dom.DOMSource 1.4

- `DOMSource(Node n)`
constructs a source from the given node. Usually, `n` is a document node.

API

javax.xml.transform.stream.StreamResult 1.4

- `StreamResult(File f)`
- `StreamResult(OutputStream out)`
- `StreamResult(Writer out)`
- `StreamResult(String systemID)`
constructs a stream result from a file, stream, writer, or system ID (usually a relative

or absolute URL).

Writing an XML Document with StAX

In the preceding section, you saw how to produce an XML document by writing a DOM tree. If you have no other use for the DOM tree, that approach is not very efficient.

The StAX API lets you write an XML tree directly. Construct an `XMLStreamWriter` from an `OutputStream`, like this:

```
XMLOutputFactory factory = XMLOutputFactory.newInstance();
XMLStreamWriter writer = factory.createXMLStreamWriter(out);
```

To produce the XML header, call

```
writer.writeStartDocument()
```

Then call

```
writer.writeStartElement(name);
```

Add attributes by calling

```
writer.writeAttribute(name, value);
```

Now you can add child elements by calling `writeStartElement` again, or write characters with

```
writer.writeCharacters(text);
```

When you have written all child nodes, call

```
writer.writeEndElement();
```

This causes the current element to be closed.

To write an element without children (such as ``), you use the call

```
writer.writeEmptyElement(name);
```

Finally, at the end of the document, call

```
writer.writeEndDocument();
```

This call closes any open elements.

As with the DOM/XSLT approach, you don't have to worry about escaping characters in attribute values and character data. However, it is possible to produce malformed XML, such as a document with multiple root nodes. Also, the current version of StAX has no support for producing indented output.

The program in [Listing 2-10](#) shows you both approaches for writing XML.

Listing 2-10. `XMLWriteTest.java`

Code View:

```
1. import java.awt.*;
2. import java.awt.geom.*;
3. import java.io.*;
4. import java.util.*;
5. import java.awt.event.*;
6. import javax.swing.*;
```

```
7. import javax.xml.parsers.*;
8. import javax.xml.stream.*;
9. import javax.xml.transform.*;
10. import javax.xml.transform.dom.*;
11. import javax.xml.transform.stream.*;
12. import org.w3c.dom.*;
13.
14. /**
15. * This program shows how to write an XML file. It saves a file describing a modern drawing
16. * in SVG format.
17. * @version 1.10 2004-09-04
18. * @author Cay Horstmann
19. */
20. public class XMLWriteTest
21. {
22.     public static void main(String[] args)
23.     {
24.         EventQueue.invokeLater(new Runnable()
25.         {
26.             public void run()
27.             {
28.                 XMLWriteFrame frame = new XMLWriteFrame();
29.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
30.                 frame.setVisible(true);
31.             }
32.         });
33.     }
34. }
35.
36. /**
37. * A frame with a component for showing a modern drawing.
38. */
39. class XMLWriteFrame extends JFrame
40. {
41.     public XMLWriteFrame()
42.     {
43.         setTitle("XMLWriteTest");
44.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
45.
46.         chooser = new JFileChooser();
47.
48.         // add component to frame
49.
50.         comp = new RectangleComponent();
51.         add(comp);
52.
53.         // set up menu bar
54.
55.         JMenuBar menuBar = new JMenuBar();
56.         setJMenuBar(menuBar);
57.
58.         JMenu menu = new JMenu("File");
59.         menuBar.add(menu);
60.
61.         JMenuItem newItem = new JMenuItem("New");
62.         menu.add(newItem);
63.         newItem.addActionListener(new ActionListener()
64.         {
65.             public void actionPerformed(ActionEvent event)
66.             {
67.                 comp.newDrawing();
68.             }
69.         });
70.
71.         JMenuItem saveItem = new JMenuItem("Save with DOM/XSLT");
72.         menu.add(saveItem);
73.         saveItem.addActionListener(new ActionListener()
74.         {
75.             public void actionPerformed(ActionEvent event)
76.             {
77.                 try
78.                 {
79.                     saveDocument();
80.                 }
81.                 catch (Exception e)
82.                 {
83.                     JOptionPane.showMessageDialog(XMLWriteFrame.this, e.toString());
84.                 }
85.             }
86.         });
87.
```

```
88.     JMenuItem saveStAXItem = new JMenuItem("Save with StAX");
89.     menu.add(saveStAXItem);
90.     saveStAXItem.addActionListener(new ActionListener()
91.     {
92.         public void actionPerformed(ActionEvent event)
93.         {
94.             try
95.             {
96.                 saveStAX();
97.             }
98.             catch (Exception e)
99.             {
100.                 JOptionPane.showMessageDialog(XMLWriteFrame.this, e.toString());
101.             }
102.         }
103.     });
104.
105.     JMenuItem exitItem = new JMenuItem("Exit");
106.     menu.add(exitItem);
107.     exitItem.addActionListener(new ActionListener()
108.     {
109.         public void actionPerformed(ActionEvent event)
110.         {
111.             System.exit(0);
112.         }
113.     });
114. }
115.
116. /**
117. * Saves the drawing in SVG format, using DOM/XSLT
118. */
119. public void saveDocument() throws TransformerException, IOException
120. {
121.     if (chooser.showSaveDialog(this) != JFileChooser.APPROVE_OPTION) return;
122.     File f = chooser.getSelectedFile();
123.     Document doc = comp.buildDocument();
124.     Transformer t = TransformerFactory.newInstance().newTransformer();
125.     t.setProperty(OutputKeys.DOCUMENT_TYPE_SYSTEM,
126.         "http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd");
127.     t.setProperty(OutputKeys.DOCUMENT_PUBLIC "-//W3C//DTD SVG 20000802//EN");
128.     t.setProperty(OutputKeys.INDENT, "yes");
129.     t.setProperty(OutputKeys.METHOD, "xml");
130.     t.setProperty("{http://xml.apache.org/xslt}indent-amount", "2");
131.     t.transform(new DOMSource(doc), new StreamResult(new FileOutputStream(f)));
132. }
133.
134. /**
135. * Saves the drawing in SVG format, using StAX
136. */
137. public void saveStAX() throws FileNotFoundException, XMLStreamException
138. {
139.     if (chooser.showSaveDialog(this) != JFileChooser.APPROVE_OPTION) return;
140.     File f = chooser.getSelectedFile();
141.     XMLOutputFactory factory = XMLOutputFactory.newInstance();
142.     XMLStreamWriter writer = factory.createXMLStreamWriter(new FileOutputStream(f));
143.     comp.writeDocument(writer);
144.     writer.close();
145. }
146.
147. public static final int DEFAULT_WIDTH = 300;
148. public static final int DEFAULT_HEIGHT = 200;
149.
150. private RectangleComponent comp;
151. private JFileChooser chooser;
152. }
153.
154. /**
155. * A component that shows a set of colored rectangles
156. */
157. class RectangleComponent extends JComponent
158. {
159.     public RectangleComponent()
160.     {
161.         rects = new ArrayList<Rectangle2D>();
162.         colors = new ArrayList<Color>();
163.         generator = new Random();
164.
165.         DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
166.         try
167.         {
168.             builder = factory.newDocumentBuilder();
```

```
169.     }
170.     catch (ParserConfigurationException e)
171.     {
172.         e.printStackTrace();
173.     }
174. }
175.
176. /**
177. * Create a new random drawing.
178. */
179. public void newDrawing()
180. {
181.     int n = 10 + generator.nextInt(20);
182.     rects.clear();
183.     colors.clear();
184.     for (int i = 1; i <= n; i++)
185.     {
186.         int x = generator.nextInt(getWidth());
187.         int y = generator.nextInt(getHeight());
188.         int width = generator.nextInt(getWidth() - x);
189.         int height = generator.nextInt(getHeight() - y);
190.         rects.add(new Rectangle(x, y, width, height));
191.         int r = generator.nextInt(256);
192.         int g = generator.nextInt(256);
193.         int b = generator.nextInt(256);
194.         colors.add(new Color(r, g, b));
195.     }
196.     repaint();
197. }
198.
199. public void paintComponent(Graphics g)
200. {
201.     if (rects.size() == 0) newDrawing();
202.     Graphics2D g2 = (Graphics2D) g;
203.     // draw all rectangles
204.     for (int i = 0; i < rects.size(); i++)
205.     {
206.         g2.setPaint(colors.get(i));
207.         g2.fill(rects.get(i));
208.     }
209. }
210.
211. /**
212. * Creates an SVG document of the current drawing.
213. * @return the DOM tree of the SVG document
214. */
215. public Document buildDocument()
216. {
217.     Document doc = builder.newDocument();
218.     Element svgElement = doc.createElement("svg");
219.     doc.appendChild(svgElement);
220.     svgElement.setAttribute("width", "" + getWidth());
221.     svgElement.setAttribute("height", "" + getHeight());
222.     for (int i = 0; i < rects.size(); i++)
223.     {
224.         Color c = colors.get(i);
225.         Rectangle2D r = rects.get(i);
226.         Element rectElement = doc.createElement("rect");
227.         rectElement.setAttribute("x", "" + r.getX());
228.         rectElement.setAttribute("y", "" + r.getY());
229.         rectElement.setAttribute("width", "" + r.getWidth());
230.         rectElement.setAttribute("height", "" + r.getHeight());
231.         rectElement.setAttribute("fill", colorToString(c));
232.         svgElement.appendChild(rectElement);
233.     }
234.     return doc;
235. }
236.
237. /**
238. * Writes an SVG document of the current drawing.
239. * @param writer the document destination
240. */
241. public void writeDocument(XMLStreamWriter writer) throws XMLStreamException
242. {
243.     writer.writeStartDocument();
244.     writer.writeDTD("<!DOCTYPE svg PUBLIC '-//W3C//DTD SVG 20000802//EN'" +
245.                     + "\n\"http://www.w3.org/TR/2000/CR-SVG-20000802/DTD/svg-20000802.dtd\">");
246.     writer.writeStartElement("svg");
247.     writer.writeAttribute("width", "" + getWidth());
248.     writer.writeAttribute("height", "" + getHeight());
249.     for (int i = 0; i < rects.size(); i++)
```

```
250.     {
251.         Color c = colors.get(i);
252.         Rectangle2D r = rects.get(i);
253.         writer.writeEmptyElement("rect");
254.         writer.writeAttribute("x", "" + r.getX());
255.         writer.writeAttribute("y", "" + r.getY());
256.         writer.writeAttribute("width", "" + r.getWidth());
257.         writer.writeAttribute("height", "" + r.getHeight());
258.         writer.writeAttribute("fill", colorToString(c));
259.     }
260.     writer.writeEndDocument(); // closes svg element
261. }
262.
263. /**
264. * Converts a color to a hex value.
265. * @param c a color
266. * @return a string of the form #rrggbb
267. */
268. private static String colorToString(Color c)
269. {
270.     StringBuffer buffer = new StringBuffer();
271.     buffer.append(Integer.toHexString(c.getRGB() & 0xFFFFFFFF));
272.     while (buffer.length() < 6)
273.         buffer.insert(0, '0');
274.     buffer.insert(0, '#');
275.     return buffer.toString();
276. }
277.
278. private ArrayList<Rectangle2D> rects;
279. private ArrayList<Color> colors;
280. private Random generator;
281. private DocumentBuilder builder;
282. }
```



javax.xml.stream.XMLOutputFactory 6

- static XMLOutputFactory newInstance()
returns an instance of the XMLOutputFactory class.
- XMLStreamWriter createXMLStreamWriter(OutputStream in)
- XMLStreamWriter createXMLStreamWriter(OutputStream in, String characterEncoding)
- XMLStreamWriter createXMLStreamWriter(Writer in)
- XMLStreamWriter createXMLStreamWriter(Result in)

creates a writer that writes to the given stream, writer, or JAXP result.



javax.xml.stream.XMLStreamWriter 6

- void writeStartDocument()
- void writeStartDocument(String xmlVersion)
- void writeStartDocument(String encoding, String xmlVersion)

writes the XML processing instruction at the top of the document. Note that the encoding parameter is only used to write the attribute. It does not set the character encoding of the output.

- `void setDefaultNamespace(String namespaceURI)`
- `void setPrefix(String prefix, String namespaceURI)`

sets the default namespace or the namespace associated with a prefix. The declaration is scoped to the current element, or, if no element has been written, to the document root.
- `void writeStartElement(String localName)`
- `void writeStartElement(String namespaceURI, String localName)`

writes a start tag, replacing the `namespaceURI` with the associated prefix.
- `void writeEndElement()`

closes the current element.
- `void writeEndDocument()`

closes all open elements.
- `void writeEmptyElement(String localName)`
- `void writeEmptyElement(String namespaceURI, String localName)`

writes a self-closing tag, replacing the `namespaceURI` with the associated prefix.
- `void writeAttribute(String localName, String value)`
- `void writeAttribute(String namespaceURI, String localName, String value)`

writes an attribute for the current element, replacing the `namespaceURI` with the associated prefix.
- `void writeCharacters(String text)`

writes character data.
- `void writeCDATA(String text)`

writes a CDATA block.
- `void writeDTD(String dtd)`

writes the `dtd` string, which is assumed to contain a DOCTYPE declaration.
- `void writeComment(String comment)`

writes a comment.
- `void close()`

closes this writer.

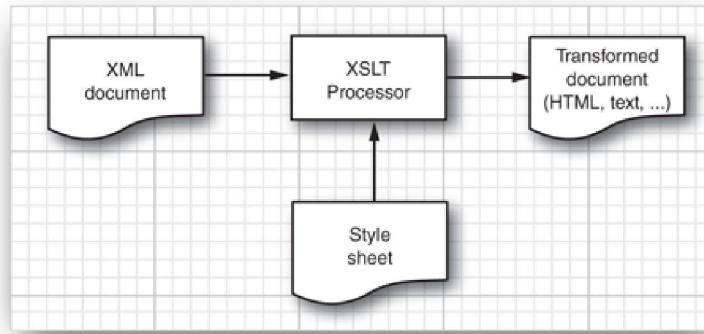


XSL Transformations

The XSL Transformations (XSLT) mechanism allows you to specify rules for transforming XML documents into other formats, such as plain text, XHTML, or any other XML format. XSLT is commonly used to translate from one machine-readable XML format to another, or to translate XML into a presentation format for human consumption.

You need to provide an XSLT style sheet that describes the conversion of XML documents into some other format. An XSLT processor reads an XML document and the style sheet, and it produces the desired output (see [Figure 2-8](#)).

Figure 2-8. Applying XSL transformations



Here is a typical example. We want to transform XML files with employee records into HTML documents. Consider this input file:

```

<staff>
  <employee>
    <name>Carl Cracker</name>
    <salary>75000</salary>
    <hiredate year="1987" month="12" day="15"/>
  </employee>
  <employee>
    <name>Harry Hacker</name>
    <salary>50000</salary>
    <hiredate year="1989" month="10" day="1"/>
  </employee>
  <employee>
    <name>Tony Tester</name>
    <salary>40000</salary>
    <hiredate year="1990" month="3" day="15"/>
  </employee>
</staff>
  
```

The desired output is an HTML table:

```

<table border="1">
<tr>
<td>Carl Cracker</td><td>$75000.0</td><td>1987-12-15</td>
</tr>
<tr>
<td>Harry Hacker</td><td>$50000.0</td><td>1989-10-1</td>
</tr>
<tr>
<td>Tony Tester</td><td>$40000.0</td><td>1990-3-15</td>
</tr>
</table>
  
```

The XSLT specification is quite complex, and entire books have been written on the subject. We can't possibly discuss all the features of XSLT, so we just work through a representative example. You can find more information in the book *Essential XML* by Don Box et al. (Addison-Wesley Professional 2000). The XSLT specification is available at <http://www.w3.org/TR/xslt>.

A style sheet with transformation templates has this form:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
  <xsl:output method="html"/>
  template1
  template2
  ...
</xsl:stylesheet>
  
```

In our example, the `xsl:output` element specifies the method as HTML. Other valid method settings are `xml` and `text`.

Here is a typical template:

```
<xsl:template match="/staff/employee">
  <tr><xsl:apply-templates/></tr>
</xsl:template>
```

The value of the `match` attribute is an XPath expression. The template states: Whenever you see a node in the XPath set `/staff/employee`, do the following:

1. Emit the string `<tr>`.
2. Keep applying templates as you process its children.
3. Emit the string `</tr>` after you are done with all children.

In other words, this template generates the HTML table row markers around every employee record.

The XSLT processor starts processing by examining the root element. Whenever a node matches one of the templates, it applies the template. (If multiple templates match, the best matching one is used—see the specification at <http://www.w3.org/TR/xslt> for the gory details.) If no template matches, the processor carries out a default action. For text nodes, the default is to include the contents in the output. For elements, the default action is to create no output but to keep processing the children.

Here is a template for transforming `name` nodes in an employee file:

```
<xsl:template match="/staff/employee/name">
  <td><xsl:apply-templates/></td>
</xsl:template>
```

As you can see, the template produces the `<td>...</td>` delimiters, and it asks the processor to recursively visit the children of the `name` element. There is just one child, the text node. When the processor visits that node, it emits the text contents (provided, of course, that there is no other matching template).

You have to work a little harder if you want to copy attribute values into the output. Here is an example:

```
<xsl:template match="/staff/employee/hiredate">
  <td><xsl:value-of select="@year"/>-<xsl:value-of
  select="@month"/>-<xsl:value-of select="@day"/></td>
</xsl:template>
```

When processing a `hiredate` node, this template emits

- The string `<td>`
- The value of the `year` attribute
- A hyphen
- The value of the `month` attribute
- A hyphen
- The value of the `day` attribute
- A hyphen
- The string `</td>`

The `xsl:value-of` statement computes the string value of a node set. The node set is specified by the XPath value of the `select` attribute. In this case, the path is relative to the currently processed node. The node set is converted to a string by concatenation of the string values of all nodes. The string value of an attribute node is its value. The string value of a text node is its contents. The string value of an element node is the concatenation of the string values of its child nodes (but not its

attributes).

[Listing 2-11](#) contains the style sheet for turning an XML file with employee records into an HTML table.

[Listing 2-12](#) shows a different set of transformations. The input is the same XML file, and the output is plain text in the familiar property file format:

```
employee.1.name=Carl Cracker
employee.1.salary=75000.0
employee.1.hiredate=1987-12-15
employee.2.name=Harry Hacker
employee.2.salary=50000.0
employee.2.hiredate=1989-10-1
employee.3.name=Tony Tester
employee.3.salary=40000.0
employee.3.hiredate=1990-3-15
```

That example uses the `position()` function, which yields the position of the current node as seen from its parent. We get an entirely different output simply by switching the style sheet. Thus, you can safely use XML to describe your data, even if some applications need the data in another format. Just use XSLT to generate the alternative format.

It is extremely simple to generate XSL transformations in the Java platform. Set up a transformer factory for each style sheet. Then get a transformer object, and tell it to transform a source to a result.

Code View:

```
File styleSheet = new File(filename);
StreamSource styleSource = new StreamSource(styleSheet);
Transformer t = TransformerFactory.newInstance().newTransformer(styleSource);
t.transform(source, result);
```

The parameters of the `transform` method are objects of classes that implement the `Source` and `Result` interfaces. There are three implementations of the `Source` interface:

```
DOMSource
SAXSource
StreamSource
```

You can construct a `StreamSource` from a file, stream, reader, or URL, and a `DOMSource` from the node of a DOM tree. For example, in the preceding section, we invoked the identity transformation as

```
t.transform(new DOMSource(doc), result);
```

In our example program, we do something slightly more interesting. Rather than starting out with an existing XML file, we produce a SAX XML reader that gives the illusion of parsing an XML file by emitting appropriate SAX events. Actually, our XML reader reads a flat file, as described in [Chapter 1](#). The input file looks like this:

```
Carl Cracker|75000.0|1987|12|15
Harry Hacker|50000.0|1989|10|1
Tony Tester|40000.0|1990|3|15
```

Our XML reader generates SAX events as it processes the input. Here is a part of the `parse` method of the `EmployeeReader` class that implements the `XMLReader` interface.

```
AttributesImpl attributes = new AttributesImpl();
handler.startDocument();
handler.startElement("", "staff", "staff", attributes);
while ((line = in.readLine()) != null)
{
    handler.startElement("", "employee", "employee", attributes);
    StringTokenizer t = new StringTokenizer(line, "|");
    handler.startElement("", "name", "name", attributes);
    String s = t.nextToken();
    handler.characters(s.toCharArray(), 0, s.length());
    handler.endElement("", "name", "name");
    .
    .
    .
    handler.endElement("", "employee", "employee");
```

```
}  
handler.endElement("", rootElement, rootElement);  
handler.endDocument();
```

The SAXSource for the transformer is constructed from the XML reader:

```
t.transform(new SAXSource(new EmployeeReader(),  
    new InputSource(new FileInputStream(filename))), result);
```

This is an ingenious trick to convert non-XML legacy data into XML. Of course, most XSLT applications will already have XML input data, and you can simply invoke the `transform` method on a StreamSource, like this:

```
t.transform(new StreamSource(file), result);
```

The transformation result is an object of a class that implements the `Result` interface. The Java library supplies three classes:

```
DOMResult  
SAXResult  
StreamResult
```

To store the result in a DOM tree, use a `DocumentBuilder` to generate a new document node and wrap it into a `DOMResult`:

```
Document doc = builder.newDocument();  
t.transform(source, new DOMResult(doc));
```

To save the output in a file, use a `StreamResult`:

```
t.transform(source, new StreamResult(file));
```

[Listing 2-13](#) contains the complete source code.

Listing 2-11. `makehtml.xsl`

Code View:

```
1. <?xml version="1.0" encoding="ISO-8859-1"?>  
2.  
3. <xsl:stylesheet  
4.     xmlns:xsl="http://www.w3.org/1999/XSL/Transform"  
5.     version="1.0">  
6.  
7.     <xsl:output method="html"/>  
8.  
9.     <xsl:template match="/staff">  
10.        <table border="1"><xsl:apply-templates/></table>  
11.    </xsl:template>  
12.  
13.    <xsl:template match="/staff/employee">  
14.        <tr><xsl:apply-templates/></tr>  
15.    </xsl:template>  
16.  
17.    <xsl:template match="/staff/employee/name">  
18.        <td><xsl:apply-templates/></td>  
19.    </xsl:template>  
20.  
21.    <xsl:template match="/staff/employee/salary">  
22.        <td>$<xsl:apply-templates/></td>  
23.    </xsl:template>  
24.  
25.    <xsl:template match="/staff/employee/hiredate">  
26.        <td><xsl:value-of select="@year" />-<xsl:value-of  
27.            select="@month" />-<xsl:value-of select="@day" /></td>  
28.    </xsl:template>  
29.  
30. </xsl:stylesheet>
```

Listing 2-12. makeprop.xsl

Code View:

```
1. <?xml version="1.0"?>
2.
3. <xsl:stylesheet
4.   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
5.   version="1.0">
6.
7.   <xsl:output method="text"/>
8.   <xsl:template match="/staff/employee">
9.     employee.<xsl:value-of select="position()" />.name=<xsl:value-of select="name/text()"/>
10.    employee.<xsl:value-of select="position()" />.salary=<xsl:value-of select="salary/text()"/>
11.    employee.<xsl:value-of select="position()" />.hiredate=<xsl:value-of select="hiredate/@year"/>
12.      -<xsl:value-of select="hiredate/@month"/>-<xsl:value-of select="hiredate/@day"/>
13.   </xsl:template>
14.
15. </xsl:stylesheet>
```

Listing 2-13. TransformTest.java

Code View:

```
1. import java.io.*;
2. import java.util.*;
3. import javax.xml.transform.*;
4. import javax.xml.transform.sax.*;
5. import javax.xml.transform.stream.*;
6. import org.xml.sax.*;
7. import org.xml.sax.helpers.*;
8.
9. /**
10.  * This program demonstrates XSL transformations. It applies a transformation to a set
11.  * of employee records. The records are stored in the file employee.dat and turned into XML
12.  * format. Specify the stylesheet on the command line, e.g. java TransformTest makeprop.xsl
13.  * @version 1.01 2007-06-25
14.  * @author Cay Horstmann
15. */
16. public class TransformTest
17. {
18.   public static void main(String[] args) throws Exception
19.   {
20.     String filename;
21.     if (args.length > 0) filename = args[0];
22.     else filename = "makehtml.xsl";
23.     File styleSheet = new File(filename);
24.     StreamSource styleSource = new StreamSource(styleSheet);
25.
26.     Transformer t = TransformerFactory.newInstance().newTransformer(styleSource);
27.     t.setOutputProperty(OutputKeys.INDENT, "yes");
28.     t.setOutputProperty(OutputKeys.METHOD, "xml");
29.     t.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "2");
30.
31.     t.transform(new SAXSource(new EmployeeReader(), new InputSource(new FileInputStream(
32.           "employee.dat"))), new StreamResult(System.out));
33.   }
34. }
35. /**
36.  * This class reads the flat file employee.dat and reports SAX parser events to act as if
37.  * it was parsing an XML file.
38. */
39. class EmployeeReader implements XMLReader
40. {
41.   public void parse(InputSource source) throws IOException, SAXException
42.   {
43.     InputStream stream = source.getByteStream();
44.     BufferedReader in = new BufferedReader(new InputStreamReader(stream));
45.     String rootElement = "staff";
```

```
46.     AttributesImpl atts = new AttributesImpl();
47.
48.     if (handler == null) throw new SAXException("No content handler");
49.
50.     handler.startDocument();
51.     handler.startElement("", rootElement, rootElement, atts);
52.     String line;
53.     while ((line = in.readLine()) != null)
54.     {
55.         handler.startElement("", "employee", "employee", atts);
56.         StringTokenizer t = new StringTokenizer(line, "|");
57.
58.         handler.startElement("", "name", "name", atts);
59.         String s = t.nextToken();
60.         handler.characters(s.toCharArray(), 0, s.length());
61.         handler.endElement("", "name", "name");
62.
63.         handler.startElement("", "salary", "salary", atts);
64.         s = t.nextToken();
65.         handler.characters(s.toCharArray(), 0, s.length());
66.         handler.endElement("", "salary", "salary");
67.
68.         atts.addAttribute("", "year", "year", "CDATA", t.nextToken());
69.         atts.addAttribute("", "month", "month", "CDATA", t.nextToken());
70.         atts.addAttribute("", "day", "day", "CDATA", t.nextToken());
71.         handler.startElement("", "hiredate", "hiredate", atts);
72.         handler.endElement("", "hiredate", "hiredate");
73.         atts.clear();
74.
75.         handler.endElement("", "employee", "employee");
76.     }
77.
78.     handler.endElement("", rootElement, rootElement);
79.     handler.endDocument();
80. }
81.
82. public void setContentHandler(ContentHandler newValue)
83. {
84.     handler = newValue;
85. }
86.
87. public ContentHandler getContentHandler()
88. {
89.     return handler;
90. }
91.
92. // the following methods are just do-nothing implementations
93. public void parse(String systemId) throws IOException, SAXException
94. {
95. }
96.
97. public void setErrorHandler(ErrorHandler handler)
98. {
99. }
100.
101. public ErrorHandler getErrorHandler()
102. {
103.     return null;
104. }
105.
106. public void setDTDHandler(DTDHandler handler)
107. {
108. }
109.
110. public DTDHandler getDTDHandler()
111. {
112.     return null;
113. }
114.
115. public void setEntityResolver(EntityResolver resolver)
116. {
117. }
118.
119. public EntityResolver getEntityResolver()
120. {
121.     return null;
122. }
123.
124. public void setProperty(String name, Object value)
125. {
126. }
```

```
127.  
128.     public Object getProperty(String name)  
129.     {  
130.         return null;  
131.     }  
132.  
133.     public void setFeature(String name, boolean value)  
134.     {  
135.     }  
136.  
137.     public boolean getFeature(String name)  
138.     {  
139.         return false;  
140.     }  
141.  
142.     private ContentHandler handler;  
143. }
```



javax.xml.transform.TransformerFactory 1.4

- `Transformer newTransformer(Source styleSheet)`

returns an instance of the `Transformer` class that reads a style sheet from the given source.



javax.xml.transform.stream.StreamSource 1.4

- `StreamSource(File f)`
- `StreamSource(InputStream in)`
- `StreamSource(Reader in)`
- `StreamSource(String systemID)`

constructs a stream source from a file, stream, reader, or system ID (usually a relative or absolute URL).



javax.xml.transform.sax.SAXSource 1.4

- `SAXSource(XMLReader reader, InputSource source)`

constructs a SAX source that obtains data from the given input source and uses the given reader to parse the input.



org.xml.sax.XMLReader 1.4

- `void setContentHandler(ContentHandler handler)`

sets the handler that is notified of parse events as the input is parsed.

- `void parse(InputSource source)`

parses the input from the given input source and sends parse events to the content

handler.

API

javax.xml.transform.dom.DOMResult 1.4

- DOMResult(Node n)

constructs a source from the given node. Usually, n is a new document node.

API

org.xml.sax.helpers.AttributesImpl 1.4

- void addAttribute(String uri, String lname, String qname, String type, String value)

adds an attribute to this attribute collection.

Parameters:	uri	The URI of the namespace
	lname	The local name without alias prefix
	qname	The qualified name with alias prefix
	type	The type, one of "CDATA", "ID", "IDREF", "IDREFS", "NMTOKEN", "NMTOKENS", "ENTITY", "ENTITIES", or "NOTATION"
	value	The attribute value

- void clear()

removes all attributes from this attribute collection.

This example concludes our discussion of XML support in the Java library. You should now have a good perspective on the major strengths of XML, in particular, for automated parsing and validation and as a powerful transformation mechanism. Of course, all this technology is only going to work for you if you design your XML formats well. You need to make sure that the formats are rich enough to express all your business needs, that they are stable over time, and that your business partners are willing to accept your XML documents. Those issues can be far more challenging than dealing with parsers, DTDs, or transformations.

In the next chapter, we discuss network programming on the Java platform, starting with the basics of network sockets and moving on to higher level protocols for e-mail and the World Wide Web.



Chapter 3. Networking

- [CONNECTING TO A SERVER](#)
- [IMPLEMENTING SERVERS](#)
- [INTERRUPTIBLE SOCKETS](#)
- [SENDING E-MAIL](#)
- [MAKING URL CONNECTIONS](#)

We begin this chapter by reviewing basic networking concepts. We then move on to writing Java programs that connect to network services. We show you how network clients and servers are implemented. Finally, you will see how to send e-mail from a Java program and how to harvest information from a web server.

Connecting to a Server

Before writing our first network program, let's learn about a great debugging tool for network programming that you already have, namely, telnet. Telnet is preinstalled on most systems. You should be able to launch it by typing `telnet` from a command shell.

Note

 In Windows Vista, telnet is installed but deactivated by default. To activate it, go to the Control Panel, select Programs, click "Turn Windows Features On or Off", and select the "Telnet client" checkbox. The Windows firewall also blocks quite a few network ports that we use in this chapter; you might need an administrator account to unblock them.

You may have used telnet to connect to a remote computer, but you can use it to communicate with other services provided by Internet hosts as well. Here is an example of what you can do. Type

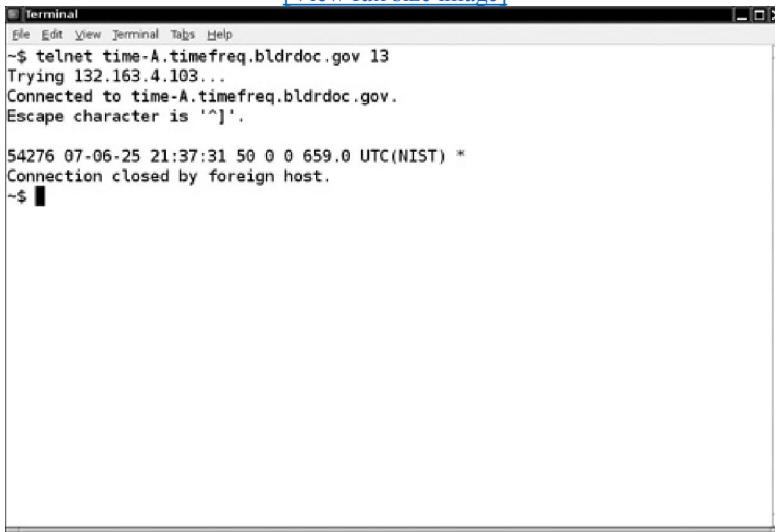
```
telnet time-A.timefreq.bldrdoc.gov 13
```

As [Figure 3-1](#) shows, you should get back a line like this:

```
54276 07-06-25 21:37:31 50 0 0 659.0 UTC(NIST) *
```

Figure 3-1. Output of the "time of day" service

[View full size image]



The screenshot shows a terminal window titled "Terminal". The window contains the following text:

```
File Edit View Terminal Tags Help
~$ telnet time-A.timefreq.bldrdoc.gov 13
Trying 132.163.4.103...
Connected to time-A.timefreq.bldrdoc.gov.
Escape character is '^]'.
54276 07-06-25 21:37:31 50 0 0 659.0 UTC(NIST) *
Connection closed by foreign host.
~$
```

What is going on? You have connected to the "time of day" service that most UNIX machines constantly run. The particular

server that you connected to is operated by the National Institute of Standards and Technology in Boulder, Colorado, and gives the measurement of a Cesium atomic clock. (Of course, the reported time is not completely accurate due to network delays.)

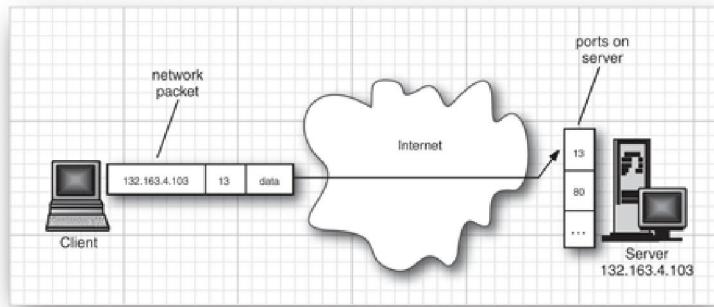
By convention, the "time of day" service is always attached to "port" number 13.

Note

- In network parlance, a port is not a physical device, but an abstraction to facilitate communication between a server and a client (see [Figure 3-2](#)).

Figure 3-2. A client connecting to a server port

[[View full size image](#)]



The server software is continuously running on the remote machine, waiting for any network traffic that wants to chat with port 13. When the operating system on the remote computer receives a network package that contains a request to connect to port number 13, it wakes up the listening server process and establishes the connection. The connection stays up until it is terminated by one of the parties.

When you began the telnet session with `time-A.timefreq.bldrdoc.gov` at port 13, a piece of network software knew enough to convert the string "`time-A.timefreq.bldrdoc.gov`" to its correct Internet Protocol (IP) address, 132.163.4.103. The telnet software then sent a connection request to that address, asking for a connection to port 13. Once the connection was established, the remote program sent back a line of data and then closed the connection. In general, of course, clients and servers engage in a more extensive dialog before one or the other closes the connection.

Here is another experiment, along the same lines, that is a bit more interesting. Do the following:

1. Use telnet to connect to `java.sun.com` on port 80.
2. Type the following, exactly as it appears, without pressing BACKSPACE. Note that there are spaces around the first slash but not the second.

`GET / HTTP/1.0`

3. Now, press the ENTER key two times.

[Figure 3-3](#) shows the response. It should look eerily familiar—you got a page of HTML-formatted text, namely, the main web page for Java technology.

Figure 3-3. Using telnet to access an HTTP port

[[View full size image](#)]

```

Terminal
File Edit View Terminal Tabs Help
~$ telnet java.sun.com 80
Trying 72.5.124.55...
Connected to java.sun.com.
Escape character is '^].
GET / HTTP/1.0

HTTP/1.1 200 OK
Server: Sun-Java-System-Web-Server-6.1
Date: Mon, 25 Jun 2007 21:42:38 GMT
Content-type: text/html; charset=ISO-8859-1
Set-cookie: JSESSIONID=14188FFAC3EAC882A0C92643F21B2FDA; Path=/
Connection: close

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Java Technology</title>
<meta name="keywords" content="Java, platform">
<meta name="collection" content="reference">
<meta name="description" content="Java technology is a portfolio of products that are based on the power of networks and the idea that the same software should run on many different kinds of systems and devices.">
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

```

This is exactly the same process that your web browser goes through to get a web page. It uses HTTP to request web pages from servers. Of course, the browser displays the HTML code more nicely.

Note

- If you try this procedure with a web server that hosts multiple domains with the same IP address, then you will not get the desired web page. (This is the case with smaller web sites that share a single server, such as horstmann.com.) When connecting to such a server, specify the desired host name, like this:

```
GET / HTTP/1.1
Host: horstmann.com
```

Then press the ENTER key two times. (Note that the HTTP version is 1.1.)

Our first network program in [Listing 3-1](#) will do the same thing we did using telnet—connect to a port and print out what it finds.

Listing 3-1. SocketTest.java

Code View:

```

1. import java.io.*;
2. import java.net.*;
3. import java.util.*;
4.
5. /**
6.  * This program makes a socket connection to the atomic clock in Boulder, Colorado, and
7.  * prints the time that the server sends.
8.  * @version 1.20 2004-08-03
9.  * @author Cay Horstmann
10. */
11. public class SocketTest
12. {
13.     public static void main(String[] args)
14.     {
15.         try
16.         {
17.             Socket s = new Socket("time-A.timefreq.bldrdoc.gov", 13);
18.             try
19.             {
20.                 InputStream inStream = s.getInputStream();
21.                 Scanner in = new Scanner(inStream);
22.
23.                 while (in.hasNextLine())
24.                 {
25.                     String line = in.nextLine();
26.                     System.out.println(line);
27.                 }
28.             }
29.             finally
30.             {
31.                 s.close();

```

```
32.         }
33.     }
34.     catch (IOException e)
35.     {
36.         e.printStackTrace();
37.     }
38. }
39. }
```

The key statements of this simple program are as follows:

```
Socket s = new Socket("time-A.timefreq.bldrdoc.gov", 13);
InputStream inStream = s.getInputStream();
```

The first line opens a socket, which is an abstraction for the network software that enables communication out of and into this program. We pass the remote address and the port number to the socket constructor. If the connection fails, then an `UnknownHostException` is thrown. If there is another problem, then an `IOException` occurs. Because `UnknownHostException` is a subclass of `IOException` and this is a sample program, we just catch the superclass.

Once the socket is open, the `getInputStream` method in `java.net.Socket` returns an `InputStream` object that you can use just like any other stream. Once you have grabbed the stream, this program simply prints each input line to standard output. This process continues until the stream is finished and the server disconnects.

This program works only with very simple servers, such as a "time of day" service. In more complex networking programs, the client sends request data to the server, and the server might not immediately disconnect at the end of a response. You will see how to implement that behavior in several examples throughout this chapter.

The `Socket` class is pleasant and easy to use because the Java library hides the complexities of establishing a networking connection and sending data across it. The `java.net` package essentially gives you the same programming interface you would use to work with a file.

Note



In this book, we cover only the Transmission Control Protocol (TCP). The Java platform also supports the User Datagram Protocol (UDP), which can be used to send packets (also called datagrams) with much less overhead than that for TCP. The drawback is that packets need not be delivered in sequential order to the receiving application and can even be dropped altogether. It is up to the recipient to put the packets in order and to request retransmission of missing packets. UDP is well suited for applications in which missing packets can be tolerated, for example, in audio or video streams, or for continuous measurements.



java.net.Socket 1.0

- `Socket(String host, int port)`
constructs a socket to connect to the given host and port.
- `InputStream getInputStream()`
- `OutputStream getOutputStream()`
gets streams to read data from the socket and write data to the socket.

Socket Timeouts

Reading from a socket blocks until data are available. If the host is unreachable, your application waits for a long time and you are at the mercy of the underlying operating system to time out eventually.

You can decide what timeout value is reasonable for your particular application. Then, call the `setSoTimeout` method to set a timeout value (in milliseconds).

```
Socket s = new Socket(.. .);
s.setSoTimeout(10000); // time out after 10 seconds
```

If the timeout value has been set for a socket, then all subsequent read and write operations throw a `SocketTimeoutException` when the timeout has been reached before the operation has completed its work. You can catch that exception and react to the timeout.

```
try
{
    InputStream in = s.getInputStream(); // read from in
    ...
}
catch (InterruptedIOException exception)
{
    react to timeout
}
```

There is one additional timeout issue that you need to address: The constructor

```
Socket(String host, int port)
```

can block indefinitely until an initial connection to the host is established.

You can overcome this problem by first constructing an unconnected socket and then connecting it with a timeout:

```
Socket s = new Socket();
s.connect(new InetSocketAddress(host, port), timeout);
```

See the "[Interruptible Sockets](#)" section beginning on page [184](#) if you want to allow users to interrupt the socket connection at any time.



java.net.Socket 1.0

- `Socket()` 1.1
creates a socket that has not yet been connected.
- `void connect(SocketAddress address)` 1.4
connects this socket to the given address.
- `void connect(SocketAddress address, int timeoutInMilliseconds)` 1.4
connects this socket to the given address or returns if the time interval expired.
- `void setSoTimeout(int timeoutInMilliseconds)` 1.1
sets the blocking time for read requests on this socket. If the timeout is reached, then an `InterruptedIOException` is raised.
- `boolean isConnected()` 1.4
returns `true` if the socket is connected.
- `boolean isClosed()` 1.4
returns `true` if the socket is closed.

Internet Addresses

Usually, you don't have to worry too much about Internet addresses—the numerical host addresses that consist of four bytes (or, with IPv6, 16 bytes) such as 132.163.4.102. However, you can use the `InetAddress` class if you need to convert between host names and Internet addresses.

The `java.net` package supports IPv6 Internet addresses, provided the host operating system does.

The static `getByName` method returns an `InetAddress` object of a host. For example,

```
InetAddress address = InetAddress.getByName("time-A.timefreq.bldrdoc.gov");
```

returns an `InetAddress` object that encapsulates the sequence of four bytes 132.163.4.104. You can access the bytes with the `getAddress` method.

```
byte[] addressBytes = address.getAddress();
```

Some host names with a lot of traffic correspond to multiple Internet addresses, to facilitate load balancing. For example, at the time of this writing, the host name `java.sun.com` corresponds to three different Internet addresses. One of them is picked at random when the host is accessed. You can get all hosts with the `getAllByName` method.

```
InetAddress[] addresses = InetAddress.getAllByName(host);
```

Finally, you sometimes need the address of the local host. If you simply ask for the address of `localhost`, you always get the local loopback address 127.0.0.1, which cannot be used by others to connect to your computer. Instead, use the static `getLocalHost` method to get the address of your local host.

```
InetAddress address = InetAddress.getLocalHost();
```

[Listing 3-2](#) is a simple program that prints the Internet address of your local host if you do not specify any command-line parameters, or all Internet addresses of another host if you specify the host name on the command line, such as

```
java InetAddressTest java.sun.com
```

Listing 3-2. InetAddressTest.java

Code View:

```
1. import java.net.*;
2.
3. /**
4.  * This program demonstrates the InetAddress class. Supply a host name as command line
5.  * argument, or run without command line arguments to see the address of the local host.
6.  * @version 1.01 2001-06-26
7.  * @author Cay Horstmann
8. */
9. public class InetAddressTest
10. {
11.     public static void main(String[] args)
12.     {
13.         try
14.         {
15.             if (args.length > 0)
16.             {
17.                 String host = args[0];
18.                 InetAddress[] addresses = InetAddress.getAllByName(host);
19.                 for (InetAddress a : addresses)
20.                     System.out.println(a);
21.             }
22.             else
23.             {
24.                 InetAddress localHostAddress = InetAddress.getLocalHost();
25.                 System.out.println(localHostAddress);
26.             }
27.         }
28.         catch (Exception e)
29.         {
30.             e.printStackTrace();
31.         }
32.     }
33. }
```



java.net.InetAddress 1.0

- static InetAddress getByName(String host)
- static InetAddress[] getAllByName(String host)
constructs an InetAddress, or an array of all Internet addresses, for the given host name.
- static InetAddress getLocalHost()
constructs an InetAddress for the local host.
- byte[] getAddress()
returns an array of bytes that contains the numerical address.
- String getHostAddress()
returns a string with decimal numbers, separated by periods, for example, "132.163.4.102".
- String getHostName()
returns the host name.



Chapter 3. Networking

- [CONNECTING TO A SERVER](#)
- [IMPLEMENTING SERVERS](#)
- [INTERRUPTIBLE SOCKETS](#)
- [SENDING E-MAIL](#)
- [MAKING URL CONNECTIONS](#)

We begin this chapter by reviewing basic networking concepts. We then move on to writing Java programs that connect to network services. We show you how network clients and servers are implemented. Finally, you will see how to send e-mail from a Java program and how to harvest information from a web server.

Connecting to a Server

Before writing our first network program, let's learn about a great debugging tool for network programming that you already have, namely, telnet. Telnet is preinstalled on most systems. You should be able to launch it by typing `telnet` from a command shell.

Note



In Windows Vista, telnet is installed but deactivated by default. To activate it, go to the Control Panel, select Programs, click "Turn Windows Features On or Off", and select the "Telnet client" checkbox. The Windows firewall also blocks quite a few network ports that we use in this chapter; you might need an administrator account to unblock them.

You may have used telnet to connect to a remote computer, but you can use it to communicate with other services provided by Internet hosts as well. Here is an example of what you can do. Type

```
telnet time-A.timefreq.bldrdoc.gov 13
```

As [Figure 3-1](#) shows, you should get back a line like this:

54276 07-06-25 21:37:31 50 0 0 659.0 UTC(NIST) *

Figure 3-1. Output of the "time of day" service

[\[View full size image\]](#)

The screenshot shows a Windows-style terminal window titled 'Terminal'. The window contains the following text:

```

File Edit View Terminal Tabs Help
~$ telnet time-A.timefreq.bldrdoc.gov 13
Trying 132.163.4.103...
Connected to time-A.timefreq.bldrdoc.gov.
Escape character is '^]'.

54276 07-06-25 21:37:31 50 0 0 659.0 UTC(NIST) *
Connection closed by foreign host.
~$ █

```

What is going on? You have connected to the "time of day" service that most UNIX machines constantly run. The particular server that you connected to is operated by the National Institute of Standards and Technology in Boulder, Colorado, and gives the measurement of a Cesium atomic clock. (Of course, the reported time is not completely accurate due to network delays.)

By convention, the "time of day" service is always attached to "port" number 13.

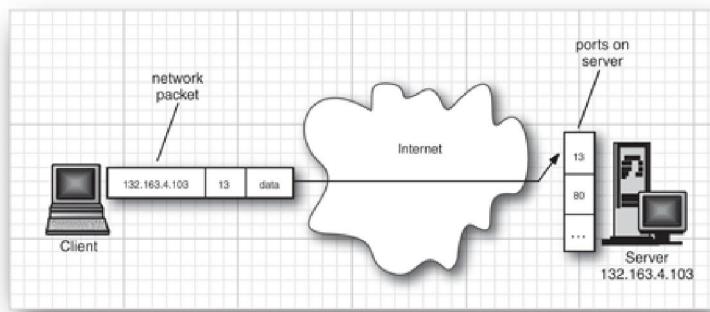
Note



In network parlance, a port is not a physical device, but an abstraction to facilitate communication between a server and a client (see [Figure 3-2](#)).

Figure 3-2. A client connecting to a server port

[\[View full size image\]](#)



The server software is continuously running on the remote machine, waiting for any network traffic that wants to chat with port 13. When the operating system on the remote computer receives a network package that contains a request to connect to port number 13, it wakes up the listening server process and establishes the connection. The connection stays up until it is terminated by one of the parties.

When you began the telnet session with `time-A.timefreq.bldrdoc.gov` at port 13, a piece of network software knew enough to convert the string "`time-A.timefreq.bldrdoc.gov`" to its correct Internet Protocol (IP) address, 132.163.4.103. The telnet software then sent a connection request to that address, asking for a connection to port 13. Once the connection was established, the remote program sent back a line of data and then closed the connection. In general, of course, clients and servers engage in a more extensive dialog before one or the other closes the connection.

Here is another experiment, along the same lines, that is a bit more interesting. Do the following:

1. Use telnet to connect to `java.sun.com` on port 80.
2. Type the following, exactly as it appears, without pressing BACKSPACE. Note that there are spaces around the first slash but not the second.

```
GET / HTTP/1.0
```

3. Now, press the ENTER key two times.

[Figure 3-3](#) shows the response. It should look eerily familiar—you got a page of HTML-formatted text, namely, the main web page for Java technology.

Figure 3-3. Using telnet to access an HTTP port

[View full size image]

```

Terminal
File Edit View Terminal Help
$ telnet java.sun.com 80
Trying 72.5.124.55...
Connected to java.sun.com.
Escape character is ']'.
GET / HTTP/1.0

HTTP/1.1 200 OK
Server: Sun-Java-System-Web-Server-6.1
Date: Mon, 25 Jun 2007 21:42:38 GMT
Content-type: text/html; charset=ISO-8859-1
Set-cookie: JSESSIONID=1418FFAC3EAC882A0C92643F21B2FDA; Path=/
Connection: close

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>Java Technology</title>
<meta name="keywords" content="Java, platform">
<meta name="collection" content="reference">
<meta name="description" content="Java technology is a portfolio of products that are based on the power of networks and the idea that the same software should run on many different kinds of systems and devices.">
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
```

This is exactly the same process that your web browser goes through to get a web page. It uses HTTP to request web pages from servers. Of course, the browser displays the HTML code more nicely.

Note

If you try this procedure with a web server that hosts multiple domains with the same IP address, then you will not get the desired web page. (This is the case with smaller web sites that share a single server, such as `horstmann.com`.) When connecting to such a server, specify the desired host name, like this:

```
GET / HTTP/1.1
Host: horstmann.com
```

Then press the ENTER key two times. (Note that the HTTP version is 1.1.)

Our first network program in [Listing 3-1](#) will do the same thing we did using telnet—connect to a port and print out what it finds.

Listing 3-1. SocketTest.java

Code View:

```

1. import java.io.*;
2. import java.net.*;
3. import java.util.*;
4.
5. /**
6. * This program makes a socket connection to the atomic clock in Boulder, Colorado, and
7. * prints the time that the server sends.
8. * @version 1.20 2004-08-03
9. * @author Cay Horstmann
10.*/
11. public class SocketTest
12. {
13.     public static void main(String[] args)

```

```

14.    {
15.        try
16.        {
17.            Socket s = new Socket("time-A.timefreq.bldrdoc.gov", 13);
18.            try
19.            {
20.                InputStream inStream = s.getInputStream();
21.                Scanner in = new Scanner(inStream);
22.
23.                while (in.hasNextLine())
24.                {
25.                    String line = in.nextLine();
26.                    System.out.println(line);
27.                }
28.            }
29.            finally
30.            {
31.                s.close();
32.            }
33.        }
34.        catch (IOException e)
35.        {
36.            e.printStackTrace();
37.        }
38.    }
39. }
```

The key statements of this simple program are as follows:

```
Socket s = new Socket("time-A.timefreq.bldrdoc.gov", 13);
InputStream inStream = s.getInputStream();
```

The first line opens a socket, which is an abstraction for the network software that enables communication out of and into this program. We pass the remote address and the port number to the socket constructor. If the connection fails, then an `UnknownHostException` is thrown. If there is another problem, then an `IOException` occurs. Because `UnknownHostException` is a subclass of `IOException` and this is a sample program, we just catch the superclass.

Once the socket is open, the `getInputStream` method in `java.net.Socket` returns an `InputStream` object that you can use just like any other stream. Once you have grabbed the stream, this program simply prints each input line to standard output. This process continues until the stream is finished and the server disconnects.

This program works only with very simple servers, such as a "time of day" service. In more complex networking programs, the client sends request data to the server, and the server might not immediately disconnect at the end of a response. You will see how to implement that behavior in several examples throughout this chapter.

The `Socket` class is pleasant and easy to use because the Java library hides the complexities of establishing a networking connection and sending data across it. The `java.net` package essentially gives you the same programming interface you would use to work with a file.

Note



In this book, we cover only the Transmission Control Protocol (TCP). The Java platform also supports the User Datagram Protocol (UDP), which can be used to send packets (also called datagrams) with much less overhead than that for TCP. The drawback is that packets need not be delivered in sequential order to the receiving application and can even be dropped altogether. It is up to the recipient to put the packets in order and to request retransmission of missing packets. UDP is well suited for applications in which missing packets can be tolerated, for example, in audio or video streams, or for continuous measurements.



java.net.Socket 1.0

- `Socket(String host, int port)`

constructs a socket to connect to the given host and port.

- `InputStream getInputStream()`
 - `OutputStream getOutputStream()`
- gets streams to read data from the socket and write data to the socket.

Socket Timeouts

Reading from a socket blocks until data are available. If the host is unreachable, your application waits for a long time and you are at the mercy of the underlying operating system to time out eventually.

You can decide what timeout value is reasonable for your particular application. Then, call the `setSoTimeout` method to set a timeout value (in milliseconds).

```
Socket s = new Socket(.. .);
s.setSoTimeout(10000); // time out after 10 seconds
```

If the timeout value has been set for a socket, then all subsequent read and write operations throw a `SocketTimeoutException` when the timeout has been reached before the operation has completed its work. You can catch that exception and react to the timeout.

```
try
{
    InputStream in = s.getInputStream(); // read from in
    ...
}
catch (InterruptedIOException exception)
{
    react to timeout
}
```

There is one additional timeout issue that you need to address: The constructor

```
Socket(String host, int port)
```

can block indefinitely until an initial connection to the host is established.

You can overcome this problem by first constructing an unconnected socket and then connecting it with a timeout:

```
Socket s = new Socket();
s.connect(new InetSocketAddress(host, port), timeout);
```

See the "[Interruptible Sockets](#)" section beginning on page [184](#) if you want to allow users to interrupt the socket connection at any time.



java.net.Socket 1.0

- `Socket() 1.1`
creates a socket that has not yet been connected.
- `void connect(SocketAddress address) 1.4`
connects this socket to the given address.
- `void connect(SocketAddress address, int timeoutInMilliseconds) 1.4`
connects this socket to the given address or returns if the time interval expired.
- `void setSoTimeout(int timeoutInMilliseconds) 1.1`
sets the blocking time for read requests on this socket. If the timeout is reached, then an `InterruptedIOException` is raised.

- `boolean isConnected()` 1.4
returns `true` if the socket is connected.
- `boolean isClosed()` 1.4
returns `true` if the socket is closed.

Internet Addresses

Usually, you don't have to worry too much about Internet addresses—the numerical host addresses that consist of four bytes (or, with IPv6, 16 bytes) such as 132.163.4.102. However, you can use the `InetAddress` class if you need to convert between host names and Internet addresses.

The `java.net` package supports IPv6 Internet addresses, provided the host operating system does.

The static `getByName` method returns an `InetAddress` object of a host. For example,

```
InetAddress address = InetAddress.getByName("time-A.timefreq.bldrdoc.gov");
```

returns an `InetAddress` object that encapsulates the sequence of four bytes 132.163.4.104. You can access the bytes with the `getAddress` method.

```
byte[] addressBytes = address.getAddress();
```

Some host names with a lot of traffic correspond to multiple Internet addresses, to facilitate load balancing. For example, at the time of this writing, the host name `java.sun.com` corresponds to three different Internet addresses. One of them is picked at random when the host is accessed. You can get all hosts with the `getAllByName` method.

```
InetAddress[] addresses = InetAddress.getAllByName(host);
```

Finally, you sometimes need the address of the local host. If you simply ask for the address of `localhost`, you always get the local loopback address 127.0.0.1, which cannot be used by others to connect to your computer. Instead, use the static `getLocalHost` method to get the address of your local host.

```
InetAddress address = InetAddress.getLocalHost();
```

[Listing 3-2](#) is a simple program that prints the Internet address of your local host if you do not specify any command-line parameters, or all Internet addresses of another host if you specify the host name on the command line, such as

```
java InetAddressTest java.sun.com
```

Listing 3-2. `InetAddressTest.java`

Code View:

```
1. import java.net.*;  
2.  
3. /**  
4.  * This program demonstrates the InetAddress class. Supply a host name as command line  
5.  * argument, or run without command line arguments to see the address of the local host.  
6.  * @version 1.01 2001-06-26  
7.  * @author Cay Horstmann  
8.  */  
9. public class InetAddressTest  
10. {  
11.     public static void main(String[] args)  
12.     {  
13.         try  
14.         {  
15.             if (args.length > 0)  
16.             {  
17.                 String host = args[0];  
18.                 InetAddress[] addresses = InetAddress.getAllByName(host);  
19.                 for (InetAddress a : addresses)
```

```

20.         System.out.println(a);
21.     }
22.     else
23.     {
24.         InetAddress localHostAddress = InetAddress.getLocalHost();
25.         System.out.println(localHostAddress);
26.     }
27. }
28. catch (Exception e)
29. {
30.     e.printStackTrace();
31. }
32. }
33. }
```

API**java.net.InetAddress 1.0**

- static InetAddress getByName(String host)
- static InetAddress[] getAllByName(String host)

constructs an InetAddress, or an array of all Internet addresses, for the given host name.
- static InetAddress getLocalHost()

constructs an InetAddress for the local host.
- byte[] getAddress()

returns an array of bytes that contains the numerical address.
- String getHostAddress()

returns a string with decimal numbers, separated by periods, for example, "132.163.4.102".
- String getHostName()

returns the host name.



Implementing Servers

Now that we have implemented a basic network client that receives data from the Internet, let's implement a simple server that can send information to clients. Once you start the server program, it waits for some client to attach to its port. We chose port number 8189, which is not used by any of the standard services. The `ServerSocket` class establishes a socket. In our case, the command

```
ServerSocket s = new ServerSocket(8189);
```

establishes a server that monitors port 8189. The command

```
Socket incoming = s.accept();
```

tells the program to wait indefinitely until a client connects to that port. Once someone connects to this port by sending the correct request over the network, this method returns a `Socket` object that represents the connection that was made. You can use this object to get input and output streams, as is shown in the following code:

```
InputStream inStream = incoming.getInputStream();
```

```
OutputStream outStream = incoming.getOutputStream();
```

Everything that the server sends to the server output stream becomes the input of the client program, and all the output from the client program ends up in the server input stream.

In all the examples in this chapter, we transmit text through sockets. We therefore turn the streams into scanners and writers.

```
Scanner in = new Scanner(inStream);
PrintWriter out = new PrintWriter(outStream, true /* autoFlush */);
```

Let's send the client a greeting:

```
out.println("Hello! Enter BYE to exit.");
```

When you use telnet to connect to this server program at port 8189, you will see the preceding greeting on the terminal screen.

In this simple server, we just read the client input, a line at a time, and echo it. This demonstrates that the program receives the client's input. An actual server would obviously compute and return an answer that depended on the input.

```
String line = in.nextLine();
out.println("Echo: " + line);
if (line.trim().equals("BYE")) done = true;
```

In the end, we close the incoming socket.

```
incoming.close();
```

That is all there is to it. Every server program, such as an HTTP web server, continues performing this loop:

1. It receives a command from the client ("get me this information") through an incoming data stream.
2. It decodes the client command.
3. It gathers the information that the client requested.
4. It sends the information to the client through the outgoing data stream.

[Listing 3-3](#) is the complete program.

Listing 3-3. EchoServer.java

Code View:

```
1. import java.io.*;
2. import java.net.*;
3. import java.util.*;
4.
5. /**
6.  * This program implements a simple server that listens to port 8189 and echoes back all
7.  * client input.
8.  * @version 1.20 2004-08-03
9.  * @author Cay Horstmann
10. */
11. public class EchoServer
12. {
13.     public static void main(String[] args)
14.     {
15.         try
16.         {
17.             // establish server socket
18.             ServerSocket s = new ServerSocket(8189);
19.
20.             // wait for client connection
21.             Socket incoming = s.accept();
22.             try
23.             {
24.                 InputStream inStream = incoming.getInputStream();
25.                 OutputStream outStream = incoming.getOutputStream();
26.             }
```

```
27.     Scanner in = new Scanner(inStream);
28.     PrintWriter out = new PrintWriter(outStream, true /* autoFlush */);
29.
30.     out.println("Hello! Enter BYE to exit.");
31.
32.     // echo client input
33.     boolean done = false;
34.     while (!done && in.hasNextLine())
35.     {
36.         String line = in.nextLine();
37.         out.println("Echo: " + line);
38.         if (line.trim().equals("BYE")) done = true;
39.     }
40. }
41. finally
42. {
43.     incoming.close();
44. }
45. }
46. catch (IOException e)
47. {
48.     e.printStackTrace();
49. }
50. }
51. }
```

To try it out, compile and run the program. Then, use telnet to connect to the server localhost (or IP address 127.0.0.1) and port 8189.

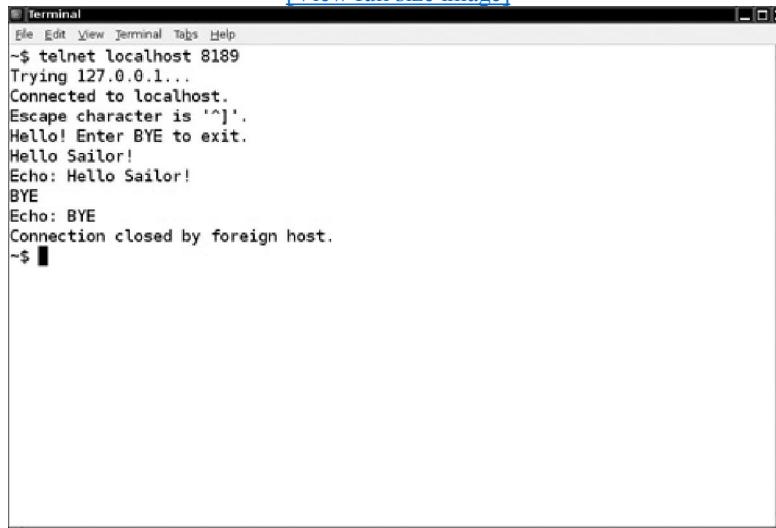
If you are connected directly to the Internet, then anyone in the world can access your echo server, provided they know your IP address and the magic port number.

When you connect to the port, you will see the message shown in [Figure 3-4](#):

Hello! Enter BYE to exit.

Figure 3-4. Accessing an echo server

[\[View full size image\]](#)



Type anything and watch the input echo on your screen. Type **BYE** (all uppercase letters) to disconnect. The server program will terminate as well.



java.net.ServerSocket 1.0

- ServerSocket(int port)

creates a server socket that monitors a port.

- `Socket accept()`

waits for a connection. This method blocks (i.e., idles) the current thread until the connection is made. The method returns a `Socket` object through which the program can communicate with the connecting client.

- `void close()`

closes the server socket.

Serving Multiple Clients

There is one problem with the simple server in the preceding example. Suppose we want to allow multiple clients to connect to our server at the same time. Typically, a server runs constantly on a server computer, and clients from all over the Internet might want to use the server at the same time. Rejecting multiple connections allows any one client to monopolize the service by connecting to it for a long time. We can do much better through the magic of threads.

Every time we know the program has established a new socket connection—that is, when the call to `accept` was successful—we will launch a new thread to take care of the connection between the server and that client. The main program will just go back and wait for the next connection. For this to happen, the main loop of the server should look like this:

```
while (true)
{
    Socket incoming = s.accept();
    Runnable r = new ThreadedEchoHandler(incoming);

    Thread t = new Thread(r);
    t.start();
}
```

The `ThreadedEchoHandler` class implements `Runnable` and contains the communication loop with the client in its `run` method.

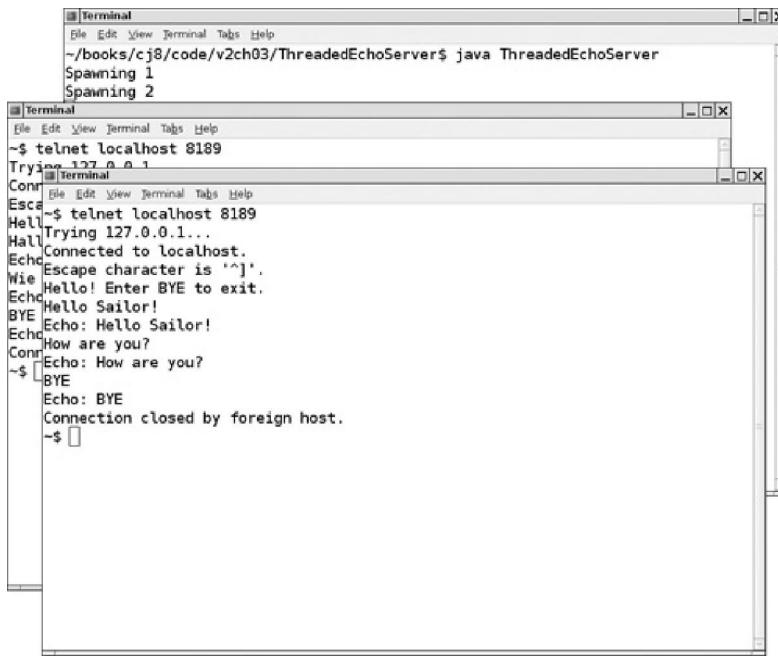
```
class ThreadedEchoHandler implements Runnable
{ . .
  public void run()
  {
    try
    {
      InputStream inStream = incoming.getInputStream();
      OutputStream outStream = incoming.getOutputStream();
      process input and send response
      incoming.close();
    }
    catch(IOException e)
    {
      handle exception
    }
  }
}
```

Because each connection starts a new thread, multiple clients can connect to the server at the same time. You can easily check this out.

1. Compile and run the server program ([Listing 3-4](#)).
2. Open several telnet windows as we have in [Figure 3-5](#).

Figure 3-5. Several telnet windows communicating simultaneously

[\[View full size image\]](#)



3. Switch between windows and type commands. Note that you can communicate through all of them simultaneously.
4. When you are done, switch to the window from which you launched the server program and use CTRL+C to kill it.

Note

 In this program, we spawn a separate thread for each connection. This approach is not satisfactory for high-performance servers. You can achieve greater server throughput by using features of the `java.nio` package. See <http://www.ibm.com/developerworks/java/library/j-javaio> for more information.

Listing 3-4. ThreadedEchoServer.java

Code View:

```

1. import java.io.*;
2. import java.net.*;
3. import java.util.*;
4.
5. /**
6.  * This program implements a multithreaded server that listens to port 8189 and echoes back
7.  * all client input.
8.  * @author Cay Horstmann
9.  * @version 1.20 2004-08-03
10.*/
11. public class ThreadedEchoServer
12. {
13.     public static void main(String[] args )
14.     {
15.         try
16.         {
17.             int i = 1;
18.             ServerSocket s = new ServerSocket(8189);
19.
20.             while (true)
21.             {
22.                 Socket incoming = s.accept();
23.                 System.out.println("Spawning " + i);
24.                 Runnable r = new ThreadedEchoHandler(incoming);
25.                 Thread t = new Thread(r);
26.                 t.start();
27.                 i++;
28.             }
29.         }
30.         catch (IOException e)
31.         {
32.             e.printStackTrace();
33.         }
34.     }
35. }
```

```
36.
37. /**
38.      This class handles the client input for one server socket connection.
39. */
40. class ThreadedEchoHandler implements Runnable
41. {
42.     /**
43.         Constructs a handler.
44.         @param i the incoming socket
45.         @param c the counter for the handlers (used in prompts)
46.     */
47.     public ThreadedEchoHandler(Socket i)
48.     {
49.         incoming = i;
50.     }
51.
52.     public void run()
53.     {
54.         try
55.         {
56.             try
57.             {
58.                 InputStream inStream = incoming.getInputStream();
59.                 OutputStream outStream = incoming.getOutputStream();
60.
61.                 Scanner in = new Scanner(inStream);
62.                 PrintWriter out = new PrintWriter(outStream, true /* autoFlush */);
63.
64.                 out.println( "Hello! Enter BYE to exit." );
65.
66.                 // echo client input
67.                 boolean done = false;
68.                 while (!done && in.hasNextLine())
69.                 {
70.                     String line = in.nextLine();
71.                     out.println("Echo: " + line);
72.                     if (line.trim().equals("BYE"))
73.                         done = true;
74.                 }
75.             }
76.             finally
77.             {
78.                 incoming.close();
79.             }
80.         }
81.         catch (IOException e)
82.         {
83.             e.printStackTrace();
84.         }
85.     }
86.
87.     private Socket incoming;
88. }
```

Half-Close

The half-close provides the ability for one end of a socket connection to terminate its output, while still receiving data from the other end.

Here is a typical situation. Suppose you transmit data to the server but you don't know at the outset how much data you have. With a file, you'd just close the file at the end of the data. However, if you close a socket, then you immediately disconnect from the server, and you cannot read the response.

The half-close overcomes this problem. You can close the output stream of a socket, thereby indicating to the server the end of the requested data, but keep the input stream open.

The client side looks like this:

```
Socket socket = new Socket(host, port);
Scanner in = new Scanner(socket.getInputStream());
PrintWriter writer = new PrintWriter(socket.getOutputStream());
// send request data
```

```
writer.print(. . .);
writer.flush();
socket.shutdownOutput();
// now socket is half closed
// read response data
while (in.hasNextLine()) != null) { String line = in.nextLine(); . . . }
socket.close();
```

The server side simply reads input until the end of the input stream is reached. Then it sends the response.

Of course, this protocol is only useful for one-shot services such as HTTP where the client connects, issues a request, catches the response, and then disconnects.

API

java.net.Socket 1.0

- void shutdownOutput() 1.3
sets the output stream to "end of stream."
- void shutdownInput() 1.3
sets the input stream to "end of stream."
- boolean isOutputShutdown() 1.4
returns true if output has been shut down.
- boolean isInputShutdown() 1.4
returns true if input has been shut down.



Interruptible Sockets

When you connect to a socket, the current thread blocks until the connection has been established or a timeout has elapsed. Similarly, when you read or write data through a socket, the current thread blocks until the operation is successful or has timed out.

In interactive applications, you would like to give users an option to simply cancel a socket connection that does not appear to produce results. However, if a thread blocks on an unresponsive socket, you cannot unblock it by calling `interrupt`.

To interrupt a socket operation, you use a `SocketChannel`, a feature of the `java.nio` package. Open the `SocketChannel` like this:

Code View:

```
SocketChannel channel = SocketChannel.open(new InetSocketAddress(host, port));
```

A channel does not have associated streams. Instead, it has `read` and `write` methods that make use of `Buffer` objects. (See [Chapter 1](#) for more information about NIO buffers.) These methods are declared in interfaces `ReadableByteChannel` and `WritableByteChannel`.

If you don't want to deal with buffers, you can use the `Scanner` class to read from a `SocketChannel` because `Scanner` has a constructor with a `ReadableByteChannel` parameter:

```
Scanner in = new Scanner(channel);
```

To turn a channel into an output stream, use the static `Channels.newOutputStream` method.

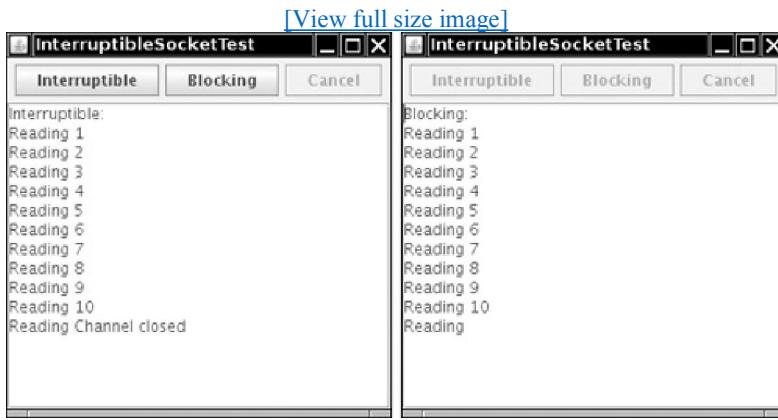
```
OutputStream outStream = Channels.newOutputStream(channel);
```

That's all you need to do. Whenever a thread is interrupted during an open, read, or write operation, the operation does not block, but is terminated with an exception.

The program in [Listing 3-5](#) contrasts interruptible and blocking sockets. A server sends numbers and pretends to be stuck after the tenth number. Click on either button, and a thread is started that connects to the server and prints the output. The first thread uses an interruptible socket; the second thread uses a blocking socket. If you click the Cancel button within the first ten numbers, you can interrupt either thread.

However, after the first ten numbers, you can only interrupt the first thread. The second thread keeps blocking until the server finally closes the connection (see [Figure 3-6](#)).

Figure 3-6. Interrupting a socket



Listing 3-5. InterruptibleSocketTest.java

Code View:

```

1. import java.awt.*;
2. import java.awt.event.*;
3. import java.util.*;
4. import java.net.*;
5. import java.io.*;
6. import java.nio.channels.*;
7. import javax.swing.*;
8.
9. /**
10. * This program shows how to interrupt a socket channel.
11. * @author Cay Horstmann
12. * @version 1.01 2007-06-25
13. */
14. public class InterruptibleSocketTest
15. {
16.     public static void main(String[] args)
17.     {
18.         EventQueue.invokeLater(new Runnable()
19.         {
20.             public void run()
21.             {
22.                 JFrame frame = new InterruptibleSocketFrame();
23.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24.                 frame.setVisible(true);
25.             }
26.         });
27.     }
28. }
29.
30. class InterruptibleSocketFrame extends JFrame
31. {
32.     public InterruptibleSocketFrame()
33.     {
34.         setSize(WIDTH, HEIGHT);
35.         setTitle("InterruptibleSocketTest");
36.
37.         JPanel northPanel = new JPanel();
38.         add(northPanel, BorderLayout.NORTH);
39.
40.         messages = new JTextArea();

```

```
41.     add(new JScrollPane(messages));
42.
43.     interruptibleButton = new JButton("Interruptible");
44.     blockingButton = new JButton("Blocking");
45.
46.     northPanel.add(interruptibleButton);
47.     northPanel.add(blockingButton);
48.
49.     interruptibleButton.addActionListener(new ActionListener()
50.     {
51.         public void actionPerformed(ActionEvent event)
52.         {
53.             interruptibleButton.setEnabled(false);
54.             blockingButton.setEnabled(false);
55.             cancelButton.setEnabled(true);
56.             connectThread = new Thread(new Runnable()
57.             {
58.                 public void run()
59.                 {
60.                     try
61.                     {
62.                         connectInterruptibly();
63.                     }
64.                     catch (IOException e)
65.                     {
66.                         messages.append("\nInterruptibleSocketTest.connectInterruptibly: "
67.                             + e);
68.                     }
69.                 }
70.             });
71.             connectThread.start();
72.         }
73.     });
74.
75.     blockingButton.addActionListener(new ActionListener()
76.     {
77.         public void actionPerformed(ActionEvent event)
78.         {
79.             interruptibleButton.setEnabled(false);
80.             blockingButton.setEnabled(false);
81.             cancelButton.setEnabled(true);
82.             connectThread = new Thread(new Runnable()
83.             {
84.                 public void run()
85.                 {
86.                     try
87.                     {
88.                         connectBlocking();
89.                     }
89.                     catch (IOException e)
90.                     {
91.                         messages.append("\nInterruptibleSocketTest.connectBlocking: " + e);
92.                     }
93.                 }
94.             });
95.             connectThread.start();
96.         }
97.     });
98. });
99.
100. cancelButton = new JButton("Cancel");
101. cancelButton.setEnabled(false);
102. northPanel.add(cancelButton);
103. cancelButton.addActionListener(new ActionListener()
104. {
105.     public void actionPerformed(ActionEvent event)
106.     {
107.         connectThread.interrupt();
108.         cancelButton.setEnabled(false);
109.     }
110. });
111. server = new TestServer();
112. new Thread(server).start();
113. }
114.
115. /**
116. * Connects to the test server, using interruptible I/O
117. */
118. public void connectInterruptibly() throws IOException
119. {
120.     messages.append("Interruptible:\n");
121.     SocketChannel channel = SocketChannel.open(new InetSocketAddress("localhost", 8189));
122.     try
```

```
123.     {
124.         in = new Scanner(channel);
125.         while (!Thread.currentThread().isInterrupted())
126.         {
127.             messages.append("Reading ");
128.             if (in.hasNextLine())
129.             {
130.                 String line = in.nextLine();
131.                 messages.append(line);
132.                 messages.append("\n");
133.             }
134.         }
135.     }
136.     finally
137.     {
138.         channel.close();
139.         EventQueue.invokeLater(new Runnable()
140.         {
141.             public void run()
142.             {
143.                 messages.append("Channel closed\n");
144.                 interruptibleButton.setEnabled(true);
145.                 blockingButton.setEnabled(true);
146.             }
147.         });
148.     }
149. }
150.
151. /**
152. * Connects to the test server, using blocking I/O
153. */
154. public void connectBlocking() throws IOException
155. {
156.     messages.append("Blocking:\n");
157.     Socket sock = new Socket("localhost", 8189);
158.     try
159.     {
160.         in = new Scanner(sock.getInputStream());
161.         while (!Thread.currentThread().isInterrupted())
162.         {
163.             messages.append("Reading ");
164.             if (in.hasNextLine())
165.             {
166.                 String line = in.nextLine();
167.                 messages.append(line);
168.                 messages.append("\n");
169.             }
170.         }
171.     }
172.     finally
173.     {
174.         sock.close();
175.         EventQueue.invokeLater(new Runnable()
176.         {
177.             public void run()
178.             {
179.                 messages.append("Socket closed\n");
180.                 interruptibleButton.setEnabled(true);
181.                 blockingButton.setEnabled(true);
182.             }
183.         });
184.     }
185. }
186.
187. /**
188. * A multithreaded server that listens to port 8189 and sends numbers to the client,
189. * simulating a hanging server after 10 numbers.
190. */
191. class TestServer implements Runnable
192. {
193.     public void run()
194.     {
195.         try
196.         {
197.             ServerSocket s = new ServerSocket(8189);
198.
199.             while (true)
200.             {
201.                 Socket incoming = s.accept();
202.                 Runnable r = new TestServerHandler(incoming);
203.                 Thread t = new Thread(r);
204.                 t.start();
```

```
205.         }
206.     }
207.     catch (IOException e)
208.     {
209.         messages.append("\nTestServer.run: " + e);
210.     }
211. }
212. }
213.
214. /**
215. * This class handles the client input for one server socket connection.
216. */
217. class TestServerHandler implements Runnable
218.
219. {
220.     /**
221.      * Constructs a handler.
222.      * @param i the incoming socket
223.     */
224.     public TestServerHandler(Socket i)
225.     {
226.         incoming = i;
227.     }
228.
229.     public void run()
230.     {
231.         try
232.         {
233.             OutputStream outStream = incoming.getOutputStream();
234.             PrintWriter out = new PrintWriter(outStream, true /* autoFlush */);
235.             while (counter < 100)
236.             {
237.                 counter++;
238.                 if (counter <= 10) out.println(counter);
239.                 Thread.sleep(100);
240.             }
241.             incoming.close();
242.             messages.append("Closing server\n");
243.         }
244.         catch (Exception e)
245.         {
246.             messages.append("\nTestServerHandler.run: " + e);
247.         }
248.     }
249.
250.     private Socket incoming;
251.     private int counter;
252.
253.     private Scanner in;
254.     private JButton interruptibleButton;
255.     private JButton blockingButton;
256.     private JButton cancelButton;
257.     private JTextArea messages;
258.     private TestServer server;
259.     private Thread connectThread;
260.
261.     public static final int WIDTH = 300;
262.     public static final int HEIGHT = 300;
263. }
```



java.net.InetSocketAddress 1.4

- InetSocketAddress(String hostname, int port)

constructs an address object with the given host and port, resolving the host name during construction. If the host name cannot be resolved, then the address object's unresolved property is set to true.

- boolean isUnresolved()

returns true if this address object could not be resolved.

API**java.nio.channels.SocketChannel 1.4**

- static SocketChannel open(SocketAddress address)
opens a socket channel and connects it to a remote address.

API**java.nio.channels.Channels 1.4**

- static InputStream newInputStream(ReadableByteChannel channel)
constructs an input stream that reads from the given channel.
- static OutputStream newOutputStream(WritableByteChannel channel)
constructs an output stream that writes to the given channel.



Sending E-Mail

In this section, we show you a practical example of socket programming: a program that sends e-mail to a remote site.

To send e-mail, you make a socket connection to port 25, the Simple Mail Transport Protocol (SMTP) port. The Simple Mail Transport Protocol (SMTP) describes the format for e-mail messages. You can connect to any server that runs an SMTP service. However, the server must be willing to accept your request. It used to be that SMTP servers were routinely willing to route e-mail from anyone, but in these days of spam floods, most servers now have built-in checks and accept requests only from users or IP address ranges that they trust.

Once you are connected to the server, send a mail header (in the SMTP format, which is easy to generate), followed by the mail message.

Here are the details:

1. Open a socket to your host.

```
Socket s = new Socket("mail.yourserver.com", 25); // 25 is SMTP
PrintWriter out = new PrintWriter(s.getOutputStream());
```

2. Send the following information to the print stream:

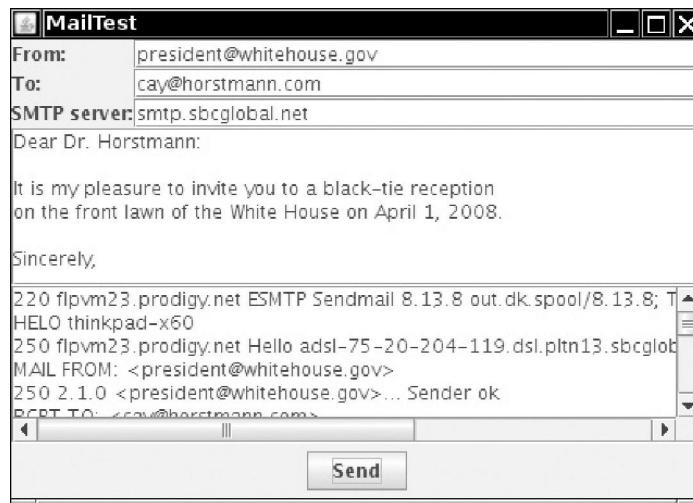
```
HELO sending host
MAIL FROM: <sender e-mail address>
RCPT TO: <recipient e-mail address>
DATA
mail message
(any number of lines)
.
QUIT
```

The SMTP specification (RFC 821) states that lines must be terminated with `\r` followed by `\n`.

Some SMTP servers do not check the veracity of the information—you might be able to supply any sender you like. (Keep this in mind the next time you get an e-mail message from president@whitehouse.gov inviting you to a black-tie affair on the front lawn. It is fairly easy to find an SMTP server that will relay a fake message.)

The program in [Listing 3-6](#) is a simple e-mail program. As you can see in [Figure 3-7](#), you type in the sender, recipient, mail message, and SMTP server. Then, click the Send button, and your message is sent.

Figure 3-7. The MailTest program



The program makes a socket connection to the SMTP server and sends the sequence of commands just discussed. It displays the commands and the responses that it receives.

Note

- When this program appeared in the first edition of Core Java in 1996, most SMTP servers accepted connections from anywhere, without making any checks at all. Nowadays, most servers are less permissive, and you might find it more difficult to run this program. The mail server of your Internet service provider may be accessible when you connect from your home, from a trusted IP address. Other servers use the "POP before SMTP" rule, requiring that you first download your e-mail (which requires a password) before you send any messages. Try fetching your e-mail before you send mail with this program. An extension to SMTP that requires an encrypted password (<http://tools.ietf.org/html/rfc2554>) is becoming more common. Our simple program does not support that authentication mechanism.

In this last section, you saw how to use socket-level programming to connect to an SMTP server and send an e-mail message. It is nice to know that this can be done and to get a glimpse of what goes on "under the hood" of an Internet service such as e-mail. However, if you are planning an application that incorporates e-mail, you will probably want to work at a higher level and use a library that encapsulates the protocol details. For example, Sun Microsystems has developed the JavaMail API as a standard extension of the Java platform. In the JavaMail API, you simply issue a call such as

```
Transport.send(message);
```

to send a message. The library takes care of message protocols, authentication, handling attachments, and so on.

Listing 3-6. MailTest.java

Code View:

```

1. import java.awt.*;
2. import java.awt.event.*;
3. import java.util.*;
4. import java.net.*;
5. import java.io.*;
6. import javax.swing.*;
7.
8. /**
9. * This program shows how to use sockets to send plain text mail messages.
10. * @author Cay Horstmann
11. * @version 1.11 2007-06-25
12. */
13. public class MailTest
14. {
15.     public static void main(String[] args)
16.     {
17.         EventQueue.invokeLater(new Runnable()
18.         {
19.             public void run()
20.             {
21.                 JFrame frame = new MailTestFrame();
22.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

```

```
23.         frame.setVisible(true);
24.     }
25.   });
26. }
28.
29. /**
30. * The frame for the mail GUI.
31. */
32. class MailTestFrame extends JFrame
33. {
34.     public MailTestFrame()
35.     {
36.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
37.         setTitle("MailTest");
38.
39.         setLayout(new GridBagLayout());
40.
41.         // we use the GBC convenience class of Core Java Volume I, Chapter 9
42.         add(new JLabel("From:"), new GBC(0, 0).setFill(GBC.HORIZONTAL));
43.
44.         from = new JTextField(20);
45.         add(from, new GBC(1, 0).setFill(GBC.HORIZONTAL).setWeight(100, 0));
46.
47.         add(new JLabel("To:"), new GBC(0, 1).setFill(GBC.HORIZONTAL));
48.
49.         to = new JTextField(20);
50.         add(to, new GBC(1, 1).setFill(GBC.HORIZONTAL).setWeight(100, 0));
51.
52.         add(new JLabel("SMTP server:"), new GBC(0, 2).setFill(GBC.HORIZONTAL));
53.
54.         smtpServer = new JTextField(20);
55.         add(smtpServer, new GBC(1, 2).setFill(GBC.HORIZONTAL).setWeight(100, 0));
56.
57.         message = new JTextArea();
58.         add(new JScrollPane(message), new GBC(0, 3, 2, 1).setFill(GBC.BOTH).setWeight(100, 100));
59.
60.         comm = new JTextArea();
61.         add(new JScrollPane(comm), new GBC(0, 4, 2, 1).setFill(GBC.BOTH).setWeight(100, 100));
62.
63.         JPanel buttonPanel = new JPanel();
64.         add(buttonPanel, new GBC(0, 5, 2, 1));
65.
66.         JButton sendButton = new JButton("Send");
67.         buttonPanel.add(sendButton);
68.         sendButton.addActionListener(new ActionListener()
69.         {
70.             public void actionPerformed(ActionEvent event)
71.             {
72.                 new SwingWorker<Void, Void>()
73.                 {
74.                     protected Void doInBackground() throws Exception
75.                     {
76.                         comm.setText("");
77.                         sendMail();
78.                         return null;
79.                     }
80.                 }.execute();
81.             }
82.         });
83.     }
84.
85. /**
86. * Sends the mail message that has been authored in the GUI.
87. */
88. public void sendMail()
89. {
90.     try
91.     {
92.         Socket s = new Socket(smtpServer.getText(), 25);
93.
94.         InputStream inStream = s.getInputStream();
95.         OutputStream outStream = s.getOutputStream();
96.
97.         in = new Scanner(inStream);
98.         out = new PrintWriter(outStream, true /* autoFlush */);
99.
100.        String hostName = InetAddress.getLocalHost().getHostName();
101.
102.        receive();
103.        send("HELO " + hostName);
104.        receive();
```

```
105.         send("MAIL FROM: <" + from.getText() + ">");  
106.         receive();  
107.         send("RCPT TO: <" + to.getText() + ">");  
108.         receive();  
109.         send("DATA");  
110.         receive();  
111.         send(message.getText());  
112.         send(".");  
113.         receive();  
114.         s.close();  
115.     }  
116.     catch (IOException e)  
117.     {  
118.         comm.append("Error: " + e);  
119.     }  
120. }  
121.  
122. /**
123. * Sends a string to the socket and echoes it in the comm text area.
124. * @param s the string to send.
125. */
126. public void send(String s) throws IOException
127. {
128.     comm.append(s);
129.     comm.append("\n");
130.     out.print(s.replaceAll("\n", "\r\n"));
131.     out.print("\r\n");
132.     out.flush();
133. }
134.  
135. /**
136. * Receives a string from the socket and displays it in the comm text area.
137. */
138. public void receive() throws IOException
139. {
140.     String line = in.nextLine();
141.     comm.append(line);
142.     comm.append("\n");
143. }
144.  
145. private Scanner in;
146. private PrintWriter out;
147. private JTextField from;
148. private JTextField to;
149. private JTextField smtpServer;
150. private JTextArea message;
151. private JTextArea comm;
152.  
153. public static final int DEFAULT_WIDTH = 300;
154. public static final int DEFAULT_HEIGHT = 300;
155. }
```



Making URL Connections

To access web servers in a Java program, you will want to work on a higher level than making a socket connection and issuing HTTP requests. In the following sections, we discuss the classes that the Java library provides for this purpose.

URLs and URIs

The `URL` and `URLConnection` classes encapsulate much of the complexity of retrieving information from a remote site. You can construct a `URL` object from a string:

```
URL url = new URL(urlString);
```

If you simply want to fetch the contents of the resource, then you can use the `openStream` method of the `URL` class. This method yields an `InputStream` object. Use it in the usual way, for example, to construct a `Scanner`:

```
InputStream inStream = url.openStream();
Scanner in = new Scanner(inStream);
```

The `java.net` package makes a useful distinction between URLs (uniform resource locators) and URIs (uniform resource identifiers).

A URI is a purely syntactical construct that contains the various parts of the string specifying a web resource. A URL is a special kind of URI, namely, one with sufficient information to locate a resource. Other URIs, such as

`mailto:cay@horstmann.com`

are not locators—there is no data to locate from this identifier. Such a URI is called a URN (uniform resource name).

In the Java library, the `URI` class has no methods for accessing the resource that the identifier specifies—its sole purpose is parsing. In contrast, the `URL` class can open a stream to the resource. For that reason, the `URL` class only works with schemes that the Java library knows how to handle, such as `http:`, `https:`, `ftp:`, the local file system (`file:`), and JAR files (`jar:`).

To see why parsing is not trivial, consider how complex URIs can be. For example,

```
http://maps.yahoo.com/py/maps.py?csz=Cupertino+CA
ftp://username:password@ftp.yourserver.com/pub/file.txt
```

The URI specification gives rules for the makeup of these identifiers. A URI has the syntax

`[scheme:] schemeSpecificPart [#fragment]`

Here, the `[. . .]` denotes an optional part, and the `:` and `#` are included literally in the identifier.

If the `scheme:` part is present, the URI is called absolute. Otherwise, it is called relative.

An absolute URI is opaque if the `schemeSpecificPart` does not begin with a `/` such as

`mailto:cay@horstmann.com`

All absolute nonopaque URIs and all relative URIs are hierarchical. Examples are

```
http://java.sun.com/index.html
.../java/net/Socket.html#Socket()
```

The `schemeSpecificPart` of a hierarchical URI has the structure

`[//authority] [path] [?query]`

where again `[. . .]` denotes optional parts.

For server-based URIs, the `authority` part has the form

`[user-info@]host[:port]`

The port must be an integer.

RFC 2396, which standardizes URIs, also supports a registry-based mechanism by which the `authority` has a different format, but this is not in common use.

One of the purposes of the `URI` class is to parse an identifier and break it up into its various components. You can retrieve them with the methods

```
getScheme
getSchemeSpecificPart
getAuthority
getUserInfo
getHost
getPort
```

```
getPath  
getQuery  
getFragment
```

The other purpose of the `URI` class is the handling of absolute and relative identifiers. If you have an absolute URI such as

```
http://docs.mycompany.com/api/java/net/ServerSocket.html
```

and a relative URI such as

```
../../../../java/net/Socket.html#Socket()
```

then you can combine the two into an absolute URI.

```
http://docs.mycompany.com/api/java/net/Socket.html#Socket()
```

This process is called resolving a relative URL.

The opposite process is called relativization. For example, suppose you have a base URI

```
http://docs.mycompany.com/api
```

and a URI

```
http://docs.mycompany.com/api/java/lang/String.html
```

Then the relativized URI is

```
java/lang/String.html
```

The `URI` class supports both of these operations:

```
relative = base.relativize(combined);  
combined = base.resolve(relative);
```

Using a `URLConnection` to Retrieve Information

If you want additional information about a web resource, then you should use the `URLConnection` class, which gives you much more control than the basic `URL` class.

When working with a `URLConnection` object, you must carefully schedule your steps, as follows:

1. Call the `openConnection` method of the `URL` class to obtain the `URLConnection` object:

```
URLConnection connection = url.openConnection();
```

2. Set any request properties, using the methods

```
setDoInput  
setDoOutput  
setIfModifiedSince  
setUseCaches  
setAllowUserInteraction  
setRequestProperty  
setConnectTimeout  
setReadTimeout
```

We discuss these methods later in this section and in the API notes.

3. Connect to the remote resource by calling the `connect` method.

```
connection.connect();
```

Besides making a socket connection to the server, this method also queries the server for header information.

4. After connecting to the server, you can query the header information. Two methods, `getHeaderFieldKey` and `getHeaderField`, enumerate all fields of the header. The method `getHeaderFields` gets a standard `Map` object containing the header fields. For your convenience, the following methods query standard fields:

```
getContentType  
getContentLength  
getContentEncoding  
getDate  
getExpiration  
getLastModified
```

5. Finally, you can access the resource data. Use the `getInputStream` method to obtain an input stream for reading the information. (This is the same input stream that the `openStream` method of the `URL` class returns.) The other method, `getContent`, isn't very useful in practice. The objects that are returned by standard content types such as `text/plain` and `image/gif` require classes in the `com.sun` hierarchy for processing. You could register your own content handlers, but we do not discuss that technique in this book.

Caution



Some programmers form the wrong mental image when using the `URLConnection` class, thinking that the `getInputStream` and `getOutputStream` methods are similar to those of the `Socket` class. But that isn't quite true. The `URLConnection` class does quite a bit of magic behind the scenes, in particular the handling of request and response headers. For that reason, it is important that you follow the setup steps for the connection.

Let us now look at some of the `URLConnection` methods in detail. Several methods set properties of the connection before connecting to the server. The most important ones are `setDoInput` and `setDoOutput`. By default, the connection yields an input stream for reading from the server but no output stream for writing. If you want an output stream (for example, for posting data to a web server), then you need to call

```
connection.setDoOutput(true);
```

Next, you may want to set some of the request headers. The request headers are sent together with the request command to the server. Here is an example:

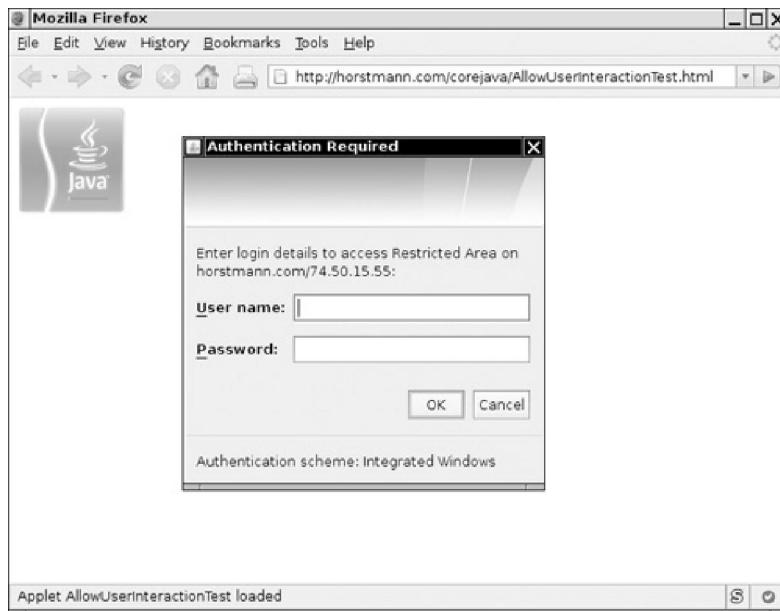
```
GET www.server.com/index.html HTTP/1.0  
Referer: http://www.somewhere.com/links.html  
Proxy-Connection: Keep-Alive  
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.8.1.4)  
Host: www.server.com  
Accept: text/html, image/gif, image/jpeg, image/png, */*  
Accept-Language: en  
Accept-Charset: iso-8859-1,*;utf-8  
Cookie: orangemilano=192218887821987
```

The `setIfModifiedSince` method tells the connection that you are only interested in data that have been modified since a certain date.

The `setUseCaches` and `setAllowUserInteraction` methods should only be called inside applets. The `setUseCaches` method directs the browser to first check the browser cache. The `setAllowUserInteraction` method allows an applet to pop up a dialog box for querying the user name and password for password-protected resources (see [Figure 3-8](#)).

Figure 3-8. A network password dialog box

[\[View full size image\]](#)



Finally, you can use the catch-all `setRequestProperty` method to set any name/value pair that is meaningful for the particular protocol. For the format of the HTTP request headers, see RFC 2616. Some of these parameters are not well documented and are passed around by word of mouth from one programmer to the next. For example, if you want to access a password-protected web page, you must do the following:

1. Concatenate the user name, a colon, and the password.

```
String input = username + ":" + password;
```

2. Compute the base64 encoding of the resulting string. (The base64 encoding encodes a sequence of bytes into a sequence of printable ASCII characters.)

```
String encoding = base64Encode(input);
```

3. Call the `setRequestProperty` method with a name of "Authorization" and value "Basic " + encoding:

```
connection.setRequestProperty("Authorization", "Basic " + encoding);
```

Tip

 You just saw how to access a password-protected web page. To access a password-protected file by FTP, you use an entirely different method. You simply construct a URL of the form

```
ftp://username:password@ftp.yourserver.com/pub/file.txt
```

Once you call the `connect` method, you can query the response header information. First, let us see how to enumerate all response header fields. The implementors of this class felt a need to express their individuality by introducing yet another iteration protocol. The call

```
String key = connection.getHeaderFieldKey(n);
```

gets the `n`th key from the response header, where `n` starts from 1! It returns `null` if `n` is zero or larger than the total number of header fields. There is no method to return the number of fields; you simply keep calling `getHeaderFieldKey` until you get `null`. Similarly, the call

```
String value = connection.getHeaderField(n);
```

returns the `n`th value.

The method `getHeaderFields` returns a `Map` of response header fields that you can access as explained in [Chapter 2](#).

```
Map<String, List<String>> headerFields = connection.getHeaderFields();
```

Here is a set of response header fields from a typical HTTP request.

```
Date: Wed, 27 Aug 2008 00:15:48 GMT
Server: Apache/2.2.2 (Unix)
Last-Modified: Sun, 22 Jun 2008 20:53:38 GMT
Accept-Ranges: bytes
Content-Length: 4813
Connection: close
Content-Type: text/html
```

As a convenience, six methods query the values of the most common header types and convert them to numeric types when appropriate. [Table 3-1](#) shows these convenience methods. The methods with return type `long` return the number of seconds since January 1, 1970 GMT.

Table 3-1. Convenience Methods for Response Header Values

Key Name	Method Name	Return Type
Date	getDate	long
Expires	getExpiration	long
Last-Modified	getLastModified	long
Content-Length	getContentLength	int
Content-Type	getContentType	String
Content-Encoding	getContentEncoding	String

The program in [Listing 3-7](#) lets you experiment with URL connections. Supply a URL and an optional user name and password on the command line when running the program, for example:

```
java URLConnectionTest http://www.yourserver.com user password
```

The program prints

- All keys and values of the header.
- The return values of the six convenience methods in [Table 3-1](#).
- The first ten lines of the requested resource.

The program is straightforward, except for the computation of the base64 encoding. There is an undocumented class, `sun.misc.BASE64Encoder`, that you can use instead of the one that we provide in the example program. Simply replace the call to `base64Encode` with

```
String encoding = new sun.misc.BASE64Encoder().encode(input.getBytes());
```

However, we supplied our own class because we do not like to rely on undocumented classes.

Note



The `javax.mail.internet.MimeUtility` class in the JavaMail standard extension package also has a method for Base64 encoding. The JDK has a class `java.util.prefs.Base64` for the same purpose, but it is not public, so you cannot use it in your code.

Listing 3-7. URLConnectionTest.java

Code View:

```
1. import java.io.*;
2. import java.net.*;
3. import java.util.*;
4.
5. /**
```

```
6.  * This program connects to a URL and displays the response header data and the first 10
7.  * lines of the requested data.
8.  *
9.  * Supply the URL and an optional username and password (for HTTP basic authentication) on
10. * the command line.
11. * @version 1.11 2007-06-26
12. * @author Cay Horstmann
13. */
14. public class URLConnectionTest
15. {
16.     public static void main(String[] args)
17.     {
18.         try
19.         {
20.             String urlName;
21.             if (args.length > 0) urlName = args[0];
22.             else urlName = "http://java.sun.com";
23.
24.             URL url = new URL(urlName);
25.            URLConnection connection = url.openConnection();
26.
27.             // set username, password if specified on command line
28.
29.             if (args.length > 2)
30.             {
31.                 String username = args[1];
32.                 String password = args[2];
33.                 String input = username + ":" + password;
34.                 String encoding = base64Encode(input);
35.                 connection.setRequestProperty("Authorization", "Basic " + encoding);
36.             }
37.
38.             connection.connect();
39.
40.             // print header fields
41.
42.             Map<String, List<String>> headers = connection.getHeaderFields();
43.             for (Map.Entry<String, List<String>> entry : headers.entrySet())
44.             {
45.                 String key = entry.getKey();
46.                 for (String value : entry.getValue())
47.                     System.out.println(key + ": " + value);
48.             }
49.
50.             // print convenience functions
51.
52.             System.out.println("-----");
53.             System.out.println("getContentType: " + connection.getContentType());
54.             System.out.println("getContentLength: " + connection.getContentLength());
55.             System.out.println("getContentEncoding: " + connection.getContentEncoding());
56.             System.out.println("getDate: " + connection.getDate());
57.             System.out.println("getExpiration: " + connection.getExpiration());
58.             System.out.println("getLastModified: " + connection.getLastModified());
59.             System.out.println("-----");
60.
61.             Scanner in = new Scanner(connection.getInputStream());
62.
63.             // print first ten lines of contents
64.
65.             for (int n = 1; in.hasNextLine() && n <= 10; n++)
66.                 System.out.println(in.nextLine());
67.                 if (in.hasNextLine()) System.out.println("... ");
68.             }
69.             catch (IOException e)
70.             {
71.                 e.printStackTrace();
72.             }
73.         }
74.         /**
75.          * Computes the Base64 encoding of a string
76.          * @param s a string
77.          * @return the Base 64 encoding of s
78.          */
79.         public static String base64Encode(String s)
80.         {
81.             ByteArrayOutputStream bOut = new ByteArrayOutputStream();
82.             Base64OutputStream out = new Base64OutputStream(bOut);
83.             try
84.             {
85.                 out.write(s.getBytes());
86.                 out.flush();
87.             }
```

```
88.         catch (IOException e)
89.         {
90.         }
91.         return bOut.toString();
92.     }
93. }
94.
95. /**
96. * This stream filter converts a stream of bytes to their Base64 encoding.
97. *
98. * Base64 encoding encodes 3 bytes into 4 characters. |11111122|22223333|33444444| Each set
99. * of 6 bits is encoded according to the toBase64 map. If the number of input bytes is not a
100. * multiple of 3, then the last group of 4 characters is padded with one or two = signs. Each
101. * output line is at most 76 characters.
102. */
103. class Base64OutputStream extends FilterOutputStream
104. {
105. /**
106. * Constructs the stream filter
107. * @param out the stream to filter
108. */
109. public Base64OutputStream(OutputStream out)
110. {
111.     super(out);
112. }
113.
114. public void write(int c) throws IOException
115. {
116.     inbuf[i] = c;
117.     i++;
118.     if (i == 3)
119.     {
120.         super.write(toBase64[((inbuf[0] & 0xFC) >> 2)]);
121.         super.write(toBase64[((inbuf[0] & 0x03) << 4) | ((inbuf[1] & 0xF0) >> 4)]);
122.         super.write(toBase64[((inbuf[1] & 0x0F) << 2) | ((inbuf[2] & 0xC0) >> 6)]);
123.         super.write(toBase64[inbuf[2] & 0x3F]);
124.         col += 4;
125.         i = 0;
126.         if (col >= 76)
127.         {
128.             super.write('\n');
129.             col = 0;
130.         }
131.     }
132. }
133.
134. public void flush() throws IOException
135. {
136.     if (i == 1)
137.     {
138.         super.write(toBase64[((inbuf[0] & 0xFC) >> 2)]);
139.         super.write(toBase64[((inbuf[0] & 0x03) << 4)]);
140.         super.write('=');
141.         super.write('=');
142.     }
143.     else if (i == 2)
144.     {
145.         super.write(toBase64[((inbuf[0] & 0xFC) >> 2)]);
146.         super.write(toBase64[((inbuf[0] & 0x03) << 4) | ((inbuf[1] & 0xF0) >> 4)]);
147.         super.write(toBase64[((inbuf[1] & 0x0F) << 2)]);
148.         super.write('=');
149.     }
150.     if (col > 0)
151.     {
152.         super.write('\n');
153.         col = 0;
154.     }
155. }
156.
157. private static char[] toBase64 = { 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K',
158.     'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'a', 'b',
159.     'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's',
160.     't', 'u', 'v', 'w', 'x', 'y', 'z', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
161.     '+', '/' };
162.
163. private int col = 0;
164. private int i = 0;
165. private int[] inbuf = new int[3];
166. }
```

Note

A commonly asked question is whether the Java platform supports access of secure web pages ([https:](https://) URLs). As of Java SE 1.4, Secure Sockets Layer (SSL) support is a part of the standard library. Before Java SE 1.4, you were only able to make SSL connections from applets by taking advantage of the SSL implementation of the browser.

**java.net.URL 1.0**

- `InputStream openStream()`
opens an input stream for reading the resource data.
- `URLConnection openConnection();`
returns a `URLConnection` object that manages the connection to the resource.

**java.netURLConnection 1.0**

- `void setDoInput(boolean doInput)`
If `doInput` is `true`, then the user can receive input from this `URLConnection`.
- `boolean getDoInput()`
If `doOutput` is `true`, then the user can send output to this `URLConnection`.
- `void setDoOutput(boolean doOutput)`
If `doOutput` is `true`, then the user can send output to this `URLConnection`.
- `boolean getDoOutput()`
If `doOutput` is `true`, then the user can send output to this `URLConnection`.
- `void setIfModifiedSince(long time)`
The `ifModifiedSince` property configures this `URLConnection` to fetch only data that have been modified since a given time. The time is given in seconds from midnight, GMT, January 1, 1970.
- `long getIfModifiedSince()`

The `ifModifiedSince` property configures this `URLConnection` to fetch only data that have been modified since a given time. The time is given in seconds from midnight, GMT, January 1, 1970.

- `void setUseCaches(boolean useCaches)`
If `useCaches` is `true`, then data can be retrieved from a local cache. Note that the `URLConnection` itself does not maintain such a cache. The cache must be supplied by an external program such as a browser.
- `boolean getUseCaches()`
- `void setAllowUserInteraction(boolean allowUserInteraction)`
If `allowUserInteraction` is `true`, then the user can be queried for passwords. Note that the `URLConnection` itself has no facilities for executing such a query. The query must be carried out by an external program such as a browser or browser plug-in.
- `boolean getAllowsUserInteraction()`
- `void setConnectTimeout(int timeout) 5.0`

- `int getConnectTimeout() 5.0`

sets or gets the timeout for the connection (in milliseconds). If the timeout has elapsed before a connection was established, the `connect` method of the associated input stream throws a `SocketTimeoutException`.

- `void setReadTimeout(int timeout) 5.0`

- `int getReadTimeout() 5.0`

sets the timeout for reading data (in milliseconds). If the timeout has elapsed before a read operation was successful, the `read` method throws a `SocketTimeoutException`.

- `void setRequestProperty(String key, String value)`

sets a request header field.

- `Map<String, List<String>> getRequestProperties() 1.4`

returns a map of request properties. All values for the same key are placed in a list.

- `void connect()`

connects to the remote resource and retrieves response header information.

- `Map<String, List<String>> Map getHeaderFields() 1.4`

returns a map of response headers. All values for the same key are placed in a map.

- `String getHeaderFieldKey(int n)`

gets the key for the `n`th response header field, or `null` if `n` is ≤ 0 or larger than the number of response header fields.

- `String getHeaderField(int n)`

gets value of the `n`th response header field, or `null` if `n` is ≤ 0 or larger than the number of response header fields.

- `int getContentLength()`

gets the content length if available, or `-1` if unknown.

- `String getContentType`

gets the content type, such as `text/plain` or `image/gif`.

- `String getContentEncoding()`

gets the content encoding, such as `gzip`. This value is not commonly used, because the default `identity` encoding is not supposed to be specified with a `Content-Encoding` header.

- `long getDate()`

- `long getExpiration()`

- `long getLastModified()`

gets the date of creation, expiration, and last modification of the resource. The dates are specified as seconds from midnight, GMT, January 1, 1970.

- `InputStream getInputStream()`

- `OutputStream getOutputStream()`

returns a stream for reading from the resource or writing to the resource.

- `Object getContent()`

selects the appropriate content handler to read the resource data and convert it into an object. This method is not useful for reading standard types such as `text/plain` or `image/gif` unless you install your own content handler.

Posting Form Data

In the preceding section, you saw how to read data from a web server. Now we will show you how your programs can send data back to a web server and to programs that the web server invokes.

To send information from a web browser to the web server, a user fills out a form, like the one in [Figure 3-9](#).

[Figure 3-9. An HTML form](#)

[View full size image]

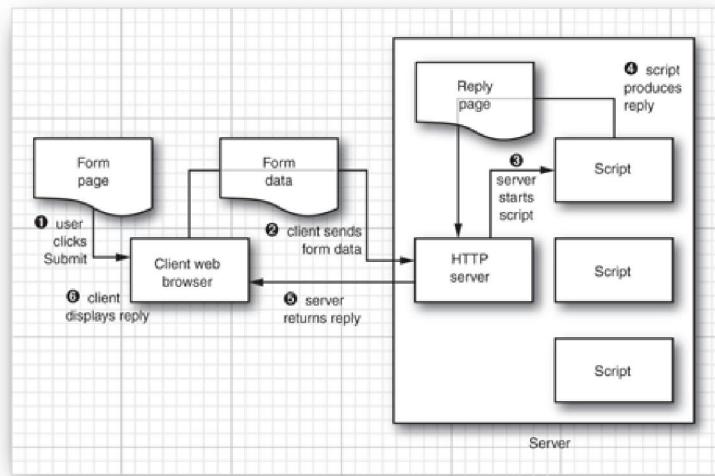
When the user clicks the Submit button, the text in the text fields and the settings of the checkboxes and radio buttons are sent back to the web server. The web server invokes a program that processes the user input.

Many technologies enable web servers to invoke programs. Among the best known ones are Java servlets, JavaServer Faces, Microsoft Active Server Pages (ASP), and Common Gateway Interface (CGI) scripts. For simplicity, we use the generic term script for a server-side program, no matter what technology is used.

The server-side script processes the form data and produces another HTML page that the web server sends back to the browser. This sequence is illustrated in [Figure 3-10](#). The response page can contain new information (for example, in an information-search program) or just an acknowledgment. The web browser then displays the response page.

[Figure 3-10. Data flow during execution of a server-side script](#)

[View full size image]



We do not discuss the implementation of server-side scripts in this book. Our interest is merely in writing client programs that interact with existing server-side scripts.

When form data are sent to a web server, it does not matter whether the data are interpreted by a servlet, a CGI script, or some other server-side technology. The client sends the data to the web server in a standard format, and the web server takes care of passing it on to the program that generates the response.

Two commands, called `GET` and `POST`, are commonly used to send information to a web server.

In the `GET` command, you simply attach parameters to the end of the URL. The URL has the form

`http://host/script?parameters`

Each parameter has the form `name=value`. Parameters are separated by `&` characters. Parameter values are encoded using the URL encoding scheme, following these rules:

- Leave the characters `A` through `Z`, `a` through `z`, `0` through `9`, and `. - * _` unchanged.
- Replace all spaces with `+` characters.
- Encode all other characters into UTF-8 and encode each byte by a `%`, followed by a two-digit hexadecimal number.

For example, to transmit the street name `S. Main`, you use `S%2e+Main`, as the hexadecimal number `2e` (or decimal `46`) is the ASCII code of the `".` character.

This encoding keeps any intermediate programs from messing with spaces and interpreting other special characters.

For example, at the time of this writing, the Yahoo! web site has a script, `py/maps.py`, at the host maps.yahoo.com. The script requires two parameters with names `addr` and `csz`. To get a map of 1 Infinite Loop, Cupertino, CA, you use the following URL:

`http://maps.yahoo.com/py/maps.py?addr=1+Infinite+Loop&csz=Cupertino+CA`

The `GET` command is simple, but it has a major limitation that makes it relatively unpopular: Most browsers have a limit on the number of characters that you can include in a `GET` request.

In the `POST` command, you do not attach parameters to a URL. Instead, you get an output stream from the `URLConnection` and write name/value pairs to the output stream. You still have to URL-encode the values and separate them with `&` characters.

Let us look at this process in more detail. To post data to a script, you first establish a `URLConnection`.

```
URL url = new URL("http://host/script");
URLConnection connection = url.openConnection();
```

Then, you call the `setDoOutput` method to set up the connection for output.

```
connection.setDoOutput(true);
```

Next, you call `getOutputStream` to get a stream through which you can send data to the server. If you are sending text to the server, it is convenient to wrap that stream into a `PrintWriter`.

```
PrintWriter out = new PrintWriter(connection.getOutputStream());
```

Now you are ready to send data to the server:

```
out.print(name1 + "=" + URLEncoder.encode(value1, "UTF-8") + "&");  
out.print(name2 + "=" + URLEncoder.encode(value2, "UTF-8"));
```

Close the output stream.

```
out.close();
```

Finally, call `getInputStream` and read the server response.

Let us run through a practical example. The web site at <http://esa.un.org/unpp/> contains a form to request population data (see [Figure 3-9](#) on page [208](#)). If you look at the HTML source, you will see the following HTML tag:

```
<form action="p2k0data.asp" method="post">
```

This tag means that the name of the script executed when the user clicks the Submit button is `p2k0data.asp` and that you need to use the `POST` command to send data to the script.

Next, you need to find out the field names that the script expects. Look at the user interface components. Each of them has a `name` attribute, for example,

```
<select name="Variable">  
<option value="12;">Population</option>  
more options . . .  
</select>
```

This tells you that the name of the field is `Variable`. This field specifies the population table type. If you specify the table type "12;", you will get a table of the total population estimates. If you look further, you will also find a field name `Location` with values such as 900 for the entire world and 404 for Kenya.

There are several other fields that need to be set. To get the population estimates of Kenya from 1950 to 2050, you construct this string:

```
Panel=1&Variable=12%3b&Location=404&Variant=2&StartYear=1950&EndYear=2050&  
DoWhat=Download+as+%2eCSV+File
```

Send the string to the URL

```
http://esa.un.org/unpp/p2k0data.asp
```

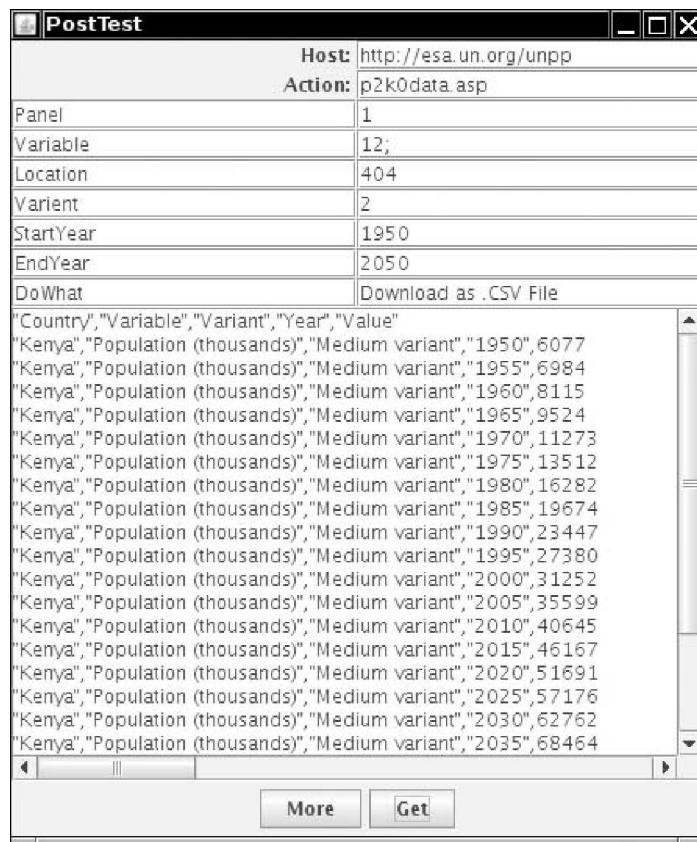
The script sends back the following reply:

```
"Country","Variable","Variant","Year","Value"  
"Kenya","Population (thousands)","Medium variant","1950",6077  
"Kenya","Population (thousands)","Medium variant","1955",6984  
"Kenya","Population (thousands)","Medium variant","1960",8115  
"Kenya","Population (thousands)","Medium variant","1965",9524  
...
```

As you can see, this particular script sends back a comma-separated data file. That is the reason we picked it as an example—it is easy to see what happens with this script, whereas it can be confusing to decipher a complex set of HTML tags that other scripts produce.

The program in [Listing 3-8](#) sends `POST` data to any script. We provide a simple GUI to set the form data and view the output (see [Figure 3-11](#)).

Figure 3-11. Harvesting information from a server



In the `doPost` method, we first open the connection, call `setDoOutput(true)`, and open the output stream. We then enumerate the names and values in a `Map` object. For each of them, we send the `name`, `=` character, `value`, and `&` separator character:

```

out.print(name);
out.print('=');
out.print(URLEncoder.encode(value, "UTF-8"));
if (more pairs) out.print('&');

```

Finally, we read the response from the server.

There is one twist with reading the response. If a script error occurs, then the call to `connection.getInputStream()` throws a `FileNotFoundException`. However, the server still sends an error page back to the browser (such as the ubiquitous "Error 404 - page not found"). To capture this error page, you cast the `URLConnection` object to the `HttpURLConnection` class and call its `getErrorStream` method:

```
InputStream err = ((HttpURLConnection) connection).getErrorStream();
```

More for curiosity's sake than for practical use, you might like to know exactly what information the `URLConnection` sends to the server in addition to the data that you supply.

The `URLConnection` object first sends a request header to the server. When posting form data, the header includes

```
Content-Type: application/x-www-form-urlencoded
```

The header for a `POST` must also include the content length, for example,

```
Content-Length: 124
```

The end of the header is indicated by a blank line. Then, the data portion follows. The web server strips off the header and routes the data portion to the server-side script.

Note that the `URLConnection` object buffers all data that you send to the output stream because it must first determine the total

content length.

The technique that this program displays is useful whenever you need to query information from an existing web site. Simply find out the parameters that you need to send (usually by inspecting the HTML source of a web page that carries out the same query), and then strip out the HTML tags and other unnecessary information from the reply.

Listing 3-8. PostTest.java

Code View:

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.io.*;
4. import java.net.*;
5. import java.util.*;
6. import javax.swing.*;
7.
8. /**
9.  * This program demonstrates how to use the URLConnection class for a POST request.
10. * @version 1.20 2007-06-25
11. * @author Cay Horstmann
12. */
13. public class PostTest
14. {
15.     public static void main(String[] args)
16.     {
17.         EventQueue.invokeLater(new Runnable()
18.         {
19.             public void run()
20.             {
21.                 JFrame frame = new PostTestFrame();
22.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23.                 frame.setVisible(true);
24.             }
25.         });
26.     }
27. }
28.
29. class PostTestFrame extends JFrame
30. {
31.     /**
32.      * Makes a POST request and returns the server response.
33.      * @param urlString the URL to post to
34.      * @param nameValuePairs a map of name/value pairs to supply in the request.
35.      * @return the server reply (either from the input stream or the error stream)
36.      */
37.     public static String doPost(String urlString, Map<String, String> nameValuePairs)
38.         throws IOException
39.     {
40.         URL url = new URL(urlString);
41.         URLConnection connection = url.openConnection();
42.         connection.setDoOutput(true);
43.
44.         PrintWriter out = new PrintWriter(connection.getOutputStream());
45.         boolean first = true;
46.         for (Map.Entry<String, String> pair : nameValuePairs.entrySet())
47.         {
48.             if (first) first = false;
49.             else out.print('&');
50.             String name = pair.getKey();
51.             String value = pair.getValue();
52.             out.print(name);
53.             out.print('=');
54.             out.print(URLEncoder.encode(value, "UTF-8"));
55.         }
56.
57.         out.close();
58.         Scanner in;
59.         StringBuilder response = new StringBuilder();
60.         try
61.         {
62.             in = new Scanner(connection.getInputStream());
63.         }
64.         catch (IOException e)
65.         {
66.             if (!(connection instanceof HttpURLConnection)) throw e;
67.             InputStream err = ((HttpURLConnection) connection).getErrorStream();
68.             if (err == null) throw e;
69.             in = new Scanner(err);
70.         }
71.     }
72. }
```

```
71.     while (in.hasNextLine())
72.     {
73.         response.append(in.nextLine());
74.         response.append("\n");
75.     }
76.
77.     in.close();
78.     return response.toString();
79. }
80.
81.
82. public PostTestFrame()
83. {
84.     setTitle("PostTest");
85.
86.     northPanel = new JPanel();
87.     add(northPanel, BorderLayout.NORTH);
88.     northPanel.setLayout(new GridLayout(0, 2));
89.     northPanel.add(new JLabel("Host: ", SwingConstants.TRAILING));
90.     final JTextField hostField = new JTextField();
91.     northPanel.add(hostField);
92.     northPanel.add(new JLabel("Action: ", SwingConstants.TRAILING));
93.     final JTextField actionField = new JTextField();
94.     northPanel.add(actionField);
95.     for (int i = 1; i <= 8; i++)
96.         northPanel.add(new JTextField());
97.
98.     final JTextArea result = new JTextArea(20, 40);
99.     add(new JScrollPane(result));
100.
101.    JPanel southPanel = new JPanel();
102.    add(southPanel, BorderLayout.SOUTH);
103.    JButton addButton = new JButton("More");
104.    southPanel.add(addButton);
105.    addButton.addActionListener(new ActionListener()
106.    {
107.        public void actionPerformed(ActionEvent event)
108.        {
109.            northPanel.add(new JTextField());
110.            northPanel.add(new JTextField());
111.            pack();
112.        }
113.    });
114.
115.    JButton getButton = new JButton("Get");
116.    southPanel.add(getButton);
117.    getButton.addActionListener(new ActionListener()
118.    {
119.        public void actionPerformed(ActionEvent event)
120.        {
121.            result.setText("");
122.            final Map<String, String> post = new HashMap<String, String>();
123.            for (int i = 4; i < northPanel.getComponentCount(); i += 2)
124.            {
125.                String name = ((JTextField) northPanel.getComponent(i)).getText();
126.                if (name.length() > 0)
127.                {
128.                    String value = ((JTextField) northPanel.getComponent(i + 1)).getText();
129.                    post.put(name, value);
130.                }
131.            }
132.            new SwingWorker<Void, Void>()
133.            {
134.                protected Void doInBackground() throws Exception
135.                {
136.                    try
137.                    {
138.                        String urlString = hostField.getText() + "/" + actionField.getText();
139.                        result.setText(doPost(urlString, post));
140.                    }
141.                    catch (IOException e)
142.                    {
143.                        result.setText("") + e;
144.                    }
145.                    return null;
146.                }
147.            }.execute();
148.        }
149.    });
150.
151.    pack();
152. }
```

```
153.  
154.     private JPanel northPanel;  
155. }
```



java.net.HttpURLConnection 1.0

- `InputStream getErrorStream()`

returns a stream from which you can read web server error messages.



java.net.URLEncoder 1.0

- `static String encode(String s, String encoding) 1.4`

returns the URL-encoded form of the string `s`, using the given character encoding scheme. (The recommended scheme is "UTF-8".) In URL encoding, the characters 'A' - 'z', 'a' - 'z', '0' - '9', '-', '_', '.', and '*' are left unchanged. Space is encoded into '+', and all other characters are encoded into sequences of encoded bytes of the form "%XY", where `0xXY` is the hexadecimal value of the byte.



java.net.URLDecoder 1.2

- `static String decode(String s, String encoding) 1.4`

returns the decoding of the URL encoded string `s` under the given character encoding scheme.

In this chapter, you have seen how to write network clients and servers in Java, and how to harvest information from web servers. The next chapter covers database connectivity. You will learn how to work with relational databases in Java, using the JDBC API. The chapter also has a brief introduction to hierarchical databases (such as LDAP directories) and the JNDI API.



Chapter 4. Database Programming

- [THE DESIGN OF JDBC](#)
- [THE STRUCTURED QUERY LANGUAGE](#)
- [JDBC CONFIGURATION](#)
- [EXECUTING SQL STATEMENTS](#)
- [QUERY EXECUTION](#)
- [SCROLLABLE AND UPDATABLE RESULT SETS](#)
- [ROW SETS](#)
- [METADATA](#)
- [TRANSACTIONS](#)
- [CONNECTION MANAGEMENT IN WEB AND ENTERPRISE APPLICATIONS](#)
- [INTRODUCTION TO LDAP](#)

In 1996, Sun released the first version of the JDBC API. This API lets programmers connect to a database and then query or update it, using the Structured Query Language (SQL). (SQL, usually pronounced "sequel," is an industry standard for relational database access.) JDBC has since become one of the most commonly used APIs in the Java library.

JDBC has been updated several times. As part of the release of Java SE 1.2 in 1998, a second version of JDBC was issued. JDBC 3 is included with Java SE 1.4 and 5.0. As this book is published, JDBC 4, the version included with Java SE 6, is the most current version.

In this chapter, we explain the key ideas behind JDBC. We introduce you to (or refresh your memory of) SQL, the industry-standard Structured Query Language for relational databases. We then provide enough details and examples to let you start using JDBC for common programming situations. The chapter close with a brief introduction to hierarchical databases, the Lightweight Directory Access Protocol (LDAP), and the Java Naming and Directory Interface (JNDI).

Note

 According to Sun, JDBC is a trademarked term and not an acronym for Java Database Connectivity. It was named to be reminiscent of ODBC, a standard database API pioneered by Microsoft and since incorporated into the SQL standard.

The Design of JDBC

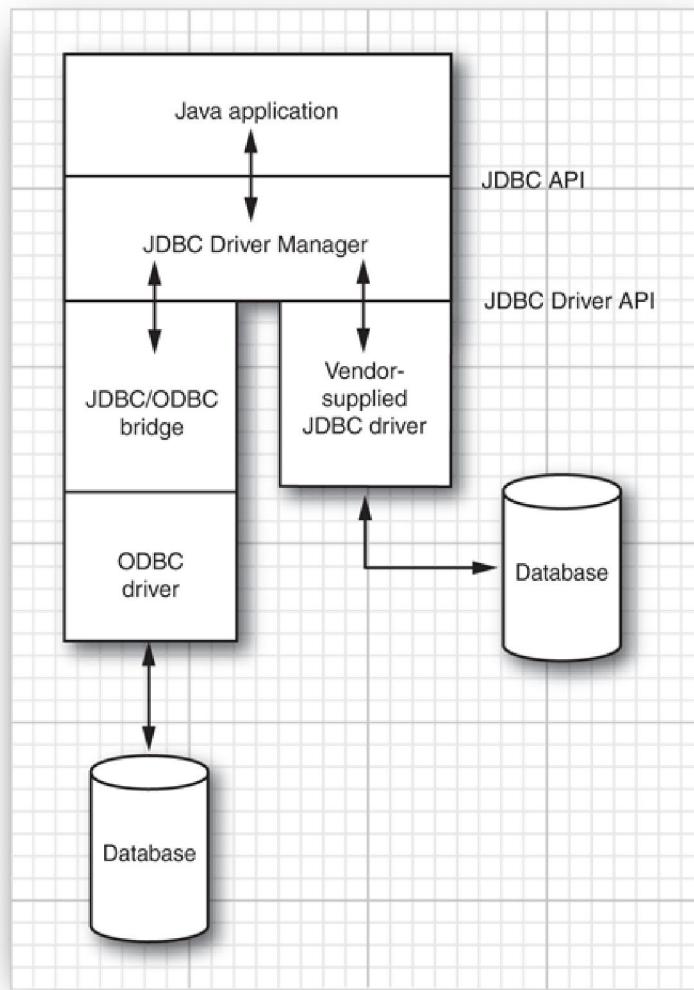
From the start, the developers of the Java technology at Sun were aware of the potential that Java showed for working with databases. In 1995, they began working on extending the standard Java library to deal with SQL access to databases. What they first hoped to do was to extend Java so that it could talk to any random database, using only "pure" Java. It didn't take them long to realize that this is an impossible task: There are simply too many databases out there, using too many protocols. Moreover, although database vendors were all in favor of Sun providing a standard network protocol for database access, they were only in favor of it if Sun decided to use their network protocol.

What all the database vendors and tool vendors did agree on was that it would be useful if Sun provided a pure Java API for SQL access along with a driver manager to allow third-party drivers to connect to specific databases. Database vendors could provide their own drivers to plug in to the driver manager. There would then be a simple mechanism for registering third-party drivers with the driver manager. As a result, two APIs were created. Application programmers use the JDBC API, and database vendors and tool providers use the JDBC Driver API.

This organization follows the very successful model of Microsoft's ODBC, which provided a C programming language interface for database access. Both JDBC and ODBC are based on the same idea: Programs written according to the API talk to the driver manager, which, in turn, uses a driver to talk to the actual database.

All this means the JDBC API is all that most programmers will ever have to deal with—see [Figure 4-1](#).

Figure 4-1. JDBC-to-database communication path



Note



A list of currently available JDBC drivers can be found at the web site
<http://developers.sun.com/product/jdbc/drivers>.

JDBC Driver Types

The JDBC specification classifies drivers into the following types:

- A type 1 driver translates JDBC to ODBC and relies on an ODBC driver to communicate with the database. Sun included one such driver, the JDBC/ODBC bridge, with earlier versions of the JDK. However, the bridge requires deployment and proper configuration of an ODBC driver. When JDBC was first released, the bridge was handy for testing, but it was never intended for production use. At this point, many better drivers are available, and we advise against using the JDBC/ODBC bridge.
- A type 2 driver is written partly in Java and partly in native code; it communicates with the client API of a database. When you use such a driver, you must install some platform-specific code onto the client in addition to a Java library.
- A type 3 driver is a pure Java client library that uses a database-independent protocol to communicate database requests to a server component, which then translates the requests into a database-specific protocol. This can simplify deployment because the platform-specific code is located only on the server.
- A type 4 driver is a pure Java library that translates JDBC requests directly to a database-specific protocol.

Most database vendors supply either a type 3 or type 4 driver with their database. Furthermore, a number of third-party

companies specialize in producing drivers with better standards conformance, support for more platforms, better performance, or, in some cases, simply better reliability than the drivers that are provided by the database vendors.

In summary, the ultimate goal of JDBC is to make possible the following:

- Programmers can write applications in the Java programming language to access any database, using standard SQL statements—or even specialized extensions of SQL—while still following Java language conventions.
- Database vendors and database tool vendors can supply the low-level drivers. Thus, they can optimize their drivers for their specific products.

Note



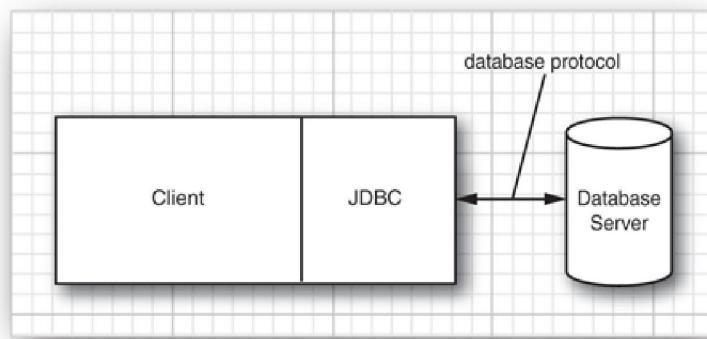
If you are curious as to why Sun just didn't adopt the ODBC model, their response, as given at the JavaOne conference in May 1996, was this:

- ODBC is hard to learn.
- ODBC has a few commands with lots of complex options. The preferred style in the Java programming language is to have simple and intuitive methods, but to have lots of them.
- ODBC relies on the use of `void*` pointers and other C features that are not natural in the Java programming language.
- An ODBC-based solution is inherently less safe and harder to deploy than a pure Java solution.

Typical Uses of JDBC

The traditional client/server model has a rich GUI on the client and a database on the server (see [Figure 4-2](#)). In this model, a JDBC driver is deployed on the client.

Figure 4-2. A traditional client/server application

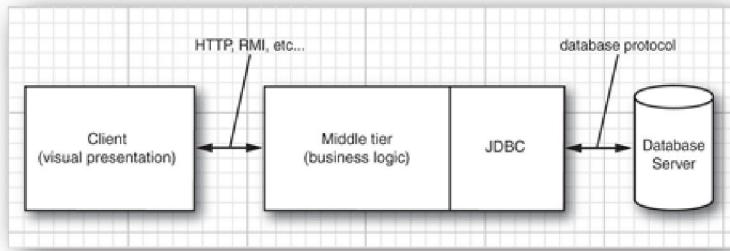


However, the world is moving away from client/server and toward a three-tier model or even more advanced n-tier models. In the three-tier model, the client does not make database calls. Instead, it calls on a middleware layer on the server that in turn makes the database queries. The three-tier model has a couple of advantages. It separates visual presentation (on the client) from the business logic (in the middle tier) and the raw data (in the database). Therefore, it becomes possible to access the same data and the same business rules from multiple clients, such as a Java application or applet or a web form.

Communication between the client and middle tier can occur through HTTP (when you use a web browser as the client) or another mechanism such as remote method invocation (RMI)—see [Chapter 10](#). JDBC manages the communication between the middle tier and the back-end database. [Figure 4-3](#) shows the basic architecture. There are, of course, many variations of this model. In particular, the Java Enterprise Edition defines a structure for application servers that manage code modules called Enterprise JavaBeans, and provides valuable services such as load balancing, request caching, security, and object-relational mapping. In that architecture, JDBC still plays an important role for issuing complex database queries. (For more information on the Enterprise Edition, see <http://java.sun.com/javaee>.)

Figure 4-3. A three-tier application

[\[View full size image\]](#)



Note

- You can use JDBC in applets and Web Start applications, but you probably don't want to. By default, the security manager permits a network connection only to the server from which the applet is downloaded. That means the web server and the database server (or the relay component of a type 3 driver) must be on the same machine, which is not a typical setup. You would need to use code signing to overcome this problem.



Chapter 4. Database Programming

- [THE DESIGN OF JDBC](#)
- [THE STRUCTURED QUERY LANGUAGE](#)
- [JDBC CONFIGURATION](#)
- [EXECUTING SQL STATEMENTS](#)
- [QUERY EXECUTION](#)
- [SCROLLABLE AND UPDATABLE RESULT SETS](#)
- [ROW SETS](#)
- [METADATA](#)
- [TRANSACTIONS](#)
- [CONNECTION MANAGEMENT IN WEB AND ENTERPRISE APPLICATIONS](#)
- [INTRODUCTION TO LDAP](#)

In 1996, Sun released the first version of the JDBC API. This API lets programmers connect to a database and then query or update it, using the Structured Query Language (SQL). (SQL, usually pronounced "sequel," is an industry standard for relational database access.) JDBC has since become one of the most commonly used APIs in the Java library.

JDBC has been updated several times. As part of the release of Java SE 1.2 in 1998, a second version of JDBC was issued. JDBC 3 is included with Java SE 1.4 and 5.0. As this book is published, JDBC 4, the version included with Java SE 6, is the most current version.

In this chapter, we explain the key ideas behind JDBC. We introduce you to (or refresh your memory of) SQL, the industry-standard Structured Query Language for relational databases. We then provide enough details and examples to let you start using JDBC for common programming situations. The chapter close with a brief introduction to hierarchical databases, the Lightweight Directory Access Protocol (LDAP), and the Java Naming and Directory Interface (JNDI).

Note

- According to Sun, JDBC is a trademarked term and not an acronym for Java Database Connectivity. It was named to be reminiscent of ODBC, a standard database API pioneered by Microsoft and since

incorporated into the SQL standard.

The Design of JDBC

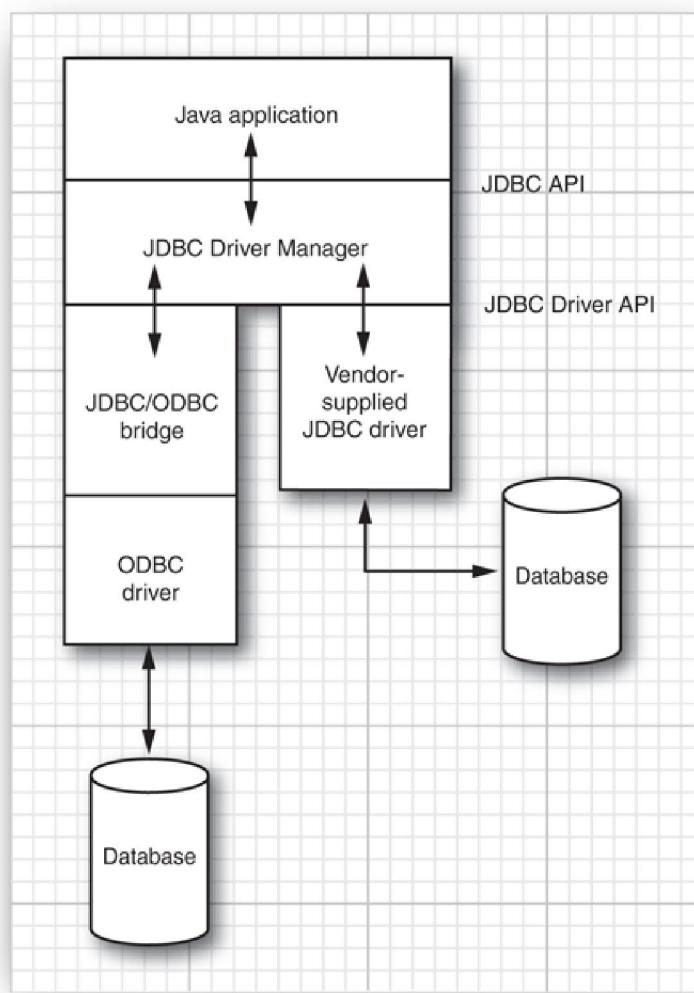
From the start, the developers of the Java technology at Sun were aware of the potential that Java showed for working with databases. In 1995, they began working on extending the standard Java library to deal with SQL access to databases. What they first hoped to do was to extend Java so that it could talk to any random database, using only "pure" Java. It didn't take them long to realize that this is an impossible task: There are simply too many databases out there, using too many protocols. Moreover, although database vendors were all in favor of Sun providing a standard network protocol for database access, they were only in favor of it if Sun decided to use their network protocol.

What all the database vendors and tool vendors did agree on was that it would be useful if Sun provided a pure Java API for SQL access along with a driver manager to allow third-party drivers to connect to specific databases. Database vendors could provide their own drivers to plug in to the driver manager. There would then be a simple mechanism for registering third-party drivers with the driver manager. As a result, two APIs were created. Application programmers use the JDBC API, and database vendors and tool providers use the JDBC Driver API.

This organization follows the very successful model of Microsoft's ODBC, which provided a C programming language interface for database access. Both JDBC and ODBC are based on the same idea: Programs written according to the API talk to the driver manager, which, in turn, uses a driver to talk to the actual database.

All this means the JDBC API is all that most programmers will ever have to deal with—see [Figure 4-1](#).

Figure 4-1. JDBC-to-database communication path



Note



A list of currently available JDBC drivers can be found at the web site
<http://developers.sun.com/product/jdbc/drivers>.

JDBC Driver Types

The JDBC specification classifies drivers into the following types:

- A type 1 driver translates JDBC to ODBC and relies on an ODBC driver to communicate with the database. Sun included one such driver, the JDBC/ODBC bridge, with earlier versions of the JDK. However, the bridge requires deployment and proper configuration of an ODBC driver. When JDBC was first released, the bridge was handy for testing, but it was never intended for production use. At this point, many better drivers are available, and we advise against using the JDBC/ODBC bridge.
- A type 2 driver is written partly in Java and partly in native code; it communicates with the client API of a database. When you use such a driver, you must install some platform-specific code onto the client in addition to a Java library.
- A type 3 driver is a pure Java client library that uses a database-independent protocol to communicate database requests to a server component, which then translates the requests into a database-specific protocol. This can simplify deployment because the platform-specific code is located only on the server.
- A type 4 driver is a pure Java library that translates JDBC requests directly to a database-specific protocol.

Most database vendors supply either a type 3 or type 4 driver with their database. Furthermore, a number of third-party companies specialize in producing drivers with better standards conformance, support for more platforms, better performance, or, in some cases, simply better reliability than the drivers that are provided by the database vendors.

In summary, the ultimate goal of JDBC is to make possible the following:

- Programmers can write applications in the Java programming language to access any database, using standard SQL statements—or even specialized extensions of SQL—while still following Java language conventions.
- Database vendors and database tool vendors can supply the low-level drivers. Thus, they can optimize their drivers for their specific products.

Note



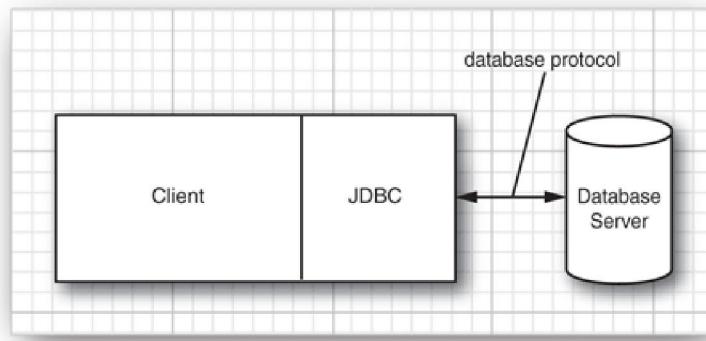
If you are curious as to why Sun just didn't adopt the ODBC model, their response, as given at the JavaOne conference in May 1996, was this:

- ODBC is hard to learn.
- ODBC has a few commands with lots of complex options. The preferred style in the Java programming language is to have simple and intuitive methods, but to have lots of them.
- ODBC relies on the use of `void*` pointers and other C features that are not natural in the Java programming language.
- An ODBC-based solution is inherently less safe and harder to deploy than a pure Java solution.

Typical Uses of JDBC

The traditional client/server model has a rich GUI on the client and a database on the server (see [Figure 4-2](#)). In this model, a JDBC driver is deployed on the client.

Figure 4-2. A traditional client/server application

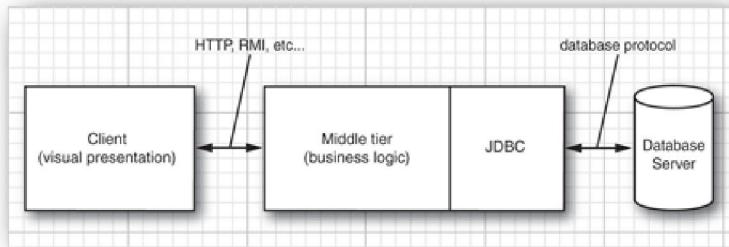


However, the world is moving away from client/server and toward a three-tier model or even more advanced n-tier models. In the three-tier model, the client does not make database calls. Instead, it calls on a middleware layer on the server that in turn makes the database queries. The three-tier model has a couple of advantages. It separates visual presentation (on the client) from the business logic (in the middle tier) and the raw data (in the database). Therefore, it becomes possible to access the same data and the same business rules from multiple clients, such as a Java application or applet or a web form.

Communication between the client and middle tier can occur through HTTP (when you use a web browser as the client) or another mechanism such as remote method invocation (RMI)—see [Chapter 10](#). JDBC manages the communication between the middle tier and the back-end database. [Figure 4-3](#) shows the basic architecture. There are, of course, many variations of this model. In particular, the Java Enterprise Edition defines a structure for application servers that manage code modules called Enterprise JavaBeans, and provides valuable services such as load balancing, request caching, security, and object-relational mapping. In that architecture, JDBC still plays an important role for issuing complex database queries. (For more information on the Enterprise Edition, see <http://java.sun.com/javaee>.)

Figure 4-3. A three-tier application

[[View full size image](#)]



Note



You can use JDBC in applets and Web Start applications, but you probably don't want to. By default, the security manager permits a network connection only to the server from which the applet is downloaded. That means the web server and the database server (or the relay component of a type 3 driver) must be on the same machine, which is not a typical setup. You would need to use code signing to overcome this problem.



The Structured Query Language

JDBC lets you communicate with databases using SQL, which is the command language for essentially all modern relational databases. Desktop databases usually have a GUI that lets users manipulate the data directly, but server-based databases are accessed purely through SQL.

The JDBC package can be thought of as nothing more than an API for communicating SQL statements to databases. We briefly introduce SQL in this section. If you have never seen SQL before, you might not find this material sufficient. If so, you should turn to one of the many books on the topic. We recommend Learning SQL by Alan Beaulieu (O'Reilly 2005) or the

opinionated classic, A Guide to the SQL Standard by C. J. Date and Hugh Darwen (Addison-Wesley 1997).

You can think of a database as a bunch of named tables with rows and columns. Each column has a column name. Each row contains a set of related data.

As the example database for this book, we use a set of database tables that describe a collection of classic computer science books (see [Table 4-1](#) through [Table 4-4](#)).

Table 4-1. The Authors Table

Author_ID	Name	Fname
ALEX	Alexander	Christopher
BROO	Brooks	Frederick P.
...

Table 4-2. The Books Table

Title	ISBN	Publisher_ID	Price
A Guide to the SQL Standard	0-201-96426-0	0201	47.95
A Pattern Language: Towns, Buildings, Construction	0-19-501919-9	019	65.00
...

Table 4-3. The BooksAuthors Table

ISBN	Author_ID	Seq_No
0-201-96426-0	DATE	1
0-201-96426-0	DARW	2
0-19-501919-9	ALEX	1
...

Table 4-4. The Publishers Table

Publisher_ID	Name	URL
0201	Addison-Wesley	www.aw-bc.com
0407	John Wiley & Sons	www.wiley.com
...

[Figure 4-4](#) shows a view of the `Books` table. [Figure 4-5](#) shows the result of joining this table with the `Publishers` table. The `Books` and the `Publishers` table each contain an identifier for the publisher. When we join both tables on the publisher code, we obtain a query result made up of values from the joined tables. Each row in the result contains the information about a book, together with the publisher name and web page URL. Note that the publisher names and URLs are duplicated across several rows because we have several rows with the same publisher.

Figure 4-4. Sample table containing books

[\[View full size image\]](#)

	Title	ISBN	Publisher_ID	Price
>	UNIX System Administration Handbook	0-13-020601-6	013	68.00
	The C Programming Language	0-13-110362-8	013	42.00
	A Pattern Language: Towns, Buildings, Construction	0-19-501919-9	019	65.00
	Introduction to Automata Theory, Languages, and Computation	0-201-44124-1	0201	105.00
	Design Patterns	0-201-63361-2	0201	54.99
	The C++ Programming Language	0-201-70073-5	0201	64.99
	The Mythical Man-Month	0-201-83565-9	0201	29.95
	Computer Graphics: Principles and Practice	0-201-84840-6	0201	79.99
	The Art of Computer Programming vol. 1	0-201-89663-4	0201	59.99
	The Art of Computer Programming vol. 2	0-201-89684-2	0201	59.99
	The Art of Computer Programming vol. 3	0-201-89665-0	0201	59.99
	A Guide to the SQL Standard	0-201-96426-0	0201	47.95
	Introduction to Algorithms	0-262-03293-7	0262	80.00
	Applied Cryptography	0-471-11709-9	0471	60.00
	JavaScript: The Definitive Guide	0-596-00048-0	0596	44.95
	The Cathedral and the Bazaar	0-596-00108-8	0596	16.95
	The Soul of a New Machine	0-679-60261-5	0679	18.95
	The Codebreakers	0-684-83130-9	07434	70.00
	Cuckoo's Egg	0-7434-1146-3	07434	13.95
	The UNIX Hater's Handbook	1-56884-203-1	0471	16.95

Record 1 of 20 [First] [Last]

Figure 4-5. Two tables joined together

[\[View full size image\]](#)

Field	Title	Publisher_ID	Price	Name	URL
Alias					
Table	Books	Books	Books	Publishers	Publishers
Sort					
Visible	<input checked="" type="checkbox"/>				
Function					
Criterion					
Or					
Pre					

The benefit of joining tables is to avoid unnecessary duplication of data in the database tables. For example, a naive database design might have had columns for the publisher name and URL right in the `Books` table. But then the database itself, and not just the query result, would have many duplicates of these entries. If a publisher's web address changed, all entries would need to be updated. Clearly, this is somewhat error prone. In the relational model, we distribute data into multiple tables such that no information is ever unnecessarily duplicated. For example, each publisher URL is contained only once in the publisher table. If the information needs to be combined, then the tables are joined.

In the figures, you can see a graphical tool to inspect and link the tables. Many vendors have tools to express queries in a simple form by connecting column names and filling information into forms. Such tools are often called query by example (QBE) tools. In contrast, a query that uses SQL is written out in text, with SQL syntax. For example,

Code View:

```
SELECT Books.Title, Books.Publisher_Id, Books.Price, Publishers.Name, Publishers.URL
FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
```

In the remainder of this section, you will learn how to write such queries. If you are already familiar with SQL, just skip this section.

By convention, SQL keywords are written in capital letters, although this is not necessary.

The `SELECT` statement is quite flexible. You can simply select all rows in the `Books` table with the following query:

```
SELECT * FROM Books
```

The `FROM` clause is required in every SQL `SELECT` statement. The `FROM` clause tells the database which tables to examine to find the data.

You can choose the columns that you want.

```
SELECT ISBN, Price, Title  
FROM Books
```

You can restrict the rows in the answer with the `WHERE` clause.

```
SELECT ISBN, Price, Title  
FROM Books  
WHERE Price <= 29.95
```

Be careful with the "equals" comparison. SQL uses = and <> rather than == or != as in the Java programming language, for equality testing.

Note

-  Some database vendors support the use of != for inequality testing. This is not standard SQL, so we recommend against such use.

The `WHERE` clause can also use pattern matching by means of the `LIKE` operator. The wildcard characters are not the usual * and ?, however. Use a % for zero or more characters and an underscore for a single character. For example,

```
SELECT ISBN, Price, Title  
FROM Books  
WHERE Title NOT LIKE '%n_x%'
```

excludes books with titles that contain words such as UNIX or Linux.

Note that strings are enclosed in single quotes, not double quotes. A single quote inside a string is denoted as a pair of single quotes. For example,

```
SELECT Title  
FROM Books  
WHERE Title LIKE '%''%'
```

reports all titles that contain a single quote.

You can select data from multiple tables.

```
SELECT * FROM Books, Publishers
```

Without a `WHERE` clause, this query is not very interesting. It lists all combinations of rows from both tables. In our case, where `Books` has 20 rows and `Publishers` has 8 rows, the result is a set of rows with 20 x 8 entries and lots of duplications. We really want to constrain the query to say that we are only interested in matching books with their publishers.

```
SELECT * FROM Books, Publishers  
WHERE Books.Publisher_Id = Publishers.Publisher_Id
```

This query result has 20 rows, one for each book, because each book has one publisher in the `Publisher` table.

Whenever you have multiple tables in a query, the same column name can occur in two different places. That happened in our example. There is a column called `Publisher_Id` in both the `Books` and the `Publishers` table. When an ambiguity would

otherwise result, you must prefix each column name with the name of the table to which it belongs, such as `Books.Publisher_Id`.

You can use SQL to change the data inside a database as well. For example, suppose you want to reduce by \$5.00 the current price of all books that have "C++" in their title.

```
UPDATE Books
SET Price = Price - 5.00
WHERE Title LIKE '%C++%'
```

Similarly, to delete all C++ books, you use a `DELETE` query.

```
DELETE FROM Books
WHERE Title LIKE '%C++%'
```

Moreover, SQL comes with built-in functions for taking averages, finding maximums and minimums in a column, and much more. A good source for this information is <http://sqlzoo.net>. (That site also contains a nifty interactive SQL tutorial.)

Typically, to insert values into a table, you use the `INSERT` statement:

```
INSERT INTO Books
VALUES ('A Guide to the SQL Standard', '0-201-96426-0', '0201', 47.95)
```

You need a separate `INSERT` statement for every row being inserted in the table.

Of course, before you can query, modify, and insert data, you must have a place to store data. Use the `CREATE TABLE` statement to make a new table. You specify the name and data type for each column. For example,

```
CREATE TABLE Books
(
    Title CHAR(60),
    ISBN CHAR(13),
    Publisher_Id CHAR(6),
    Price DECIMAL(10,2)
)
```

[Table 4-5](#) shows the most common SQL data types.

Table 4-5. Common SQL Data Types

Data Types	Description
INTEGER or INT	Typically, a 32-bit integer
SMALLINT	Typically, a 16-bit integer
NUMERIC(m,n), DECIMAL(m,n) or DEC(m,n)	Fixed-point decimal number with m total digits and n digits after the decimal point
FLOAT(n)	A floating-point number with n binary digits of precision
REAL	Typically, a 32-bit floating-point number
DOUBLE	Typically, a 64-bit floating-point number
CHARACTER(n) or CHAR(n)	Fixed-length string of length n
VARCHAR(n)	Variable-length strings of maximum length n
BOOLEAN	A Boolean value
DATE	Calendar date, implementation dependent
TIME	Time of day, implementation dependent
TIMESTAMP	Date and time of day, implementation dependent
BLOB	A binary large object
CLOB	A character large object

In this book, we do not discuss the additional clauses, such as keys and constraints, that you can use with the `CREATE TABLE` statement.



JDBC Configuration

Of course, you need a database program for which a JDBC driver is available. There are many excellent choices, such as IBM DB2, Microsoft SQL Server, MySQL, Oracle, and PostgreSQL.

You must also create a database for your experimental use. We assume you name it `COREJAVA`. Create a new database, or have your database administrator create one with the appropriate permissions. You need to be able to create, update, and drop tables in the database.

If you have never installed a client/server database before, you might find that setting up the database is somewhat complex and that diagnosing the cause for failure can be difficult. It might be best to seek expert help if your setup is not working correctly.

If this is your first experience with databases, we recommend that you use the Apache Derby database that is a part of some versions of JDK 6. (If you use a JDK that doesn't include it, download Apache Derby from <http://db.apache.org/derby>.)

Note



Sun refers to the version of Apache Derby that is included in the JDK as JavaDB. To avoid confusion, we call it Derby in this chapter.

You need to gather a number of items before you can write your first database program. The following sections cover these items.

Database URLs

When connecting to a database, you must use various database-specific parameters such as host names, port numbers, and database names.

JDBC uses a syntax similar to that of ordinary URLs to describe data sources. Here are examples of the syntax:

```
jdbc:derby://localhost:1527/COREJAVA;create=true  
jdbc:postgresql:COREJAVA
```

These JDBC URLs specify a Derby database and a PostgreSQL database named `COREJAVA`.

The general syntax is

```
jdbc:subprotocol:other stuff
```

where a subprotocol selects the specific driver for connecting to the database.

The format for the other stuff parameter depends on the subprotocol used. You will need to look up your vendor's documentation for the specific format.

Driver JAR Files

You need to obtain the JAR file in which the driver for your database is located. If you use Derby, you need the file `derbyclient.jar`. With another database, you need to locate the appropriate driver. For example, the PostgreSQL drivers are available at <http://jdbc.postgresql.org>.

Include the driver JAR file on the class path when running a program that accesses the database. (You don't need the JAR file for compiling.)

When you launch programs from the command line, simply use the command

```
java -classpath .:driverJar ProgramName
```

On Windows, use a semicolon to separate the current directory (denoted by the . character) from the driver JAR location.

Starting the Database

The database server needs to be started before you can connect to it. The details depend on your database.

With the Derby database, follow these steps:

1. Open a command shell and change to a directory that will hold the database files.
2. Locate the file `derbyrun.jar`. With some versions of the JDK, it is contained in the `jdk/db/lib` directory, with others in a separate JavaDB installation directory. We denote the directory containing `lib/derbyrun.jar` with `derby`.
3. Run the command

```
java -jar derby/lib/derbyrun.jar server start
```

4. Double-check that the database is working correctly. Create a file `ij.properties` that contains these lines:

```
ij.driver=org.apache.derby.jdbc.ClientDriver  
ij.protocol=jdbc:derby://localhost:1527/  
ij.database=COREJAVA;create=true
```

From another command shell, run Derby's interactive scripting tool (called `ij`) by executing

```
java -jar derby/lib/derbyrun.jar ij -p ij.properties
```

Now you can issue SQL commands such as

```
CREATE TABLE Greetings (Message CHAR(20));  
INSERT INTO Greetings VALUES ('Hello, World!');  
SELECT * FROM Greetings;  
DROP TABLE Greetings;
```

Note that each command must be terminated by a semicolon. To exit, type

```
EXIT;
```

5. When you are done using the database, stop the server with the command

```
java -jar derby/lib/derbyrun.jar server shutdown
```

If you use another database, you need to consult the documentation to find out how to start and stop your database server, and how to connect to it and issue SQL commands.

Registering the Driver Class

Some JDBC JAR files (such as the Derby driver that is included with Java SE 6) automatically register the driver class. In that case, you can skip the manual registration step that we describe in this section. A JAR file can automatically register the driver class if it contains a file `META-INF/services/java.sql.Driver`. You can simply unzip your driver JAR file to check.

Note



This registration mechanism uses a little-known part of the JAR specification; see <http://java.sun.com/javase/6/docs/technotes/guides/jar/jar.html#Service%20Provider>. Automatic registration is a requirement for a JDBC4-compliant driver.

If your driver JAR doesn't support automatic registration, you need to find out the name of the JDBC driver classes used by

your vendor. Typical driver names are

```
org.apache.derby.jdbc.ClientDriver  
org.postgresql.Driver
```

There are two ways to register the driver with the `DriverManager`. One way is to load the driver class in your Java program. For example,

```
Class.forName("org.postgresql.Driver"); // force loading of driver class
```

This statement causes the driver class to be loaded, thereby executing a static initializer that registers the driver.

Alternatively, you can set the `jdbc.drivers` property. You can specify the property with a command-line argument, such as

```
java -Djdbc.drivers=org.postgresql.Driver ProgramName
```

Or your application can set the system property with a call such as

```
System.setProperty("jdbc.drivers", "org.postgresql.Driver");
```

You can also supply multiple drivers; separate them with colons, such as

```
org.postgresql.Driver:org.apache.derby.jdbc.ClientDriver
```

Connecting to the Database

In your Java program, you open a database connection with code that is similar to the following example:

```
String url = "jdbc:postgresql:COREJAVA";  
String username = "dbuser";  
String password = "secret";  
Connection conn = DriverManager.getConnection(url, username, password);
```

The driver manager iterates through the registered drivers to find a driver that can use the subprotocol specified in the database URL.

The `getConnection` method returns a `Connection` object. In the following sections, you will see how to use the `Connection` object to execute SQL statements.

To connect to the database, you will need to know your database user name and password.

Note



By default, Derby lets you connect with any user name, and it does not check passwords. A separate schema is generated for each user. The default user name is `app`.

The test program in [Listing 4-1](#) puts these steps to work. It loads connection parameters from a file named `database.properties` and connects to the database. The `database.properties` file supplied with the sample code contains connection information for the Derby database. If you use a different database, you need to put your database-specific connection information into that file. Here is an example for connecting to a PostgreSQL database:

```
jdbc.drivers=org.postgresql.Driver  
jdbc.url=jdbc:postgresql:COREJAVA  
jdbc.username=dbuser  
jdbc.password=secret
```

After connecting to the database, the test program executes the following SQL statements:

```
CREATE TABLE Greetings (Message CHAR(20))  
INSERT INTO Greetings VALUES ('Hello, World!')  
SELECT * FROM Greetings
```

The result of the `SELECT` statement is printed, and you should see an output of

```
Hello, World!
```

Then the table is removed by executing the statement

```
DROP TABLE Greetings
```

To run this test, start your database and launch the program as

```
java -classpath .:driverJAR TestDB
```

Tip

 One way to debug JDBC-related problems is to enable JDBC tracing. Call the `DriverManager.setLogWriter` method to send trace messages to a `PrintWriter`. The trace output contains a detailed listing of the JDBC activity. Most JDBC driver implementations provide additional mechanisms for tracing. For example, with Derby, add a `traceFile` option to the JDBC URL, such as `jdbc:derby://localhost:1527/COREJAVA;create=true;traceFile=trace.out`.

Listing 4-1. `TestDB.java`

Code View:

```
1. import java.sql.*;
2. import java.io.*;
3. import java.util.*;
4.
5. /**
6.  * This program tests that the database and the JDBC driver are correctly configured.
7.  * @version 1.01 2004-09-24
8.  * @author Cay Horstmann
9.  */
10. class TestDB
11. {
12.     public static void main(String args[])
13.     {
14.         try
15.         {
16.             runTest();
17.         }
18.         catch (SQLException ex)
19.         {
20.             for (Throwable t : ex)
21.                 t.printStackTrace();
22.         }
23.         catch (IOException ex)
24.         {
25.             ex.printStackTrace();
26.         }
27.     }
28.
29. /**
30.  * Runs a test by creating a table, adding a value, showing the table contents, and
31.  * removing the table.
32. */
33. public static void runTest() throws SQLException, IOException
34. {
35.     Connection conn = getConnection();
36.     try
37.     {
38.         Statement stat = conn.createStatement();
39.
40.         stat.executeUpdate("CREATE TABLE Greetings (Message CHAR(20))");
41.         stat.executeUpdate("INSERT INTO Greetings VALUES ('Hello, World!')");
42.
43.         ResultSet result = stat.executeQuery("SELECT * FROM Greetings");
44.         if (result.next())
45.             System.out.println(result.getString(1));
```

```

46.         result.close();
47.         stat.executeUpdate("DROP TABLE Greetings");
48.     }
49.     finally
50.     {
51.         conn.close();
52.     }
53. }
54.
55. /**
56. * Gets a connection from the properties specified in the file database.properties
57. * @return the database connection
58. */
59. public static Connection getConnection() throws SQLException, IOException
60. {
61.     Properties props = new Properties();
62.     FileInputStream in = new FileInputStream("database.properties");
63.     props.load(in);
64.     in.close();
65.
66.     String drivers = props.getProperty("jdbc.drivers");
67.     if (drivers != null) System.setProperty("jdbc.drivers", drivers);
68.     String url = props.getProperty("jdbc.url");
69.     String username = props.getProperty("jdbc.username");
70.     String password = props.getProperty("jdbc.password");
71.
72.     return DriverManager.getConnection(url, username, password);
73. }
74. }
```



java.sql.DriverManager 1.1

- static Connection getConnection(String url, String user, String password)

establishes a connection to the given database and returns a Connection object.



Executing SQL Statements

To execute a SQL statement, you first create a Statement object. To create statement objects, use the Connection object that you obtained from the call to DriverManager.getConnection.

```
Statement stat = conn.createStatement();
```

Next, place the statement that you want to execute into a string, for example,

```
String command = "UPDATE Books"
+ " SET Price = Price - 5.00"
+ " WHERE Title NOT LIKE '%Introduction%';
```

Then call the executeUpdate method of the Statement class:

```
stat.executeUpdate(command);
```

The executeUpdate method returns a count of the rows that were affected by the SQL statement, or zero for statements that do not return a row count. For example, the call to executeUpdate in the preceding example returns the number of rows whose price was lowered by \$5.00.

The `executeUpdate` method can execute actions such as `INSERT`, `UPDATE`, and `DELETE` as well as data definition statements such as `CREATE TABLE` and `DROP TABLE`. However, you need to use the `executeQuery` method to execute `SELECT` queries. There is also a catch-all `execute` statement to execute arbitrary SQL statements. It's commonly used only for queries that a user supplies interactively.

When you execute a query, you are interested in the result. The `executeQuery` object returns an object of type `ResultSet` that you use to walk through the result one row at a time.

```
ResultSet rs = stat.executeQuery("SELECT * FROM Books")
```

The basic loop for analyzing a result set looks like this:

```
while (rs.next())
{
    look at a row of the result set
}
```

Caution



The iteration protocol of the `ResultSet` class is subtly different from the protocol of the `java.util.Iterator` interface. Here, the iterator is initialized to a position before the first row. You must call the `next` method once to move the iterator to the first row. Also, there is no `hasNext` method. You keep calling `next` until it returns `false`.

The order of the rows in a result set is completely arbitrary. Unless you specifically ordered the result with an `ORDER BY` clause, you should not attach any significance to the row order.

When inspecting an individual row, you will want to know the contents of the fields. A large number of accessor methods give you this information.

```
String isbn = rs.getString(1);
double price = rs.getDouble("Price");
```

There are accessors for various types, such as `getString` and `getDouble`. Each accessor has two forms, one that takes a numeric argument and one that takes a string argument. When you supply a numeric argument, you refer to the column with that number. For example, `rs.getString(1)` returns the value of the first column in the current row.

Caution



Unlike array indexes, database column numbers start at 1.

When you supply a string argument, you refer to the column in the result set with that name. For example, `rs.getDouble("Price")` returns the value of the column with name `Price`. Using the numeric argument is a bit more efficient, but the string arguments make the code easier to read and maintain.

Each `get` method makes reasonable type conversions when the type of the method doesn't match the type of the column. For example, the call `rs.getString("Price")` converts the floating-point value of the `Price` column to a string.



java.sql.Connection 1.1

- `Statement createStatement()`

creates a `Statement` object that can be used to execute SQL queries and updates without parameters.

- `void close()`

immediately closes the current connection and the JDBC resources that it created.

API**java.sql.Statement 1.1**

- `ResultSet executeQuery(String sqlQuery)`

executes the SQL statement given in the string and returns a `ResultSet` object to view the query result.

- `int executeUpdate(String sqlStatement)`

executes the SQL `INSERT`, `UPDATE`, or `DELETE` statement specified by the string. Also executes Data Definition Language (DDL) statements such as `CREATE TABLE`. Returns the number of rows affected, or -1 for a statement without an update count.

- `boolean execute(String sqlStatement)`

executes the SQL statement specified by the string. Multiple result sets and update counts may be produced. Returns `true` if the first result is a result set, `false` otherwise. Call `getResultSet` or `getUpdateCount` to retrieve the first result. See the section "[Multiple Results](#)" on page [253](#) for details on processing multiple results.

- `ResultSet getResultSet()`

returns the result set of the preceding query statement, or `null` if the preceding statement did not have a result set. Call this method only once per executed statement.

- `int getUpdateCount()`

returns the number of rows affected by the preceding update statement, or -1 if the preceding statement was a statement without an update count. Call this method only once per executed statement.

- `void close()`

closes this statement object and its associated result set.

- `boolean isClosed()`

returns `true` if this statement is closed.

API**java.sql.ResultSet 1.1**

- `boolean next()`

makes the current row in the result set move forward by one. Returns `false` after the last row. Note that you must call this method to advance to the first row.

- `Xxx getXxx(int columnNumber)`

- `Xxx getXxx(String columnLabel)`

(`Xxx` is a type such as `int`, `double`, `String`, `Date`, etc.)

returns the value of the column with the given column number or label, converted to the specified type. The column label is the label specified in the SQL `AS` clause or the column name if `AS` is not used.

- `int findColumn(String columnName)`

gives the column index associated with a column name.

- `void close()`
immediately closes the current result set.
- `boolean isClosed() 6`
returns `true` if this statement is closed.

Managing Connections, Statements, and Result Sets

Every `Connection` object can create one or more `Statement` objects. You can use the same `Statement` object for multiple, unrelated commands and queries. However, a statement has at most one open result set. If you issue multiple queries whose results you analyze concurrently, then you need multiple `Statement` objects.

Be forewarned, though, that at least one commonly used database (Microsoft SQL Server) has a JDBC driver that allows only one active statement at a time. Use the `getMaxStatements` method of the `DatabaseMetaData` class to find out the number of concurrently open statements that your JDBC driver supports.

This sounds restrictive, but in practice, you should probably not fuss with multiple concurrent result sets. If the result sets are related, then you should be able to issue a combined query and analyze a single result. It is much more efficient to let the database combine queries than it is for a Java program to iterate through multiple result sets.

When you are done using a `ResultSet`, `Statement`, or `Connection`, you should call the `close` method immediately. These objects use large data structures, and you don't want to wait for the garbage collector to deal with them.

The `close` method of a `Statement` object automatically closes the associated result set if the statement has an open result set. Similarly, the `close` method of the `Connection` class closes all statements of the connection.

If your connections are short-lived, you don't have to worry about closing statements and result sets. Just make absolutely sure that a connection object cannot possibly remain open by placing the `close` statement in a `finally` block:

```
try
{
    Connection conn = . . .;
    try
    {
        Statement stat = conn.createStatement();
        ResultSet result = stat.executeQuery(queryString);
        process query result
    }
    finally
    {
        conn.close();
    }
}
catch (SQLException ex)
{
    handle exception
}
```

Tip

 Use the `try/finally` block just to close the connection, and use a separate `try/catch` block to handle exceptions. Separating the `try` blocks makes your code easier to read and maintain.

Analyzing SQL Exceptions

Each `SQLException` has a chain of `SQLException` objects that is retrieved with the `getNextException` method. This exception chain is in addition to the "cause" chain of `Throwable` objects that every exception has. (See Volume I, Chapter 11 for details about Java exceptions.) One would need two nested loops to fully enumerate all these exceptions. Fortunately, Java SE 6 enhanced the `SQLException` class to implement the `Iterable<Throwable>` interface. The `iterator()` method yields an `Iterator<Throwable>` that iterates through both chains, first moving through the cause chain of the first `SQLException`, then moving on to the next `SQLException`, and so on. You can simply use an enhanced `for` loop:

```
for (Throwable t : sqlException)
{
```

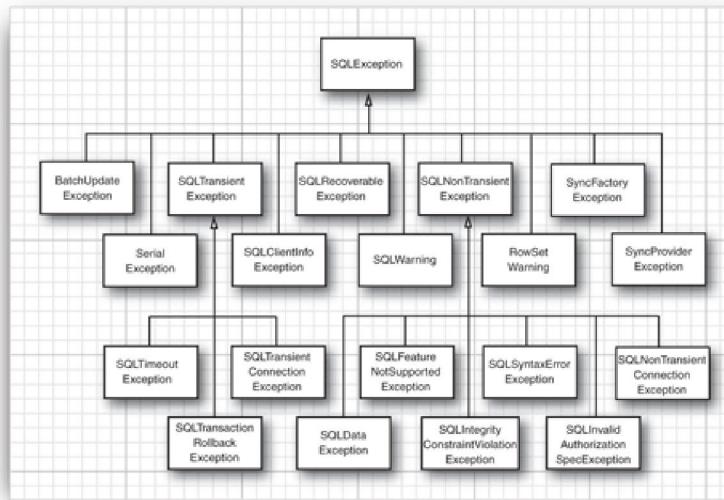
```
do something with t
}
```

You can call `getSQLState` and `getErrorCode` on an `SQLException` to analyze it further. The first method yields a string that is standardized by either X/Open or SQL:2003. (Call the `DatabaseMetaData` method `getSQLStateType` to find out which standard is used by your driver.) The error code is vendor specific.

As of Java SE 6, the SQL exceptions have been organized into an inheritance tree (shown in [Figure 4-6](#)). This allows you to catch specific error types in a vendor-independent way.

Figure 4-6. SQL exception types

[\[View full size image\]](#)



In addition, the database driver can report nonfatal conditions as warnings. You can retrieve warnings from connections, statements, and result sets. The `SQLWarning` class is a subclass of `SQLException` (even though a `SQLWarning` is not thrown as an exception). You call `getSQLState` and `getErrorCode` to get further information about the warnings. Similar to SQL exceptions, warnings are chained. To retrieve all warnings, use this loop:

```
SQLWarning w = stat.getWarning();
while (w != null)
{
    do something with w
    w = w.nextWarning();
}
```

The `DataTruncation` subclass of `SQLWarning` is used when data are read from the database and unexpectedly truncated. If data truncation happens in an update statement, a `DataTruncation` is thrown as an exception.



java.sql.SQLException 1.1

- `SQLException getNextException()`
gets the next SQL exception chained to this one, or `null` at the end of the chain.
- `Iterator<Throwable> iterator()`
gets an iterator that yields the chained SQL exceptions and their causes.
- `String getSQLState()`
gets the "SQL state," a standardized error code.
- `int getErrorCode()`

gets the vendor-specific error code.



java.sql.Warning 1.1

- SQLWarning getNextWarning()

returns the next warning chained to this one, or `null` at the end of the chain.



java.sql.Connection 1.1

java.sql.Statement 1.1

java.sql.ResultSet 1.1

- SQLWarning getWarnings()
- SQLWarning getWarnings()

returns the first of the pending warnings, or `null` if no warnings are pending.



java.sql.DataTruncation 1.1

- boolean getParameter()

returns `true` if the data truncation applies to a parameter, `false` if it applies to a column.

- int getIndex()

returns the index of the truncated parameter or column.

- int getDataSize()

returns the number of bytes that should have been transferred, or -1 if the value is unknown.

- int getTransferSize()

returns the number of bytes that were actually transferred, or -1 if the value is unknown.

Populating a Database

We now want to write our first real JDBC program. Of course, it would be nice if we could execute some of the fancy queries that we discussed earlier. Unfortunately, we have a problem: Right now, there are no data in the database. We need to populate the database, and there is a simple way of doing that: with a set of SQL instructions to create tables and insert data into them. Most database programs can process a set of SQL instructions from a text file, but there are pesky differences about statement terminators and other syntactical issues.

For that reason, we used JDBC to create a simple program that reads a file with SQL instructions, one instruction per line, and executes them.

Specifically, the program reads data from a text file in a format such as

Code View:

```
CREATE TABLE Publisher (Publisher_Id CHAR(6), Name CHAR(30), URL CHAR(80));
INSERT INTO Publishers VALUES ('0201', 'Addison-Wesley', 'www.aw-bc.com');
INSERT INTO Publishers VALUES ('0471', 'John Wiley & Sons', 'www.wiley.com');
```

Listing 4-2 contains the code for the program that reads the SQL statement file and executes the statements. It is not important that you read through the code; we merely provide the program so that you can populate your database and run the examples in the remainder of this chapter.

Make sure that your database server is running, and run the program as follows:

```
java -classpath .:driverPath ExecSQL Books.sql  
java -classpath .:driverPath ExecSQL Authors.sql  
java -classpath .:driverPath ExecSQL Publishers.sql  
java -classpath .:driverPath ExecSQL BooksAuthors.sql
```

Before running the program, check that the file `database.properties` is set up properly for your environment—see "Connecting to the Database" on page 229.

Note



Your database may also have a utility to read the SQL files directly. For example, with Derby, you can run

```
java -jar derby/lib/derbyrun.jar ij -p ij.properties Books.sql
```

(The `ij.properties` file is described in the section "Starting the Database" on page 228.)

Alternatively, if you are familiar with Ant, you can use the Ant `sql` task.

In the data format for the `ExecSQL` command, we allow an optional semicolon at the end of each line because most database utilities, as well as Ant, expect this format.

The following steps briefly describe the `ExecSQL` program:

1. Connect to the database. The `getConnection` method reads the properties in the file `database.properties` and adds the `jdbc.drivers` property to the system properties. The driver manager uses the `jdbc.drivers` property to load the appropriate database driver. The `getConnection` method uses the `jdbc.url`, `jdbc.username`, and `jdbc.password` properties to open the database connection.
2. Open the file with the SQL statements. If no file name was supplied, then prompt the user to enter the statements on the console.
3. Execute each statement with the generic `execute` method. If it returns `true`, the statement had a result set. The four SQL files that we provide for the book database all end in a `SELECT *` statement so that you can see that the data were successfully inserted.
4. If there was a result set, print out the result. Because this is a generic result set, we need to use metadata to find out how many columns the result has. For more information, see the section "Metadata" on page 263.
5. If there is any SQL exception, print the exception and any chained exceptions that may be contained in it.
6. Close the connection to the database.

Listing 4-2 shows the code for the program.

Listing 4-2. ExecSQL.java

Code View:

```
1. import java.io.*;  
2. import java.util.*;  
3. import java.sql.*;
```

```
4.
5. /**
6.  * Executes all SQL statements in a file. Call this program as <br>
7.  * java -classpath driverPath:. ExecSQL commandFile
8.  * @version 1.30 2004-08-05
9.  * @author Cay Horstmann
10. */
11. class ExecSQL
12. {
13.     public static void main(String args[])
14.     {
15.         try
16.         {
17.             Scanner in;
18.             if (args.length == 0) in = new Scanner(System.in);
19.             else in = new Scanner(new File(args[0]));
20.
21.             Connection conn = getConnection();
22.             try
23.             {
24.                 Statement stat = conn.createStatement();
25.
26.                 while (true)
27.                 {
28.                     if (args.length == 0) System.out.println("Enter command or EXIT to exit:");
29.
30.                     if (!in.hasNextLine()) return;
31.
32.                     String line = in.nextLine();
33.                     if (line.equalsIgnoreCase("EXIT")) return;
34.                     if (line.trim().endsWith(";")) // remove trailing semicolon
35.                     {
36.                         line = line.trim();
37.                         line = line.substring(0, line.length() - 1);
38.                     }
39.                     try
40.                     {
41.                         boolean hasResultSet = stat.execute(line);
42.                         if (hasResultSet) showResultSet(stat);
43.                     }
44.                     catch (SQLException ex)
45.                     {
46.                         for (Throwable e : ex)
47.                             e.printStackTrace();
48.                     }
49.                 }
50.             }
51.             finally
52.             {
53.                 conn.close();
54.             }
55.         }
56.         catch (SQLException e)
57.         {
58.             for (Throwable t : e)
59.                 t.printStackTrace();
60.         }
61.         catch (IOException e)
62.         {
63.             e.printStackTrace();
64.         }
65.     }
66.
67. /**
68.  * Gets a connection from the properties specified in the file database.properties
69.  * @return the database connection
70. */
71. public static Connection getConnection() throws SQLException, IOException
72. {
73.     Properties props = new Properties();
74.     FileInputStream in = new FileInputStream("database.properties");
75.     props.load(in);
76.     in.close();
77.
78.     String drivers = props.getProperty("jdbc.drivers");
79.     if (drivers != null) System.setProperty("jdbc.drivers", drivers);
80.
81.     String url = props.getProperty("jdbc.url");
82.     String username = props.getProperty("jdbc.username");
83.     String password = props.getProperty("jdbc.password");
```

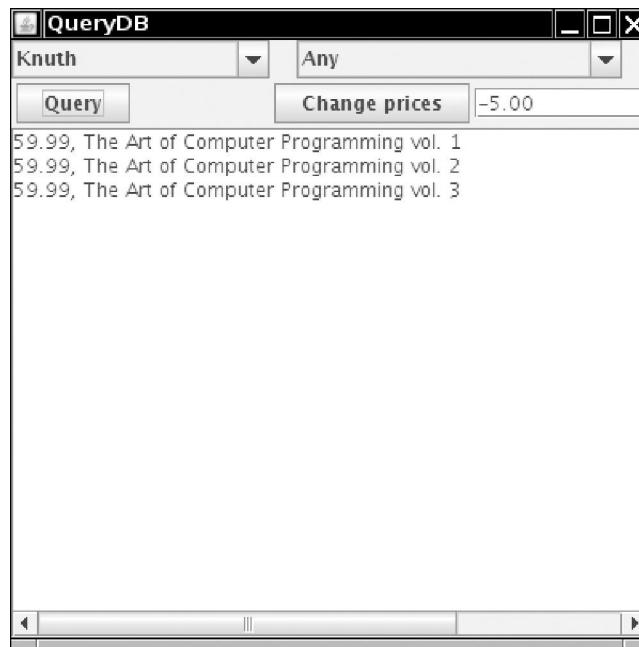
```
84.         return DriverManager.getConnection(url, username, password);
85.     }
86.
87.
88.    /**
89.     * Prints a result set.
90.     * @param stat the statement whose result set should be printed
91.     */
92.    public static void showResultSet(Statement stat) throws SQLException
93.    {
94.        ResultSet result = stat.getResultSet();
95.        ResultSetMetaData metaData = result.getMetaData();
96.        int columnCount = metaData.getColumnCount();
97.
98.        for (int i = 1; i <= columnCount; i++)
99.        {
100.            if (i > 1) System.out.print(", ");
101.            System.out.print(metaData.getColumnLabel(i));
102.        }
103.        System.out.println();
104.
105.        while (result.next())
106.        {
107.            for (int i = 1; i <= columnCount; i++)
108.            {
109.                if (i > 1) System.out.print(", ");
110.                System.out.print(result.getString(i));
111.            }
112.            System.out.println();
113.        }
114.        result.close();
115.    }
116. }
```



Query Execution

In this section, we write a program that executes queries against the `COREJAVA` database. For this program to work, you must have populated the `COREJAVA` database with tables, as described in the preceding section. [Figure 4-7](#) shows the `QueryDB` application in action.

Figure 4-7. The QueryDB application



You can select the author and the publisher or leave either of them as "Any." Click the Query button; all books matching your selection will be displayed in the text area.

You can also change the data in the database. Select a publisher and type an amount into the text box next to the Change prices button. When you click the button, all prices of that publisher are adjusted by the amount you entered, and the text area contains a message indicating how many rows were changed. However, to minimize unintended changes to the database, you can't change all prices at once. The author field is ignored when you change prices. After a price change, you might want to run a query to verify the new prices.

Prepared Statements

In this program, we use one new feature, prepared statements. Consider the query for all books by a particular publisher, independent of the author. The SQL query is

```
SELECT Books.Price, Books.Title
FROM Books, Publishers
WHERE Books.Publisher_Id = Publishers.Publisher_Id
AND Publishers.Name = the name from the list box
```

Rather than build a separate query statement every time the user launches such a query, we can prepare a query with a host variable and use it many times, each time filling in a different string for the variable. That technique benefits performance. Whenever the database executes a query, it first computes a strategy of how to efficiently execute the query. By preparing the query and reusing it, you ensure that the planning step is done only once.

Each host variable in a prepared query is indicated with a ?. If there is more than one variable, then you must keep track of the positions of the ? when setting the values. For example, our prepared query becomes

Code View:

```
String publisherQuery =
    "SELECT Books.Price, Books.Title" +
    " FROM Books, Publishers" +
    " WHERE Books.Publisher_Id = Publishers.Publisher_Id AND Publishers.Name = ?";
PreparedStatement publisherQueryStat = conn.prepareStatement(publisherQuery);
```

Before executing the prepared statement, you must bind the host variables to actual values with a `set` method. As with the `ResultSet get` methods, there are different `set` methods for the various types. Here, we want to set a string to a publisher name.

```
publisherQueryStat.setString(1, publisher);
```

The first argument is the position number of the host variable that we want to set. The position 1 denotes the first ?. The second argument is the value that we want to assign to the host variable.

If you reuse a prepared query that you have already executed, all host variables stay bound unless you change them with a `set` method or call the `clearParameters` method. That means you only need to call a `setXxx` method on those host variables that change from one query to the next.

Once all variables have been bound to values, you can execute the query

```
ResultSet rs = publisherQueryStat.executeQuery();
```

Tip



Building a query manually, by concatenating strings, is tedious and potentially dangerous. You have to worry about special characters such as quotes and, if your query involves user input, you have to guard against injection attacks. Therefore, you should use prepared statements whenever your query involves variables.

The price update feature is implemented as an `UPDATE` statement. Note that we call `executeUpdate`, not `executeQuery`, because the `UPDATE` statement does not return a result set. The return value of `executeUpdate` is the count of changed rows. We display the count in the text area.

```
int r = priceUpdateStmt.executeUpdate();
result.setText(r + " rows updated");
```

Note



A `PreparedStatement` object becomes invalid after the associated `Connection` object is closed. However, many database drivers automatically cache prepared statements. If the same query is prepared twice, the database simply reuses the query strategy. Therefore, don't worry about the overhead of calling `prepareStatement`.

The following list briefly describes the structure of the example program.

- The author and publisher text boxes are populated by running two queries that return all author and publisher names in the database.
- The listener for the Query button checks which query type is requested. If this is the first time this query type is executed, then the prepared statement variable is `null`, and the prepared statement is constructed. Then, the values are bound to the query and the query is executed.
- The queries involving authors are complex. Because a book can have multiple authors, the `BooksAuthors` table gives the correspondence between authors and books. For example, the book with ISBN 0-201-96426-0 has two authors with codes `DATE` and `DARW`. The `BooksAuthors` table has the rows

```
0-201-96426-0, DATE, 1
0-201-96426-0, DARW, 2
```

to indicate this fact. The third column lists the order of the authors. (We can't just use the position of the rows in the table. There is no fixed row ordering in a relational table.) Thus, the query has to join the `Books`, `BooksAuthors`, and `Authors` tables to compare the author name with the one selected by the user.

Code View:

```
SELECT Books.Price, Books.Title FROM Books, BooksAuthors, Authors, Publishers
WHERE Authors.Author_Id = BooksAuthors.Author_Id AND BooksAuthors.ISBN = Books.ISBN
AND Books.Publisher_Id = Publishers.Publisher_Id AND Authors.Name = ? AND Publishers.Name = ?
```

Tip



Some Java programmers avoid complex SQL statements such as this one. A surprisingly common, but

very inefficient, workaround is to write lots of Java code that iterates through multiple result sets. But the database is a lot better at executing query code than a Java program can be—that's the core competency of a database. A rule of thumb: If you can do it in SQL, don't do it in Java.

- The listener of the Change prices button executes an UPDATE statement. Note that the WHERE clause of the UPDATE statement needs the publisher code and we know only the publisher name. This problem is solved with a nested subquery.

Code View:

```
UPDATE Books
SET Price = Price + ?
WHERE Books.Publisher_Id = (SELECT Publisher_Id FROM Publishers WHERE Name = ?)
```

- We initialize the connection and statement objects in the constructor. We hang on to them for the life of the program. Just before the program exits, we trap the "window closing" event, and these objects are closed.

[Listing 4-3](#) is the complete program code.

Listing 4-3. QueryDB.java

Code View:

```
1. import java.sql.*;
2. import java.awt.*;
3. import java.awt.event.*;
4. import java.io.*;
5. import java.util.*;
6. import javax.swing.*;
7.
8. /**
9.  * This program demonstrates several complex database queries.
10. * @version 1.23 2007-06-28
11. * @author Cay Horstmann
12. */
13. public class QueryDB
14. {
15.     public static void main(String[] args)
16.     {
17.         EventQueue.invokeLater(new Runnable()
18.         {
19.             public void run()
20.             {
21.                 JFrame frame = new QueryDBFrame();
22.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23.                 frame.setVisible(true);
24.             }
25.         });
26.     }
27. }
28.
29. /**
30. * This frame displays combo boxes for query parameters, a text area for command results,
31. * and buttons to launch a query and an update.
32. */
33. class QueryDBFrame extends JFrame
34. {
35.     public QueryDBFrame()
36.     {
37.         setTitle("QueryDB");
38.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
39.         setLayout(new GridLayout());
40.
41.         authors = new JComboBox();
42.         authors.setEditable(false);
43.         authors.addItem("Any");
44.
45.         publishers = new JComboBox();
46.         publishers.setEditable(false);
47.         publishers.addItem("Any");
48.
49.         result = new JTextArea(4, 50);
```

```
50.     result.setEditable(false);
51.
52.     priceChange = new JTextField(8);
53.     priceChange.setText("-5.00");
54.
55.     try
56.     {
57.         conn = getConnection();
58.         Statement stat = conn.createStatement();
59.         String query = "SELECT Name FROM Authors";
60.         ResultSet rs = stat.executeQuery(query);
61.         while (rs.next())
62.             authors.addItem(rs.getString(1));
63.         rs.close();
64.
65.         query = "SELECT Name FROM Publishers";
66.         rs = stat.executeQuery(query);
67.         while (rs.next())
68.             publishers.addItem(rs.getString(1));
69.         rs.close();
70.         stat.close();
71.     }
72.     catch (SQLException e)
73.     {
74.         for (Throwable t : e)
75.             result.append(t.getMessage());
76.     }
77.     catch (IOException e)
78.     {
79.         result.setText("") + e);
80.     }
81.
82. // we use the GBC convenience class of Core Java Volume I, Chapter 9
83. add(authors, new GBC(0, 0, 2, 1));
84.
85. add(publishers, new GBC(2, 0, 2, 1));
86.
87. JButton queryButton = new JButton("Query");
88. queryButton.addActionListener(new ActionListener()
89. {
90.     public void actionPerformed(ActionEvent event)
91.     {
92.         executeQuery();
93.     }
94. });
95. add(queryButton, new GBC(0, 1, 1, 1).setInsets(3));
96.
97. JButton changeButton = new JButton("Change prices");
98. changeButton.addActionListener(new ActionListener()
99. {
100.     public void actionPerformed(ActionEvent event)
101.     {
102.         changePrices();
103.     }
104. });
105. add(changeButton, new GBC(2, 1, 1, 1).setInsets(3));
106.
107. add(priceChange, new GBC(3, 1, 1, 1).setFill(GBC.HORIZONTAL));
108.
109. add(new JScrollPane(result), new GBC(0, 2, 4, 1).setFill(GBC.BOTH).setWeight(100, 100));
110.
111. addWindowListener(new WindowAdapter()
112. {
113.     public void windowClosing(WindowEvent event)
114.     {
115.         try
116.         {
117.             if (conn != null) conn.close();
118.         }
119.         catch (SQLException e)
120.         {
121.             for (Throwable t : e)
122.                 t.printStackTrace();
123.         }
124.     }
125. });
126. }
127.
128. /**
129. * Executes the selected query.
```

```
130.     */
131.     private void executeQuery()
132.     {
133.         ResultSet rs = null;
134.         try
135.         {
136.             String author = (String) authors.getSelectedItem();
137.             String publisher = (String) publishers.getSelectedItem();
138.             if (!author.equals("Any") && !publisher.equals("Any"))
139.             {
140.                 if (authorPublisherQueryStmt == null) authorPublisherQueryStmt = conn
141.                     .prepareStatement(authorPublisherQuery);
142.                     authorPublisherQueryStmt.setString(1, author);
143.                     authorPublisherQueryStmt.setString(2, publisher);
144.                     rs = authorPublisherQueryStmt.executeQuery();
145.             }
146.             else if (!author.equals("Any") && publisher.equals("Any"))
147.             {
148.                 if (authorQueryStmt == null) authorQueryStmt = conn.prepareStatement(authorQuery);
149.                 authorQueryStmt.setString(1, author);
150.                 rs = authorQueryStmt.executeQuery();
151.             }
152.             else if (author.equals("Any") && !publisher.equals("Any"))
153.             {
154.                 if (publisherQueryStmt == null) publisherQueryStmt = conn
155.                     .prepareStatement(publisherQuery);
156.                     publisherQueryStmt.setString(1, publisher);
157.                     rs = publisherQueryStmt.executeQuery();
158.             }
159.             else
160.             {
161.                 if (allQueryStmt == null) allQueryStmt = conn.prepareStatement(allQuery);
162.                 rs = allQueryStmt.executeQuery();
163.             }
164.
165.             result.setText("");
166.             while (rs.next())
167.             {
168.                 result.append(rs.getString(1));
169.                 result.append(", ");
170.                 result.append(rs.getString(2));
171.                 result.append("\n");
172.             }
173.             rs.close();
174.         }
175.         catch (SQLException e)
176.         {
177.             for (Throwable t : e)
178.                 result.append(t.getMessage());
179.         }
180.     }
181.
182. /**
183. * Executes an update statement to change prices.
184. */
185. public void changePrices()
186. {
187.     String publisher = (String) publishers.getSelectedItem();
188.     if (publisher.equals("Any"))
189.     {
190.         result.setText("I am sorry, but I cannot do that.");
191.         return;
192.     }
193.     try
194.     {
195.         if (priceUpdateStmt == null) priceUpdateStmt = conn.prepareStatement(priceUpdate);
196.         priceUpdateStmt.setString(1, priceChange.getText());
197.         priceUpdateStmt.setString(2, publisher);
198.         int r = priceUpdateStmt.executeUpdate();
199.         result.setText(r + " records updated.");
200.     }
201.     catch (SQLException e)
202.     {
203.         for (Throwable t : e)
204.             result.append(t.getMessage());
205.     }
206. }
207.
208. /**
209. * Gets a connection from the properties specified in the file database.properties
```

```

210.     * @return the database connection
211.     */
212.    public static Connection getConnection() throws SQLException, IOException
213.    {
214.        Properties props = new Properties();
215.        FileInputStream in = new FileInputStream("database.properties");
216.        props.load(in);
217.        in.close();
218.
219.        String drivers = props.getProperty("jdbc.drivers");
220.        if (drivers != null) System.setProperty("jdbc.drivers", drivers);
221.        String url = props.getProperty("jdbc.url");
222.        String username = props.getProperty("jdbc.username");
223.        String password = props.getProperty("jdbc.password");
224.
225.        return DriverManager.getConnection(url, username, password);
226.    }
227.
228.    public static final int DEFAULT_WIDTH = 400;
229.    public static final int DEFAULT_HEIGHT = 400;
230.
231.    private JComboBox authors;
232.    private JComboBox publishers;
233.    private JTextField priceChange;
234.    private JTextArea result;
235.    private Connection conn;
236.    private PreparedStatement authorQueryStmt;
237.    private PreparedStatement authorPublisherQueryStmt;
238.    private PreparedStatement publisherQueryStmt;
239.    private PreparedStatement allQueryStmt;
240.    private PreparedStatement priceUpdateStmt;
241.
242.    private static final String authorPublisherQuery = "SELECT Books.Price,
243.        Books.Title FROM Books, BooksAuthors, Authors, Publishers"
244.        + " WHERE Authors.Author_Id = BooksAuthors.Author_Id AND
245.        BooksAuthors.ISBN = Books.ISBN" + " AND Books.Publisher_Id =
246.        Publishers.Publisher_Id AND Authors.Name = ?" + " AND Publishers.Name = ?";
247.
248.    private static final String authorQuery = "SELECT Books.Price, Books.Title FROM Books,
249.        BooksAuthors, Authors" + " WHERE Authors.Author_Id =
250.        BooksAuthors.Author_Id AND BooksAuthors.ISBN = Books.ISBN"
251.        + " AND Authors.Name = ?";
252.
253.    private static final String publisherQuery = "SELECT Books.Price, Books.Title FROM Books,
254.        Publishers" + " WHERE Books.Publisher_Id = Publishers.Publisher_Id
255.        AND Publishers.Name = ?";
256.
257.    private static final String allQuery = "SELECT Books.Price, Books.Title FROM Books";
258.
259.    private static final String priceUpdate = "UPDATE Books " + "SET Price = Price + ? "
260.        + " WHERE Books.Publisher_Id = (SELECT Publisher_Id FROM Publishers WHERE Name = ?)";
261. }
```



java.sql.Connection 1.1

- PreparedStatement prepareStatement(String sql)

returns a PreparedStatement object containing the precompiled statement. The string sql contains a SQL statement that can contain one or more parameter placeholders denoted by ? characters.



java.sql.PreparedStatement 1.1

- void setXXX(int n, Xxx x)

(Xxx is a type such as int, double, String, Date, etc.)

sets the value of the nth parameter to x.

- void clearParameters()

clears all current parameters in the prepared statement.

- ResultSet executeQuery()

executes a prepared SQL query and returns a ResultSet object.

- int executeUpdate()

executes the prepared SQL INSERT, UPDATE, or DELETE statement represented by the PreparedStatement object. Returns the number of rows affected, or 0 for DDL statements such as CREATE TABLE.

Reading and Writing LOBs

In addition to numbers, strings, and dates, many databases can store large objects (LOBs) such as images or other data. In SQL, binary large objects are called BLOBS, and character large objects are called CLOBS.

To read a LOB, execute a SELECT statement and then call the getBlob or getClob method on the ResultSet. You get an object of type Blob or Clob. To get the binary data from a Blob, call the getBytes or getInputStream. For example, if you have a table with book cover images, you can retrieve an image like this:

Code View:

```
PreparedStatement stat = conn.prepareStatement("SELECT Cover FROM BookCovers WHERE ISBN=?");
stat.setInt(1, isbn);
ResultSet result = stat.executeQuery();
if (result.next())
{
    Blob coverBlob = result.getBlob(1);
    Image coverImage = ImageIO.read(coverBlob.getInputStream());
}
```

Similarly, if you retrieve a Clob object, you can get character data by calling the getSubString or getCharacterStream method.

To place a LOB into a database, you call createBlob or createClob on your Connection object, get an output stream or writer to the LOB, write the data, and store the object in the database. For example, here is how you store an image:

Code View:

```
Blob coverBlob = connection.createBlob();
int offset = 0;
OutputStream out = coverBlob.setBinaryStream(offset);
ImageIO.write(coverImage, "PNG", out);
PreparedStatement stat = conn.prepareStatement("INSERT INTO Cover VALUES (?, ?)");
stat.setInt(1, isbn);
stat.setBinaryStream(2, coverBlob);
stat.executeUpdate();
```



java.sql.ResultSet 1.1

- Blob getBlob(int columnIndex) 1.2

- Blob getBlob(String columnLabel) 1.2

- Clob getClob(int columnIndex) 1.2

- Clob getClob(String columnLabel) 1.2

gets the BLOB or CLOB at the given column.



java.sql.Blob 1.2

- long length()

gets the length of this BLOB.

- byte[] getBytes(long startPosition, long length)

gets the data in the given range from this BLOB.

- InputStream getBinaryStream()

- InputStream getBinaryStream(long startPosition, long length)

returns a stream to read the data in this BLOB or the given range.

- OutputStream setBinaryStream(long startPosition) 1.4

returns an output stream for writing into this BLOB, starting at the given position.



java.sql.Clob 1.4

- long length()

gets the number of characters of this CLOB.

- String getSubString(long startPosition, long length)

gets the characters in the given range from this BLOB.

- Reader getCharacterStream()

- Reader getCharacterStream(long startPosition, long length)

returns a reader (not a stream) to read the characters in this CLOB or the given range.

- Writer setCharacterStream(long startPosition) 1.4

returns a writer (not a stream) for writing into this CLOB, starting at the given position.



java.sql.Connection 1.1

- Blob createBlob() 6

- Clob createClob() 6

creates an empty BLOB or CLOB.

SQL Escapes

The "escape" syntax supports features that are commonly supported by databases, but with database-specific syntax variations. It is the job of the JDBC driver to translate the escape syntax to the syntax of a particular database.

Escapes are provided for the following features:

- Date and time literals
- Calling scalar functions
- Calling stored procedures
- Outer joins
- The escape character in `LIKE` clauses

Date and time literals vary widely among databases. To embed a date or time literal, specify the value in ISO 8601 format (<http://www.cl.cam.ac.uk/~mgk25/iso-time.html>). The driver will then translate it into the native format. Use `d`, `t`, `ts` for `DATE`, `TIME`, or `TIMESTAMP` values:

```
{d '2008-01-24'}  
{t '23:59:59'}  
{ts '2008-01-24 23:59:59.999'}
```

A scalar function is a function that returns a single value. Many functions are widely available in databases, but with varying names. The JDBC specification provides standard names and translates them into the database-specific names. To call a function, embed the standard function name and arguments like this:

```
{fn left(?, 20)}  
{fn user()}
```

You can find a complete list of supported function names in the JDBC specification.

A stored procedure is a procedure that executes in the database, written in a database-specific language. To call a stored procedure, use the `call` escape. You need not supply parentheses if the procedure has no parameters. Use `=` to capture a return value:

```
{call PROC1(?, ?)}  
{call PROC2}  
{call ? = PROC3(?)}
```

An outer join of two tables does not require that the rows of each table match according to the join condition. For example, the query

Code View:

```
SELECT * FROM {oj Books LEFT OUTER JOIN Publishers ON Books.Publisher_Id = Publisher.Publisher_Id}
```

contains books for which `Publisher_Id` has no match in the `Publishers` table, with `NULL` values to indicate that no match exists. You would need a `RIGHT OUTER JOIN` to include publishers without matching books, or a `FULL OUTER JOIN` to return both. The escape syntax is needed because not all databases use a standard notation for these joins.

Finally, the `_` and `%` characters have special meanings in a `LIKE` clause, to match a single character or a sequence of characters. There is no standard way to use them literally. If you want to match all strings containing a `_`, use this construct:

```
... WHERE ? LIKE %!_% {escape '!'}
```

Here we define `!` as the escape character. The combination `!_` denotes a literal underscore.

Multiple Results

It is possible for a query to return multiple results. This can happen when executing a stored procedure, or with databases that also allow submission of multiple `SELECT` statements in a single query. Here is how you retrieve all result sets.

1. Use the `execute` method to execute the SQL statement.
2. Retrieve the first result or update count.
3. Repeatedly call the `getMoreResults` method to move on to the next result set. (This call automatically closes the previous result set.)
4. Finish when there are no more result sets or update counts.

The `execute` and `getMoreResults` methods return `true` if the next item in the chain is a result set. The `getUpdateCount` method returns -1 if the next item in the chain is not an update count.

The following loop traverses all results:

```
boolean done = false;
boolean isResult = stmt.execute(command);
while (!done)
{
    if (isResult)
    {
        ResultSet result = stmt.getResultSet();
        do something with result
    }
    else
    {
        int updateCount = stmt.getUpdateCount();
        if (updateCount >= 0)
            do something with updateCount
        else
            done = true;
    }
    isResult = stmt.getMoreResults();
}
```



java.sql.Statement 1.1

- `boolean getMoreResults()`

gets the next result for this statement. Returns `true` if the next result exists and is a result set.

Retrieving Autogenerated Keys

Most databases support some mechanism for auto-numbering rows in a database. Unfortunately, the mechanisms differ widely among vendors. These automatic numbers are often used as primary keys. Although JDBC doesn't offer a vendor-independent solution for generating these keys, it does provide an efficient way of retrieving them. When you insert a new row into a table and a key is automatically generated, you can retrieve it with the following code:

```
stmt.executeUpdate(insertStatement, Statement.RETURN_GENERATED_KEYS);
ResultSet rs = stmt.getGeneratedKeys();
if (rs.next())
{
    int key = rs.getInt(1);
    . . .
}
```



java.sql.Statement 1.1

- `boolean execute(String statement, int autogenerated) 1.4`

- int executeUpdate(String statement, int autogenerated) 1.4
executes the given SQL statement, as previously described. If `autogenerated` is set to `Statement.RETURN_GENERATED_KEYS` and the statement is an `INSERT` statement, the first column contains the autogenerated key.



Scorable and Updatable Result Sets

As you have seen, the `next` method of the `ResultSet` class iterates over the rows in a result set. That is certainly adequate for a program that needs to analyze the data. However, consider a visual data display that shows a table or query result (such as [Figure 4-5](#) on page [224](#)). You usually want the user to be able to move both forward and backward in the result set. In a scrollable result, you can move forward and backward through a result set and even jump to any position.

Furthermore, once users see the contents of a result set displayed, they may be tempted to edit it. In an updatable result set, you can programmatically update entries so that the database is automatically updated. We discuss these capabilities in the following sections.

Scorable Result Sets

By default, result sets are not scrollable or updatable. To obtain scrollable result sets from your queries, you must obtain a different `Statement` object with the method

```
Statement stat = conn.createStatement(type, concurrency);
```

For a prepared statement, use the call

```
PreparedStatement stat = conn.prepareStatement(command, type, concurrency);
```

The possible values of `type` and `concurrency` are listed in [Table 4-6](#) and [Table 4-7](#). You have the following choices:

- Do you want the result set to be scrollable or not? If not, use `ResultSet.TYPE_FORWARD_ONLY`.
- If the result set is scrollable, do you want it to be able to reflect changes in the database that occurred after the query that yielded it? (In our discussion, we assume the `ResultSet.TYPE_SCROLL_INSENSITIVE` setting for scrollable result sets. This assumes that the result set does not "sense" database changes that occurred after execution of the query.)
- Do you want to be able to update the database by editing the result set? (See the next section for details.)

Table 4-6. ResultSet Type Values

Value	Explanation
<code>TYPE_FORWARD_ONLY</code>	The result set is not scrollable (default).
<code>TYPE_SCROLL_INSENSITIVE</code>	The result set is scrollable but not sensitive to database changes.
<code>TYPE_SCROLL_SENSITIVE</code>	The result set is scrollable and sensitive to database changes.

Table 4-7. ResultSet Concurrency Values

Value	Explanation
<code>CONCUR_READ_ONLY</code>	The result set cannot be used to update the database (default).
<code>CONCUR_UPDATABLE</code>	The result set can be used to update the database.

For example, if you simply want to be able to scroll through a result set but you don't want to edit its data, you use:

```
Statement stat = conn.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY);
```

All result sets that are returned by method calls

```
ResultSet rs = stat.executeQuery(query)
```

are now scrollable. A scrollable result set has a cursor that indicates the current position.

Note



Not all database drivers support scrollable or updatable result sets. (The `supportsResultSetType` and `supportsResultSetConcurrency` methods of the `DatabaseMetaData` class tell you which types and concurrency modes are supported by a particular database, using a particular driver.) Even if a database supports all result set modes, a particular query might not be able to yield a result set with all the properties that you requested. (For example, the result set of a complex query might not be updatable.) In that case, the `executeQuery` method returns a `ResultSet` of lesser capabilities and adds an `SQLWarning` to the connection object. (The section "[Analyzing SQL Exceptions](#)" on page [236](#) shows how to retrieve the warning.) Alternatively, you can use the `getType` and `getConcurrency` methods of the `ResultSet` class to find out what mode a result set actually has. If you do not check the result set capabilities and issue an unsupported operation, such as `previous` on a result set that is not scrollable, then the operation throws a `SQLException`.

Scrolling is very simple. You use

```
if (rs.previous()) . . .
```

to scroll backward. The method returns `true` if the cursor is positioned on an actual row; `false` if it now is positioned before the first row.

You can move the cursor backward or forward by a number of rows with the call

```
rs.relative(n);
```

If `n` is positive, the cursor moves forward. If `n` is negative, it moves backward. If `n` is zero, the call has no effect. If you attempt to move the cursor outside the current set of rows, it is set to point either after the last row or before the first row, depending on the sign of `n`. Then, the method returns `false` and the cursor does not move. The method returns `true` if the cursor is positioned on an actual row.

Alternatively, you can set the cursor to a particular row number:

```
rs.absolute(n);
```

You get the current row number with the call

```
int currentRow = rs.getRow();
```

The first row in the result set has number 1. If the return value is 0, the cursor is not currently on a row—it is either before the first row or after the last row.

The convenience methods `first`, `last`, `beforeFirst`, and `afterLast` move the cursor to the first, to the last, before the first, or after the last position.

Finally, the methods `isFirst`, `isLast`, `isBeforeFirst`, and `isAfterLast` test whether the cursor is at one of these special positions.

Using a scrollable result set is very simple. The hard work of caching the query data is carried out behind the scenes by the database driver.

Updatable Result Sets

If you want to edit result set data and have the changes automatically reflected in the database, you create an updatable result set. Updatable result sets don't have to be scrollable, but if you present data to a user for editing, you usually want to allow scrolling as well.

To obtain updatable result sets, you create a statement as follows:

```
Statement stat = conn.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE);
```

The result sets returned by a call to `executeQuery` are then updatable.

Note



Not all queries return updatable result sets. If your query is a join that involves multiple tables, the result might not be updatable. If your query involves only a single table or if it joins multiple tables by their primary keys, you should expect the result set to be updatable. Call the `getConcurrency` method of the `ResultSet` class to find out for sure.

For example, suppose you want to raise the prices of some books, but you don't have a simple criterion for issuing an `UPDATE` statement. Then, you can iterate through all books and update prices, based on arbitrary conditions.

```
String query = "SELECT * FROM Books";
ResultSet rs = stat.executeQuery(query);
while (rs.next())
{
    if (...)
    {
        double increase = ...
        double price = rs.getDouble("Price");
        rs.updateDouble("Price", price + increase);
        rs.updateRow(); // make sure to call updateRow after updating fields
    }
}
```

There are `updateXxx` methods for all data types that correspond to SQL types, such as `updateDouble`, `updateString`, and so on. As with the `getXxx` methods, you specify the name or the number of the column. You then specify the new value for the field.

Note



If you use the `updateXxx` method whose first parameter is the column number, be aware that this is the column number in the result set. It could well be different from the column number in the database.

The `updateXxx` method changes only the row values, not the database. When you are done with the field updates in a row, you must call the `updateRow` method. That method sends all updates in the current row to the database. If you move the cursor to another row without calling `updateRow`, all updates are discarded from the row set and they are never communicated to the database. You can also call the `cancelRowUpdates` method to cancel the updates to the current row.

The preceding example shows how you modify an existing row. If you want to add a new row to the database, you first use the `moveToInsertRow` method to move the cursor to a special position, called the insert row. You build up a new row in the insert row position by issuing `updateXxx` instructions. Finally, when you are done, call the `insertRow` method to deliver the new row to the database. When you are done inserting, call `moveToCurrentRow` to move the cursor back to the position before the call to `moveToInsertRow`. Here is an example:

```
rs.moveToInsertRow();
rs.updateString("Title", title);
rs.updateString("ISBN", isbn);
rs.updateString("Publisher_Id", pubid);
rs.updateDouble("Price", price);
rs.insertRow();
rs.moveToCurrentRow();
```

Note that you cannot influence where the new data is added in the result set or the database.

If you don't specify a column value in the insert row, it is set to a SQL `NULL`. However, if the column has a `NOT NULL` constraint, an exception is thrown and the row is not inserted.

Finally, you can delete the row under the cursor.

```
rs.deleteRow();
```

The `deleteRow` method immediately removes the row from both the result set and the database.

The `updateRow`, `insertRow`, and `deleteRow` methods of the `ResultSet` class give you the same power as executing `UPDATE`, `INSERT`, and `DELETE` SQL statements. However, programmers who are accustomed to the Java programming language might find it more natural to manipulate the database contents through result sets than by constructing SQL statements.[java.sql.ResultSet 1.1](#)

Caution



If you are not careful, you can write staggeringly inefficient code with updatable result sets. It is much more efficient to execute an `UPDATE` statement than it is to make a query and iterate through the result, changing data along the way. Updatable result sets make sense for interactive programs in which a user can make arbitrary changes, but for most programmatic changes, a SQL `UPDATE` is more appropriate.

Note



JDBC 2 delivered further enhancements to result sets, such as the capability of updating a result set with the most recent data if the data have been modified by another concurrent database connection. JDBC 3 added yet another refinement, specifying the behavior of result sets when a transaction is committed. However, these advanced features are outside the scope of this introductory chapter. We refer you to the JDBC API Tutorial and Reference by Maydene Fisher, Jon Ellis, and Jonathan Bruce (Addison-Wesley 2003) and the JDBC specification documents at <http://java.sun.com/javase/technologies/database> for more information.



java.sql.Connection 1.1

- `Statement createStatement(int type, int concurrency)` 1.2
- `PreparedStatement prepareStatement(String command, int type, int concurrency)` 1.2

creates a statement or prepared statement that yields result sets with the given type and concurrency.

Parameters:	<code>command</code>	The command to prepare
	<code>type</code>	One of the constants <code>TYPE_FORWARD_ONLY</code> , <code>TYPE_SCROLL_INSENSITIVE</code> , or <code>TYPE_SCROLL_SENSITIVE</code> of the <code>ResultSet</code> interface
	<code>concurrency</code>	One of the constants <code>CONCUR_READ_ONLY</code> or <code>CONCUR_UPDATABLE</code> of the <code>ResultSet</code> interface



java.sql.ResultSet 1.1

- `int getType()` 1.2
- returns the type of this result set, one of `TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE`, or `TYPE_SCROLL_SENSITIVE`.
- `int getConcurrency()` 1.2
- returns the concurrency setting of this result set, one of `CONCUR_READ_ONLY` or `CONCUR_UPDATABLE`.

- `boolean previous() 1.2`

moves the cursor to the preceding row. Returns `true` if the cursor is positioned on a row or `false` if the cursor is positioned before the first row.

- `int getRow() 1.2`

gets the number of the current row. Rows are numbered starting with 1.

- `boolean absolute(int r) 1.2`

moves the cursor to row `r`. Returns `true` if the cursor is positioned on a row.

- `boolean relative(int d) 1.2`

moves the cursor by `d` rows. If `d` is negative, the cursor is moved backward. Returns `true` if the cursor is positioned on a row.

- `boolean first() 1.2`

- `boolean last() 1.2`

moves the cursor to the first or last row. Returns `true` if the cursor is positioned on a row.

- `void beforeFirst() 1.2`

- `void afterLast() 1.2`

moves the cursor before the first or after the last row.

- `boolean isFirst() 1.2`

- `boolean isLast() 1.2`

tests whether the cursor is at the first or last row.

- `boolean isBeforeFirst() 1.2`

- `boolean isAfterLast() 1.2`

tests whether the cursor is before the first or after the last row.

- `void moveToInsertRow() 1.2`

moves the cursor to the insert row. The insert row is a special row for inserting new data with the `updateXxx` and `insertRow` methods.

- `void moveToCurrentRow() 1.2`

moves the cursor back from the insert row to the row that it occupied when the `moveToInsertRow` method was called.

- `void insertRow() 1.2`

inserts the contents of the insert row into the database and the result set.

- `void deleteRow() 1.2`

deletes the current row from the database and the result set.

- `void updateXxx(int column, Xxx data) 1.2`

- `void updateXxx(String columnName, Xxx data) 1.2`

(Xxx is a type such as int, double, String, Date, etc.)

updates a field in the current row of the result set.

- void updateRow() 1.2

sends the current row updates to the database.

- void cancelRowUpdates() 1.2

cancels the current row updates.



java.sql.DatabaseMetaData 1.1

- boolean supportsResultSetType(int type) 1.2

returns `true` if the database can support result sets of the given type. `type` is one of the constants `TYPE_FORWARD_ONLY`, `TYPE_SCROLL_INSENSITIVE`, or `TYPE_SCROLL_SENSITIVE` of the `ResultSet` interface.

- boolean supportsResultSetConcurrency(int type, int concurrency) 1.2

returns `true` if the database can support result sets of the given combination of type and concurrency.

Parameters: type One of the constants TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE, or TYPE_SCROLL_SENSITIVE of the ResultSet interface

`concurrency` One of the constants `CONCUR_READ_ONLY` or `CONCUR_UPDATABLE` of the `ResultSet` interface



Row Sets

Scalable result sets are powerful, but they have a major drawback. You need to keep the database connection open during the entire user interaction. However, users can walk away from their computer for a long time, leaving the connection occupied. That is not good—database connections are scarce resources. In such a situation, use a row set. The `RowSet` interface extends the `ResultSet` interface, but row sets don't have to be tied to a database connection.

Row sets are also suitable if you need to move a query result to a different tier of a complex application, or to another device such as a cell phone. You would never want to move a result set—its data structures can be huge, and it is tethered to the database connection.

The `javax.sql.rowset` package provides the following interfaces that extend the `RowSet` interface:

- A `CachedRowSet` allows disconnected operation. We discuss cached row sets in the following section.
 - A `WebRowSet` is a cached row set that can be saved to an XML file. The XML file can be moved to another tier of a web application, where it is opened by another `WebRowSet` object.
 - The `FilteredRowSet` and `JoinRowSet` interfaces support lightweight operations on row sets that are equivalent to SQL `SELECT` and `JOIN` operations. These operations are carried out on the data stored in row sets, without having to make a database connection.
 - A `JdbcRowSet` is a thin wrapper around a `ResultSet`. It adds useful getters and setters from the `RowSet` interface, turning a result set into a "bean." (See [Chapter 8](#) for more information on beans.)

Sun Microsystems expects database vendors to produce efficient implementations of these interfaces. Fortunately, they also supply reference implementations so that you can use row sets even if your database vendor doesn't support them. The reference implementations are in the package `com.sun.rowset`. The class names end in `Impl`, for example, `CachedRowSetImpl`.

Cached Row Sets

A cached row set contains all data from a result set. Because `CachedRowSet` is a subinterface of the `ResultSet` interface, you can use a cached row set exactly as you would use a result set. Cached row sets confer an important benefit: You can close the connection and still use the row set. As you will see in our sample program in [Listing 4-4](#), this greatly simplifies the implementation of interactive applications. Each user command simply opens the database connection, issues a query, puts the result in a cached row set, and then closes the database connection.

It is even possible to modify the data in a cached row set. Of course, the modifications are not immediately reflected in the database. Instead, you need to make an explicit request to accept the accumulated changes. The `CachedRowSet` then reconnects to the database and issues SQL statements to write the accumulated changes.

You can populate a `CachedRowSet` from a result set:

```
ResultSet result = . . .;
CachedRowSet crs = new com.sun.rowset.CachedRowSetImpl();
    // or use an implementation from your database vendor
crs.populate(result);
conn.close(); // now ok to close the database connection
```

Alternatively, you can let the `CachedRowSet` object establish a connection automatically. Set up the database parameters:

```
crs.setURL("jdbc:derby://localhost:1527/COREJAVA");
crs.setUsername("dbuser");
crs.setPassword("secret");
```

Then set the query statement and any parameters.

```
crs.setCommand("SELECT * FROM Books WHERE PUBLISHER = ?");
crs.setString(1, publisherName);
```

Finally, populate the row set with the query result:

```
crs.execute();
```

This call establishes a database connection, issues the query, populates the row set, and disconnects.

If your query result is very large, you would not want to put it into the row set in its entirety. After all, your users will probably only look at a few of the rows. In that case, specify a page size:

```
CachedRowSet crs = . . .;
crs.setCommand(command);
crs.setPageSize(20);
...
crs.execute();
```

Now you will only get 20 rows. To get the next batch of rows, call

```
crs.nextPage();
```

You can inspect and modify the row set with the same methods you use for result sets. If you modified the row set contents, you must write it back to the database by calling

```
crs.acceptChanges(conn);
```

or

```
crs.acceptChanges();
```

The second call works only if you configured the row set with the information (such as URL, user name, and password) that is required to connect to a database.

In the section "[Updatable Result Sets](#)" on page [256](#), you saw that not all result sets are updatable. Similarly, a row set that contains the result of a complex query will not be able to write back changes to the database. You should be safe if your row set contains data from a single table.

Caution



If you populated the row set from a result set, the row set does not know the name of the table to update. You need to call `setTable` to set the table name.

Another complexity arises if data in the database have changed after you populated the row set. This is clearly a sign of trouble that could lead to inconsistent data. The reference implementation checks whether the original row set values (that is, the values before editing) are identical to the current values in the database. If so, they are replaced with the edited values. Otherwise, a `SyncProviderException` is thrown, and none of the changes are written. Other implementations may use other strategies for synchronization.



javax.sql.RowSet 1.4

- `String getURL()`
gets or sets the database URL.
- `void setURL(String url)`
gets or sets the user name for connecting to the database.
- `String getUsername()`
gets or sets the password for connecting to the database.
- `void setUsername(String username)`
gets or sets the command that is executed to populate this row set.
- `String getPassword()`
populates this row set by issuing the statement set with `setCommand`. For the driver manager to obtain a connection, the URL, user name, and password must be set.
- `void setPassword(String password)`
populates this row set by issuing the statement set with `setCommand`. This method uses the given connection and closes it.
- `void execute()`



javax.sql.rowset.CachedRowSet 5.0

- `void execute(Connection conn)`
populates this row set by issuing the statement set with `setCommand`. This method uses the given connection and closes it.
- `void populate(ResultSet result)`
populates this cached row set with the data from the given result set.

```
• String getTableName()  
• void setTableName(String tableName)  
    gets or sets the name of the table from which this cached row set was populated.  
• int getPageSize()  
• void setPageSize(int size)  
    gets or sets the page size.  
• boolean nextPage()  
• boolean previousPage()  
    loads the next or previous page of rows. Returns true if there is a next or previous page.  
• void acceptChanges()  
• void acceptChanges(Connection conn)  
    reconnects to the database and writes the changes that are the result of editing the row set. May throw a SyncProviderException if the data cannot be written back because the database data have changed.
```



Metadata

In the preceding sections, you saw how to populate, query, and update database tables. However, JDBC can give you additional information about the structure of a database and its tables. For example, you can get a list of the tables in a particular database or the column names and types of a table. This information is not useful when you are implementing a business application with a predefined database. After all, if you design the tables, you know their structure. Structural information is, however, extremely useful for programmers who write tools that work with any database.

In SQL, data that describe the database or one of its parts are called metadata (to distinguish them from the actual data stored in the database). You can get three kinds of metadata: about a database, about a result set, and about parameters of prepared statements.

To find out more about the database, you request an object of type `DatabaseMetaData` from the database connection.

```
DatabaseMetaData meta = conn.getMetaData();
```

Now you are ready to get some metadata. For example, the call

```
ResultSet mrs = meta.getTables(null, null, null, new String[] { "TABLE" });
```

returns a result set that contains information about all tables in the database. (See the API note at the end of this section for other parameters to this method.)

Each row in the result set contains information about a table in the database. The third column is the name of the table. (Again, see the API note for the other columns.) The following loop gathers all table names:

```
while (mrs.next())  
    tableNames.addItem(mrs.getString(3));
```

There is a second important use for database metadata. Databases are complex, and the SQL standard leaves plenty of room for variability. Well over 100 methods in the `DatabaseMetaData` class can inquire about the database, including calls with exotic names such as

```
meta.supportsCatalogsInPrivilegeDefinitions()
```

and

```
meta.isNullPlusNonNullIsNotNull()
```

Clearly, these are geared toward advanced users with special needs, in particular, those who need to write highly portable code that works with multiple databases.

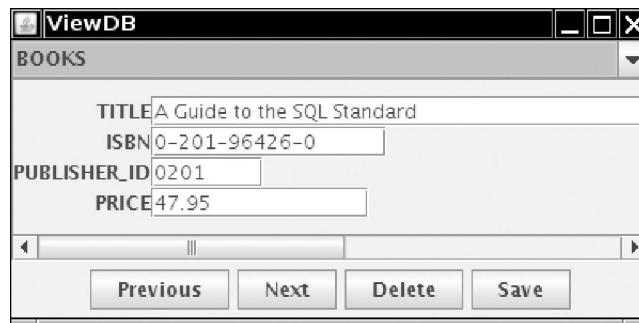
The `DatabaseMetaData` class gives data about the database. A second metadata class, `ResultSetMetaData`, reports information about a result set. Whenever you have a result set from a query, you can inquire about the number of columns and each column's name, type, and field width. Here is a typical loop:

```
ResultSet mrs = stat.executeQuery("SELECT * FROM " + tableName);
ResultSetMetaData meta = mrs.getMetaData();
for (int i = 1; i <= meta.getColumnCount(); i++)
{
    String columnName = meta.getColumnLabel(i);
    int columnWidth = meta getColumnDisplaySize(i);
    ...
}
```

In this section, we show you how to write such a simple tool. The program in [Listing 4-4](#) uses metadata to let you browse all tables in a database. The program also illustrates the use of a cached row set.

The combo box on top displays all tables in the database. Select one of them, and the center of the frame is filled with the field names of that table and the values of the first row, as shown in [Figure 4-8](#). Click Next and Previous to scroll through the rows in the table. You can also delete a row and edit the row values. Click the Save button to save the changes to the database.

Figure 4-8. The ViewDB application



Note



Many databases come with much more sophisticated tools for viewing and editing tables. If your database doesn't, check out iSQL-Viewer (<http://isql.sourceforge.net>) or SQuirreL (<http://squirrel-sql.sourceforge.net>). These programs can view the tables in any JDBC database. Our example program is not intended as a replacement for these tools, but it shows you how to implement a tool for working with arbitrary tables.

Listing 4-4. `ViewDB.java`

Code View:

```
1. import com.sun.rowset.*;
2. import java.sql.*;
3. import java.awt.*;
4. import java.awt.event.*;
5. import java.io.*;
6. import java.util.*;
7. import javax.swing.*;
8. import javax.sql.*;
9. import javax.sql.rowset.*;
10.
```

```
11. /**
12. * This program uses metadata to display arbitrary tables in a database.
13. * @version 1.31 2007-06-28
14. * @author Cay Horstmann
15. */
16. public class ViewDB
17. {
18.     public static void main(String[] args)
19.     {
20.         EventQueue.invokeLater(new Runnable()
21.         {
22.             public void run()
23.             {
24.                 JFrame frame = new ViewDBFrame();
25.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26.                 frame.setVisible(true);
27.             }
28.         });
29.     }
30. }
31.
32. /**
33. * The frame that holds the data panel and the navigation buttons.
34. */
35. class ViewDBFrame extends JFrame
36. {
37.     public ViewDBFrame()
38.     {
39.         setTitle("ViewDB");
40.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
41.
42.         tableNames = new JComboBox();
43.         tableNames.addActionListener(new ActionListener()
44.         {
45.             public void actionPerformed(ActionEvent event)
46.             {
47.                 showTable((String) tableNames.getSelectedItem());
48.             }
49.         });
50.         add(tableNames, BorderLayout.NORTH);
51.
52.         try
53.         {
54.             readDatabaseProperties();
55.             Connection conn = getConnection();
56.             try
57.             {
58.                 DatabaseMetaData meta = conn.getMetaData();
59.                 ResultSet mrs = meta.getTables(null, null, null, new String[] { "TABLE" });
60.                 while (mrs.next())
61.                     tableNames.addItem(mrs.getString(3));
62.             }
63.             finally
64.             {
65.                 conn.close();
66.             }
67.         }
68.         catch (SQLException e)
69.         {
70.             JOptionPane.showMessageDialog(this, e);
71.         }
72.         catch (IOException e)
73.         {
74.             JOptionPane.showMessageDialog(this, e);
75.         }
76.
77.         JPanel buttonPanel = new JPanel();
78.         add(buttonPanel, BorderLayout.SOUTH);
79.
80.         previousButton = new JButton("Previous");
81.         previousButton.addActionListener(new ActionListener()
82.         {
83.             public void actionPerformed(ActionEvent event)
84.             {
85.                 showPreviousRow();
86.             }
87.         });
88.         buttonPanel.add(previousButton);
89.
90.         nextButton = new JButton("Next");
```

```
91.         nextButton.addActionListener(new ActionListener()
92.         {
93.             public void actionPerformed(ActionEvent event)
94.             {
95.                 showNextRow();
96.             }
97.         });
98.     buttonPanel.add(nextButton);
99.
100.    deleteButton = new JButton("Delete");
101.    deleteButton.addActionListener(new ActionListener()
102.        {
103.            public void actionPerformed(ActionEvent event)
104.            {
105.                deleteRow();
106.            }
107.        });
108.    buttonPanel.add(deleteButton);
109.
110.    saveButton = new JButton("Save");
111.    saveButton.addActionListener(new ActionListener()
112.        {
113.            public void actionPerformed(ActionEvent event)
114.            {
115.                saveChanges();
116.            }
117.        });
118.    buttonPanel.add(saveButton);
119. }
120.
121. /**
122. * Prepares the text fields for showing a new table, and shows the first row.
123. * @param tableName the name of the table to display
124. */
125. public void showTable(String tableName)
126. {
127.     try
128.     {
129.         // open connection
130.         Connection conn = getConnection();
131.         try
132.         {
133.             // get result set
134.             Statement stat = conn.createStatement();
135.             ResultSet result = stat.executeQuery("SELECT * FROM " + tableName);
136.             // copy into cached row set
137.             crs = new CachedRowSetImpl();
138.             crs.setTableName(tableName);
139.             crs.populate(result);
140.         }
141.         finally
142.         {
143.             conn.close();
144.         }
145.
146.         if (scrollPane != null) remove(scrollPane);
147.         dataPanel = new DataPanel(crs);
148.         scrollPane = new JScrollPane(dataPanel);
149.         add(scrollPane, BorderLayout.CENTER);
150.         validate();
151.         showNextRow();
152.     }
153.     catch (SQLException e)
154.     {
155.         JOptionPane.showMessageDialog(this, e);
156.     }
157. }
158.
159. /**
160. * Moves to the previous table row.
161. */
162. public void showPreviousRow()
163. {
164.     try
165.     {
166.         if (crs == null || crs.isFirst()) return;
167.         crs.previous();
168.         dataPanel.showRow(crs);
169.     }
170.     catch (SQLException e)
```

```
171.      {
172.          for (Throwable t : e)
173.              t.printStackTrace();
174.      }
175.  }
176.
177. /**
178. * Moves to the next table row.
179. */
180. public void showNextRow()
181. {
182.     try
183.     {
184.         if (crs == null || crs.isLast()) return;
185.         crs.next();
186.         dataPanel.showRow(crs);
187.     }
188.     catch (SQLException e)
189.     {
190.         JOptionPane.showMessageDialog(this, e);
191.     }
192. }
193.
194. /**
195. * Deletes current table row.
196. */
197. public void deleteRow()
198. {
199.     try
200.     {
201.         Connection conn = getConnection();
202.         try
203.         {
204.             crs.deleteRow();
205.             crs.acceptChanges(conn);
206.             if (!crs.isLast()) crs.next();
207.             else if (!crs.isFirst()) crs.previous();
208.             else crs = null;
209.             dataPanel.showRow(crs);
210.         }
211.         finally
212.         {
213.             conn.close();
214.         }
215.     }
216.     catch (SQLException e)
217.     {
218.         JOptionPane.showMessageDialog(this, e);
219.     }
220. }
221.
222. /**
223. * Saves all changes.
224. */
225. public void saveChanges()
226. {
227.     try
228.     {
229.         Connection conn = getConnection();
230.         try
231.         {
232.             dataPanel.setRow(crs);
233.             crs.acceptChanges(conn);
234.         }
235.         finally
236.         {
237.             conn.close();
238.         }
239.     }
240.     catch (SQLException e)
241.     {
242.         JOptionPane.showMessageDialog(this, e);
243.     }
244. }
245.
246. private void readDatabaseProperties() throws IOException
247. {
248.     props = new Properties();
249.     FileInputStream in = new FileInputStream("database.properties");
250.     props.load(in);
```

```
251.     in.close();
252.     String drivers = props.getProperty("jdbc.drivers");
253.     if (drivers != null) System.setProperty("jdbc.drivers", drivers);
254. }
255.
256. /**
257. * Gets a connection from the properties specified in the file database.properties
258. * @return the database connection
259. */
260. private Connection getConnection() throws SQLException
261. {
262.     String url = props.getProperty("jdbc.url");
263.     String username = props.getProperty("jdbc.username");
264.     String password = props.getProperty("jdbc.password");
265.
266.     return DriverManager.getConnection(url, username, password);
267. }
268.
269. public static final int DEFAULT_WIDTH = 400;
270. public static final int DEFAULT_HEIGHT = 200;
271.
272. private JButton previousButton;
273. private JButton nextButton;
274. private JButton deleteButton;
275. private JButton saveButton;
276. private DataPanel dataPanel;
277. private Component scrollPane;
278. private JComboBox tableNames;
279. private Properties props;
280. private CachedRowSet crs;
281. }
282.
283. /**
284. * This panel displays the contents of a result set.
285. */
286. class DataPanel extends JPanel
287. {
288.     /**
289.      * Constructs the data panel.
290.      * @param rs the result set whose contents this panel displays
291.      */
292.     public DataPanel(ResultSet rs) throws SQLException
293.     {
294.         fields = new ArrayList<JTextField>();
295.         setLayout(new GridBagLayout());
296.         GridBagConstraints gbc = new GridBagConstraints();
297.         gbc.gridwidth = 1;
298.         gbc.gridheight = 1;
299.
300.         ResultSetMetaData rsmd = rs.getMetaData();
301.         for (int i = 1; i <= rsmd.getColumnCount(); i++)
302.         {
303.             gbc.gridx = i - 1;
304.
305.             String columnName = rsmd.getColumnName(i);
306.             gbc.gridx = 0;
307.             gbc.anchor = GridBagConstraints.EAST;
308.             add(new JLabel(columnName), gbc);
309.
310.             int columnWidth = rsmd getColumnDisplaySize(i);
311.             JTextField tb = new JTextField(columnWidth);
312.             if (!rsmd.getColumnClassName(i).equals("java.lang.String"))
313.                 tb.setEditable(false);
314.
315.             fields.add(tb);
316.
317.             gbc.gridx = 1;
318.             gbc.anchor = GridBagConstraints.WEST;
319.             add(tb, gbc);
320.         }
321.     }
322.
323. /**
324. * Shows a database row by populating all text fields with the column values.
325. */
326. public void showRow(ResultSet rs) throws SQLException
327. {
328.     for (int i = 1; i <= fields.size(); i++)
329.     {
330.         String field = rs.getString(i);
```

```

331.         JTextField tb = (JTextField) fields.get(i - 1);
332.         tb.setText(field);
333.     }
334. }
335.
336. /**
337. * Updates changed data into the current row of the row set
338. */
339. public void setRow(ResultSet rs) throws SQLException
340. {
341.     for (int i = 1; i <= fields.size(); i++)
342.     {
343.         String field = rs.getString(i);
344.         JTextField tb = (JTextField) fields.get(i - 1);
345.         if (!field.equals(tb.getText()))
346.             rs.updateString(i, tb.getText());
347.     }
348.     rs.updateRow();
349. }
350.
351. private ArrayList<JTextField> fields;
352. }
```



java.sql.Connection 1.1

- `DatabaseMetaData getMetaData()`

returns the metadata for the connection as a `DatabaseMetaData` object.



java.sql.DatabaseMetaData 1.1

- `ResultSet getTables(String catalog, String schemaPattern, String tableNamePattern, String types[])`

returns a description of all tables in a catalog that match the schema and table name patterns and the type criteria. (A schema describes a group of related tables and access permissions. A catalog describes a related group of schemas. These concepts are important for structuring large databases.)

The `catalog` and `schemaPattern` parameters can be "" to retrieve those tables without a catalog or schema, or `null` to return tables regardless of catalog or schema.

The `types` array contains the names of the table types to include. Typical types are `TABLE`, `VIEW`, `SYSTEM TABLE`, `GLOBAL TEMPORARY`, `LOCAL TEMPORARY`, `ALIAS`, and `SYNONYM`. If `types` is `null`, then tables of all types are returned.

The result set has five columns, all of which are of type `String`, as shown in [Table 4-8](#).

Table 4-8. The Result Set of the `getTables` Method

Column	Name	Explanation
1	TABLE_CAT	Table catalog (may be <code>null</code>)
2	TABLE_SCHEM	Table schema (may be <code>null</code>)
3	TABLE_NAME	Table name
4	TABLE_TYPE	Table type

5	REMARKS	Comment on the table
	<ul style="list-style-type: none">• <code>int getJDBCMajorVersion() 1.4</code>• <code>int getJDBCMinorVersion() 1.4</code>	returns the major or minor JDBC version numbers of the driver that established the database connection. For example, a JDBC 3.0 driver has major version number 3 and minor version number 0.
	<ul style="list-style-type: none">• <code>int getMaxConnections()</code>	returns the maximum number of concurrent connections allowed to this database.
	<ul style="list-style-type: none">• <code>int getMaxStatements()</code>	returns the maximum number of concurrently open statements allowed per database connection, or 0 if the number is unlimited or unknown.



java.sql.ResultSet 1.1

- `ResultSetMetaData getMetaData()`

returns the metadata associated with the current `ResultSet` columns.



java.sql.ResultSetMetaData 1.1

- `int getColumnCount()`

returns the number of columns in the current `ResultSet` object.

- `int getColumnDisplaySize(int column)`

returns the maximum width of the column specified by the index parameter.

Parameters: `column` The column number

- `String getColumnLabel(int column)`

returns the suggested title for the column.

Parameters: `column` The column number

- `String getColumnName(int column)`

returns the column name associated with the column index specified.

Parameters: `column` The column number



Transactions

You can group a set of statements to form a transaction. The transaction can be committed when all has gone well. Or, if an error has occurred in one of them, it can be rolled back as if none of the statements had been issued.

The major reason for grouping statements into transactions is database integrity. For example, suppose we want to transfer money from one bank account to another. Then, it is important that we simultaneously debit one account and credit another. If the system fails after debiting the first account but before crediting the other account, the debit needs to be undone.

If you group update statements to a transaction, then the transaction either succeeds in its entirety and it can be committed, or it fails somewhere in the middle. In that case, you can carry out a rollback and the database automatically undoes the effect of all updates that occurred since the last committed transaction.

By default, a database connection is in autocommit mode, and each SQL statement is committed to the database as soon as it is executed. Once a statement is committed, you cannot roll it back. Turn off this default when you use transactions:

```
conn.setAutoCommit(false);
```

Create a statement object in the normal way:

```
Statement stat = conn.createStatement();
```

Call `executeUpdate` any number of times:

```
stat.executeUpdate(command1);
stat.executeUpdate(command2);
stat.executeUpdate(command3);
...
```

If all statements have been executed without error, call the `commit` method:

```
conn.commit();
```

However, if an error occurred, call

```
conn.rollback();
```

Then, all statements until the last commit are automatically reversed. You typically issue a rollback when your transaction was interrupted by a `SQLException`.

Save Points

With some drivers, you can gain finer-grained control over the rollback process by using save points. Creating a save point marks a point to which you can later return without having to abandon the entire transaction. For example,

Code View:

```
Statement stat = conn.createStatement(); // start transaction; rollback() goes here
stat.executeUpdate(command1);
Savepoint svpt = conn.setSavepoint(); // set savepoint; rollback(svpt) goes here
stat.executeUpdate(command2);
if (...) conn.rollback(svpt); // undo effect of command2
...
conn.commit();
```

When you no longer need a save point, you should release it:

```
conn.releaseSavepoint(svpt);
```

Batch Updates

Suppose a program needs to execute many `INSERT` statements to populate a database table. You can improve the performance of the program by using a batch update. In a batch update, a sequence of statements is collected and submitted as a batch.

Note



Use the `supportsBatchUpdates` method of the `DatabaseMetaData` class to find out if your database supports this feature.

The statements in a batch can be actions such as `INSERT`, `UPDATE`, and `DELETE` as well as data definition statements such as `CREATE TABLE` and `DROP TABLE`. An exception is thrown if you add a `SELECT` statement to a batch. (Conceptually, a `SELECT` statement makes no sense in a batch because it returns a result set without updating the database.)

To execute a batch, you first create a `Statement` object in the usual way:

```
Statement stat = conn.createStatement();
```

Now, instead of calling `executeUpdate`, you call the `addBatch` method:

```
String command = "CREATE TABLE . . .";
stat.addBatch(command);

while (. . .)
{
    command = "INSERT INTO . . . VALUES (" + . . . + ")";
    stat.addBatch(command);
}
```

Finally, you submit the entire batch:

```
int[] counts = stat.executeBatch();
```

The call to `executeBatch` returns an array of the row counts for all submitted statements.

For proper error handling in batch mode, you want to treat the batch execution as a single transaction. If a batch fails in the middle, you want to roll back to the state before the beginning of the batch.

First, turn autocommit mode off, then collect the batch, execute it, commit it, and finally restore the original autocommit mode:[java.sql.Connection 1.1](#)

```
boolean autoCommit = conn.getAutoCommit();
conn.setAutoCommit(false);
Statement stat = conn.createStatement();
. . .
// keep calling stat.addBatch(. . .);
. . .
stat.executeBatch();
conn.commit();
conn.setAutoCommit(autoCommit);
```



java.sql.Connection 1.1

- `boolean getAutoCommit()`
- `void setAutoCommit(boolean b)`

gets or sets the autocommit mode of this connection to `b`. If autocommit is `true`, all statements are committed as soon as their execution is completed.

- `void commit()`

commits all statements that were issued since the last commit.

- `void rollback()`
undoes the effect of all statements that were issued since the last commit.
- `Savepoint setSavepoint() 1.4`
- `Savepoint setSavepoint(String name) 1.4`
sets an unnamed or named save point.
- `void rollback(Savepoint svpt) 1.4`
rolls back until the given save point.
- `void releaseSavepoint(Savepoint svpt) 1.4`
releases the given save point.



java.sql.Savepoint 1.4

- `int getSavepointId()`
gets the ID of this unnamed save point, or throws a `SQLException` if this is a named save point.
- `String getSavepointName()`
gets the name of this save point, or throws a `SQLException` if this is an unnamed save point.



java.sql.Statement 1.1

- `void addBatch(String command) 1.2`
adds the command to the current batch of commands for this statement.
- `int[] executeBatch() 1.2`
executes all commands in the current batch. Each value in the returned array corresponds to one of the batch statements. If it is nonnegative, it is a row count. If it is the value `SUCCESS_NO_INFO`, the statement succeeded, but no row count is available. If it is `EXECUTE_FAILED`, then the statement failed.



java.sql.DatabaseMetaData 1.1

- `boolean supportsBatchUpdates() 1.2`
returns `true` if the driver supports batch updates.

Advanced SQL Types

[Table 4-9](#) lists the SQL data types supported by JDBC and their equivalents in the Java programming language.

Table 4-9. SQL Data Types and Their Corresponding Java Types

SQL Data Type	Java Data Type
INTEGER OR INT	int

SMALLINT	short
NUMERIC (m, n), DECIMAL (m, n) or DEC (m, n)	java.math.BigDecimal
FLOAT (n)	double
REAL	float
DOUBLE	double
CHARACTER (n) or CHAR (n)	String
VARCHAR (n), LONG VARCHAR	String
BOOLEAN	boolean
DATE	java.sql.Date
TIME	java.sql.Time
TIMESTAMP	java.sql.Timestamp
BLOB	java.sql.Blob
CLOB	java.sql.Clob
ARRAY	java.sql.Array
ROWID	java.sql.RowId
NCHAR (n), NVARCHAR (n), LONG NVARCHAR	String
NCLOB	java.sql.NClob
SQLXML	java.sql.SQLXML

A SQL ARRAY is a sequence of values. For example, in a `Student` table, you can have a `Scores` column that is an `ARRAY OF INTEGER`. The `getArray` method returns an object of the interface type `java.sql.Array`. That interface has methods to fetch the array values.

When you get a LOB or an array from a database, the actual contents are fetched from the database only when you request individual values. This is a useful performance enhancement, as the data can be quite voluminous.

Some databases support `ROWID` values that describe the location of a row such that it can be retrieved very rapidly. JDBC 4 introduced an interface `java.sql.RowId` and supplied methods to supply the row ID in queries and retrieve it from results.

A national character string (`NCHAR` and its variants) stores strings in a local character encoding and sorts them using a local sorting convention. JDBC 4 provided methods for converting between Java `String` objects and national character strings in queries and results.

Some databases can store user-defined structured types. JDBC 3 provided a mechanism for automatically mapping structured SQL types to Java objects.

Some databases provide native storage for XML data. JDBC 4 introduced a `SQLXML` interface that can mediate between the internal XML representation and the DOM `Source/Result` interfaces, as well as binary streams. See the API documentation for the `SQLXML` class for details.

We do not discuss these advanced SQL types any further. You can find more information on these topics in the JDBC API Tutorial and Reference and the JDBC 4 specifications.



Connection Management in Web and Enterprise Applications

The simplistic database connection setup with a `database.properties` file, as described in the preceding sections, is suitable for small test programs, but it won't scale for larger applications.

When a JDBC application is deployed in a web or enterprise environment, the management of database connections is integrated with the JNDI. The properties of data sources across the enterprise can be stored in a directory. Using a directory allows for centralized management of user names, passwords, database names, and JDBC URLs.

In such an environment, you use the following code to establish a database connection:

Code View:

```
Context jndiContext = new InitialContext();
DataSource source = (DataSource) jndiContext.lookup("java:comp/env/jdbc/corejava");
Connection conn = source.getConnection();
```

Note that the `DriverManager` is no longer involved. Instead, the JNDI service locates a data source. A data source is an interface that allows for simple JDBC connections as well as more advanced services, such as executing distributed transactions that involve multiple databases. The `DataSource` interface is defined in the `javax.sql` standard extension package.

Note



In a Java EE 5 container, you don't even have to program the JNDI lookup. Simply use the `Resource` annotation on a `DataSource` field, and the data source reference will be set when your application is loaded:

```
@Resource("jdbc/corejava")
private DataSource source;
```

Of course, the data source needs to be configured somewhere. If you write database programs that execute in a servlet container such as Apache Tomcat or in an application server such as GlassFish, then you place the database configuration (including the JNDI name, JDBC URL, user name, and password) in a configuration file, or you set it in an admin GUI.

Management of user names and logins is just one of the issues that require special attention. A second issue involves the cost of establishing database connections. Our sample database programs used two strategies for obtaining a database connection. The `QueryDB` program in [Listing 4-3](#) established a single database connection at the start of the program and closed it at the end of the program. The `ViewDB` program in [Listing 4-4](#) opened a new connection whenever one was needed.

However, neither of these approaches is satisfactory. Database connections are a finite resource. If a user walks away from an application for some time, the connection should not be left open. Conversely, obtaining a connection for each query and closing it afterward is very costly.

The solution is to pool the connections. This means that database connections are not physically closed but are kept in a queue and reused. Connection pooling is an important service, and the JDBC specification provides hooks for implementors to supply it. However, the JDK itself does not provide any implementation, and database vendors don't usually include one with their JDBC driver either. Instead, vendors of web containers and application servers supply connection pool implementations.

Using a connection pool is completely transparent to the programmer. You acquire a connection from a source of pooled connections by obtaining a data source and calling `getConnection`. When you are done using the connection, call `close`. That doesn't close the physical connection but tells the pool that you are done using it. The connection pool typically makes an effort to pool prepared statements as well.

You have now learned about the JDBC fundamentals and know enough to implement simple database applications. However, as we mentioned at the beginning of this chapter, databases are complex and quite a few advanced topics are beyond the scope of this introductory chapter. For an overview of advanced JDBC capabilities, refer to the JDBC API Tutorial and Reference or the JDBC specifications.



Introduction to LDAP

In the preceding sections, you have seen how to interact with a relational database. In this section, we briefly look at hierarchical databases that use LDAP, the Lightweight Directory Access Protocol. This section is adapted from Core JavaServer Faces, 2nd ed., by Geary and Horstmann (Prentice Hall PTR 2007).

A hierarchical database is preferred over a relational database when the application data naturally follows a tree structure and when read operations greatly outnumber write operations. LDAP is most commonly used for the storage of directories that contain data such as user names, passwords, and permissions.

Note



For an in-depth discussion of LDAP, we recommend the "LDAP bible": Understanding and Deploying

LDAP Directory Services, 2nd ed., by Timothy Howes et al. (AddisonWesley Professional 2003).

An LDAP directory keeps all data in a tree structure, not in a set of tables as a relational database would. Each entry in the tree has the following:

- Zero or more attributes. An attribute has an ID and a value. An example attribute is `cn=John Q. Public`. (The ID `cn` stores the "common name." See [Table 4-10](#) for the meaning of commonly used LDAP attributes.)

Table 4-10. Commonly Used LDAP Attributes

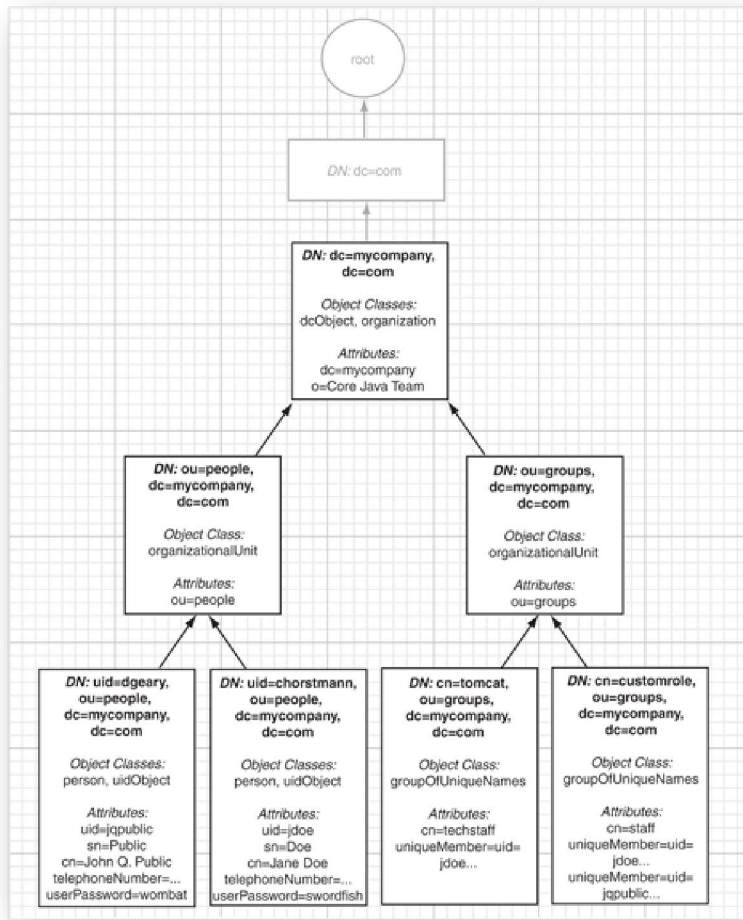
Attribute ID	Meaning
<code>dc</code>	Domain component
<code>cn</code>	Common name
<code>sn</code>	Surname
<code>dn</code>	Distinguished name
<code>o</code>	Organization
<code>ou</code>	Organizational unit
<code>uid</code>	Unique identifier

- One or more object classes. An object class defines the set of required and optional attributes for this element. For example, the object class `person` defines a required attribute `cn` and an optional attribute `telephoneNumber`. Of course, the object classes are different from Java classes, but they also support a notion of inheritance. For example, `organizationalPerson` is a subclass of `person` with additional attributes.
- A distinguished name (for example, `uid=jqpublic,ou=people,dc=mycompany,dc=com`). A distinguished name is a sequence of attributes that trace a path joining the entry with the root of the tree. There might be alternate paths, but one of them must be specified as distinguished.

[Figure 4-9](#) on the following page shows an example of a directory tree.

Figure 4-9. A directory tree

[\[View full size image\]](#)



How to organize a directory tree, and what information to put in it, can be a matter of intense debate. We do not discuss the issues here. Instead, we simply assume that an organizational scheme has been established and that the directory has been populated with the relevant user data.

Configuring an LDAP Server

You have several options for running an LDAP server to try out the programs in this section. Here are the most common choices:

- IBM Tivoli Directory Server
- Microsoft Active Directory
- Novell eDirectory
- OpenLDAP
- Sun Java System Directory Server for Solaris

We give you brief instructions for configuring OpenLDAP (<http://openldap.org>), a free server available for Linux and Windows and built into Mac OS X. If you use another directory server, the basic steps are similar.

If you use OpenLDAP, you need to edit the `slapd.conf` file before starting the LDAP server. (On Linux, the default location for the `slapd.conf` file is `/etc/ldap`, `/etc/openldap`, or `/usr/local/etc/openldap`.) Edit the `suffix` entry in `slapd.conf` to match the sample data set. This entry specifies the distinguished name suffix for this server. It should read

```
suffix "dc=mycompany, dc=com"
```

You also need to configure an LDAP user with administrative rights to edit the directory data. In OpenLDAP, add these lines to `slapd.conf`:

```
rootdn "cn=Manager,dc=mycompany,dc=com"
rootpw secret
```

We recommend that you specify authorization settings, although they are not strictly necessary for running the examples in this section. The following settings in `slapd.conf` permit the Manager user to read and write passwords, and everyone else to read all other attributes.

```
access to attr=userPassword
  by dn.base="cn=Manager,dc=mycompany,dc=com" write
  by self write
  by * none
access to *
  by dn.base="cn=Manager,dc=mycompany,dc=com" write
  by self write
  by * read
```

You can now start the LDAP server. On Linux, run the `slapd` service (typically in the `/usr/sbin` or `/usr/local/libexec` directory).

Next, populate the server with the sample data. Most LDAP servers allow the import of Lightweight Directory Interchange Format (LDIF) data. LDIF is a human-readable format that simply lists all directory entries, including their distinguished names, object classes, and attributes. [Listing 4-5](#) shows an LDIF file that describes our sample data.

For example, with OpenLDAP, you use the `ldapadd` tool to add the data to the directory:

```
ldapadd -f sample.ldif -x -D "cn=Manager,dc=mycompany,dc=com" -w secret
```

Listing 4-5. sample.ldif

Code View:

```
1. # Define top-level entry
2. dn: dc=mycompany,dc=com
3. objectClass: dcObject
4. objectClass: organization
5. dc: mycompany
6. o: Core Java Team
7.
8. # Define an entry to contain people
9. # searches for users are based on this entry
10. dn: ou=people,dc=mycompany,dc=com
11. objectClass: organizationalUnit
12. ou: people
13.
14. # Define a user entry for John Q. Public
15. dn: uid=jqpublic,ou=people,dc=mycompany,dc=com
16. objectClass: person
17. objectClass: uidObject
18. uid: jqpublic
19. sn: Public
20. cn: John Q. Public
21. telephoneNumber: +1 408 555 0017
22. userPassword: wombat
23.
24. # Define a user entry for Jane Doe
25. dn: uid=jdoe,ou=people,dc=mycompany,dc=com
26. objectClass: person
27. objectClass: uidObject
28. uid: jdoe
29. sn: Doe
30. cn: Jane Doe
31. telephoneNumber: +1 408 555 0029
32. userPassword: heffalump
33.
34. # Define an entry to contain LDAP groups
35. # searches for roles are based on this entry
36. dn: ou=groups,dc=mycompany,dc=com
37. objectClass: organizationalUnit
38. ou: groups
39.
```

```

40. # Define an entry for the "techstaff" group
41. dn: cn=techstaff,ou=groups,dc=mycompany,dc=com
42. objectClass: groupOfUniqueNames
43. cn: techstaff
44. uniqueMember: uid=jdoe,ou=people,dc=mycompany,dc=com
45.
46. # Define an entry for the "staff" group
47. dn: cn=staff,ou=groups,dc=mycompany,dc=com
48. objectClass: groupOfUniqueNames
49. cn: staff
50. uniqueMember: uid=jqpublic,ou=people,dc=mycompany,dc=com
51. uniqueMember: uid=jdoe,ou=people,dc=mycompany,dc=com

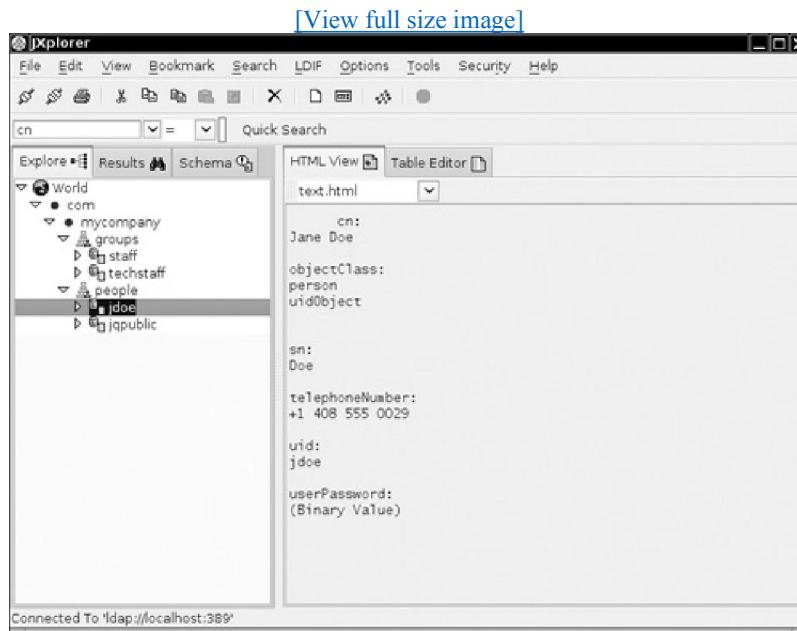
```

Before proceeding, it is a good idea to double-check that the directory contains the data that you need. We suggest that you download JXplorer (<http://www.jxplorer.org>) or Jarek Gawor's LDAP Browser/Editor (<http://www-unix.mcs.anl.gov/~gawor/ldap>). These convenient Java programs let you browse the contents of any LDAP server. Supply the following options:

- Host: localhost
- Port: 389
- Base DN: dc=mycompany, dc=com
- User DN: cn=Manager, dc=mycompany, dc=com
- Password: secret

Make sure the LDAP server has started, then connect. If everything is in order, you should see a directory tree similar to that shown in [Figure 4-10](#).

Figure 4-10. Inspecting an LDAP directory tree



Accessing LDAP Directory Information

Once your LDAP database is populated, connect to it with a Java program. Start by getting a directory context to the LDAP directory, with the following incantation:

```

Hashtable env = new Hashtable();
env.put(Context.SECURITY_PRINCIPAL, username);
env.put(Context.SECURITY_CREDENTIALS, password);

```

```
DirContext initial = new InitialDirContext(env);
DirContext context = (DirContext) initial.lookup("ldap://localhost:389");
```

Here, we connect to the LDAP server at the local host. The port number 389 is the default LDAP port.

If you connect to the LDAP database with an invalid user/password combination, an `AuthenticationException` is thrown.

Note



Sun's JNDI tutorial suggests an alternative way to connect to the server:

Code View:

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY, "com.sun.jndi.ldap.LdapCtxFactory");
env.put(Context.PROVIDER_URL, "ldap://localhost:389");
env.put(Context.SECURITY_PRINCIPAL, userDN);
env.put(Context.SECURITY_CREDENTIALS, password);
DirContext context = new InitialDirContext(env);
```

However, it seems undesirable to hardwire the Sun LDAP provider into your code. JNDI has an elaborate mechanism for configuring providers, and you should not lightly bypass it.

To list the attributes of a given entry, specify its distinguished name and then use the `getAttributes` method:

Code View:

```
Attributes attrs = context.getAttributes("uid=jqpublic,ou=people,dc=mycompany,dc=com");
```

You can get a specific attribute with the `get` method, for example,

```
Attribute commonNameAttribute = attrs.get("cn");
```

To enumerate all attributes, you use the `NamingEnumeration` class. The designers of this class felt that they too could improve on the standard Java iteration protocol, and they gave us this usage pattern:

```
NamingEnumeration<? extends Attribute> attrEnum = attrs.getAll();
while (attrEnum.hasMore())
{
    Attribute attr = attrEnum.next();
    String id = attr.getID();
    ...
}
```

Note the use of `hasMore` instead of `hasNext`.

If you know that an attribute has a single value, you can call the `get` method to retrieve it:

```
String commonName = (String) commonNameAttribute.get();
```

If an attribute can have multiple values, you need to use another `NamingEnumeration` to list them all:

```
NamingEnumeration<?> valueEnum = attr.getAll();
while (valueEnum.hasMore())
{
    Object value = valueEnum.next();
    ...
}
```

Note



As of Java SE 5.0, `NamingEnumeration` is a generic type. The type bound `<? extends Attribute>` means that the enumeration yields objects of some subtype of `Attribute`. Therefore, you don't need to cast the value that `next` returns—it has type `Attribute`. However, a `NamingEnumeration<?>` has no idea what it enumerates. Its `next` method returns an `Object`.

You now know how to query the directory for user data. Next, let us take up operations for modifying the directory contents.

To add a new entry, gather the set of attributes in a `BasicAttributes` object. (The `BasicAttributes` class implements the `Attributes` interface.)

```
Attributes attrs = new BasicAttributes();
attrs.put("uid", "alee");
attrs.put("sn", "Lee");
attrs.put("cn", "Amy Lee");
attrs.put("telephoneNumber", "+1 408 555 0033");
String password = "woozle";
attrs.put("userPassword", password.getBytes());
// the following attribute has two values
Attribute objclass = new BasicAttribute("objectClass");
objclass.add("uidObject");
objclass.add("person");
attrs.put(objclass);
```

Then call the `createSubcontext` method. Provide the distinguished name of the new entry and the attribute set.

```
context.createSubcontext("uid=alee,ou=people,dc=mycompany,dc=com", attrs);
```

Caution



When assembling the attributes, remember that the attributes are checked against the schema. Don't supply unknown attributes, and be sure to supply all attributes that are required by the object class. For example, if you omit the `sn` of `person`, the `createSubcontext` method will fail.

To remove an entry, call the `destroySubcontext` method:

```
context.destroySubcontext("uid=alee,ou=people,dc=mycompany,dc=com");
```

Finally, you might want to edit the attributes of an existing entry with this call:

```
context.modifyAttributes(distinguishedName, flag, attrs);
```

The `flag` parameter is one of the three constants `ADD_ATTRIBUTE`, `REMOVE_ATTRIBUTE`, or `REPLACE_ATTRIBUTE` defined in the `DirContext` class. The `attrs` parameter contains a set of the attributes to be added, removed, or replaced.

Conveniently, the `BasicAttributes(String, Object)` constructor constructs an attribute set with a single attribute. For example,

```
context.modifyAttributes("uid=alee,ou=people,dc=mycompany,dc=com",
    DirContext.ADD_ATTRIBUTE,
    new BasicAttributes("title", "CTO"));

context.modifyAttributes("uid=alee,ou=people,dc=mycompany,dc=com",
    DirContext.REMOVE_ATTRIBUTE,
    new BasicAttributes("telephoneNumber", "+1 408 555 0033"));

context.modifyAttributes("uid=alee,ou=people,dc=mycompany,dc=com",
    DirContext.REPLACE_ATTRIBUTE,
    new BasicAttributes("userPassword", password.getBytes()));
```

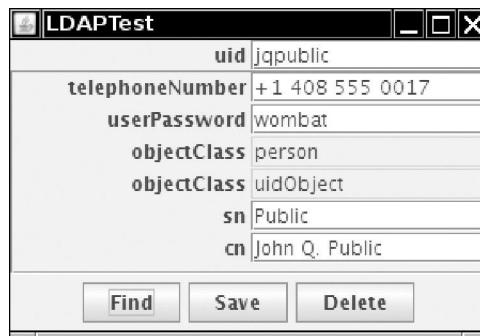
Finally, when you are done with a context, you should close it:

```
context.close();
```

The program in [Listing 4-6](#) demonstrates how to access a hierarchical database through LDAP. The program lets you view, modify, and delete information in a database with the sample data in [Listing 4-5](#).

Enter a `uid` into the text field and click the Find button to find an entry. If you edit the entry and click Save, your changes are saved. If you edited the `uid` field, a new entry is created. Otherwise, the existing entry is updated. You can also delete the entry by clicking the Delete button (see [Figure 4-11](#)).

Figure 4-11. Accessing a hierarchical database



Here is a brief description of the program:

- The configuration for the LDAP server is contained in the file `ldapserver.properties`. The file defines the URL, user name, and password of the server, like this:

```
ldap.username=cn=Manager,dc=mycompany,dc=com
ldap.password=secret
ldap.url=ldap://localhost:389
```

The `getContext` method reads the file and obtains the directory context.

- When the user clicks the Find button, the `findEntry` method fetches the attribute set for the entry with the given `uid`. The attribute set is used to construct a new `DataPanel`.
- The `DataPanel` constructor iterates over the attribute set and adds a label and text field for each ID/value pair.
- When the user clicks the Delete button, the `deleteEntry` method deletes the entry with the given `uid` and discards the data panel.
- When the user clicks the Save button, the `DataPanel` constructs a `BasicAttributes` object with the current contents of the text fields. The `saveEntry` method checks whether the `uid` has changed. If the user edited the `uid`, a new entry is created. Otherwise, the modified attributes are updated. The modification code is simple because we have only one attribute with multiple values, namely, `objectClass`. In general, you would need to work harder to handle multiple values for each attribute.
- Similar to the program in [Listing 4-4](#), we close the directory context when the frame window is closing.

You now know enough about directory operations to carry out the tasks that you will commonly need when working with LDAP directories. A good source for more advanced information is the JNDI tutorial at <http://java.sun.com/products/jndi/tutorial/>.

Listing 4-6. `LDAPTest.java`

Code View:

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.io.*;
4. import java.util.*;
5. import javax.naming.*;
6. import javax.naming.directory.*;
7. import javax.swing.*;
8.
9. /**
10. * This program demonstrates access to a hierarchical database through LDAP
```

```
11. * @version 1.01 2007-06-28
12. * @author Cay Horstmann
13. */
14. public class LDAPTest
15. {
16.     public static void main(String[] args)
17.     {
18.         EventQueue.invokeLater(new Runnable()
19.         {
20.             public void run()
21.             {
22.                 JFrame frame = new LDAPFrame();
23.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
24.                 frame.setVisible(true);
25.             }
26.         });
27.     }
28. }
29.
30. /**
31. * The frame that holds the data panel and the navigation buttons.
32. */
33. class LDAPFrame extends JFrame
34. {
35.     public LDAPFrame()
36.     {
37.         setTitle("LDAPTest");
38.         setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
39.
40.         JPanel northPanel = new JPanel();
41.         northPanel.setLayout(new java.awt.GridLayout(1, 2, 3, 1));
42.         northPanel.add(new JLabel("uid", SwingConstants.RIGHT));
43.         uidField = new JTextField();
44.         northPanel.add(uidField);
45.         add(northPanel, BorderLayout.NORTH);
46.
47.         JPanel buttonPanel = new JPanel();
48.         add(buttonPanel, BorderLayout.SOUTH);
49.
50.         findButton = new JButton("Find");
51.         findButton.addActionListener(new ActionListener()
52.         {
53.             public void actionPerformed(ActionEvent event)
54.             {
55.                 findEntry();
56.             }
57.         });
58.         buttonPanel.add(findButton);
59.
60.         saveButton = new JButton("Save");
61.         saveButton.addActionListener(new ActionListener()
62.         {
63.             public void actionPerformed(ActionEvent event)
64.             {
65.                 saveEntry();
66.             }
67.         });
68.         buttonPanel.add(saveButton);
69.
70.         deleteButton = new JButton("Delete");
71.         deleteButton.addActionListener(new ActionListener()
72.         {
73.             public void actionPerformed(ActionEvent event)
74.             {
75.                 deleteEntry();
76.             }
77.         });
78.         buttonPanel.add(deleteButton);
79.
80.         addWindowListener(new WindowAdapter()
81.         {
82.             public void windowClosing(WindowEvent event)
83.             {
84.                 try
85.                 {
86.                     if (context != null) context.close();
87.                 }
88.                 catch (NamingException e)
89.                 {
90.                     e.printStackTrace();
91.                 }
92.             }
93.         });
94.     }
95. }
```

```
91.         }
92.     }
93.   });
94. }
95.
96. /**
97. * Finds the entry for the uid in the text field.
98. */
99. public void findEntry()
100. {
101.     try
102.     {
103.         if (scrollPane != null) remove(scrollPane);
104.         String dn = "uid=" + uidField.getText() + ",ou=people,dc=mycompany,dc=com";
105.         if (context == null) context = getContext();
106.         attrs = context.getAttributes(dn);
107.         dataPanel = new DataPanel(attrs);
108.         scrollPane = new JScrollPane(dataPanel);
109.         add(scrollPane, BorderLayout.CENTER);
110.         validate();
111.         uid = uidField.getText();
112.     }
113.     catch (NamingException e)
114.     {
115.         JOptionPane.showMessageDialog(this, e);
116.     }
117.     catch (IOException e)
118.     {
119.         JOptionPane.showMessageDialog(this, e);
120.     }
121. }
122.
123. /**
124. * Saves the changes that the user made.
125. */
126. public void saveEntry()
127. {
128.     try
129.     {
130.         if (dataPanel == null) return;
131.         if (context == null) context = getContext();
132.         if (uidField.getText().equals(uid)) // update existing entry
133.         {
134.             String dn = "uid=" + uidField.getText() + ",ou=people,dc=mycompany,dc=com";
135.             Attributes editedAttrs = dataPanel.getEditedAttributes();
136.             NamingEnumeration<? extends Attribute> attrEnum = attrs.getAll();
137.             while (attrEnum.hasMore())
138.             {
139.                 Attribute attr = attrEnum.next();
140.                 String id = attr.getID();
141.                 Attribute editedAttr = editedAttrs.get(id);
142.                 if (editedAttr != null && !attr.get().equals(editedAttr.get())) context
143.                     .modifyAttributes(dn, DirContext.REPLACE_ATTRIBUTE,
144.                         new BasicAttributes(id, editedAttr.get()));
145.             }
146.         }
147.         else
148.             // create new entry
149.         {
150.             String dn = "uid=" + uidField.getText() + ",ou=people,dc=mycompany,dc=com";
151.             attrs = dataPanel.getEditedAttributes();
152.             Attribute objclass = new BasicAttribute("objectClass");
153.             objclass.add("uidObject");
154.             objclass.add("person");
155.             attrs.put(objclass);
156.             attrs.put("uid", uidField.getText());
157.             context.createSubcontext(dn, attrs);
158.         }
159.
160.         findEntry();
161.     }
162.     catch (NamingException e)
163.     {
164.         JOptionPane.showMessageDialog(LDAPFrame.this, e);
165.         e.printStackTrace();
166.     }
167.     catch (IOException e)
168.     {
169.         JOptionPane.showMessageDialog(LDAPFrame.this, e);
170.         e.printStackTrace();
```

```
171.      }
172.    }
173.
174. /**
175.  * Deletes the entry for the uid in the text field.
176. */
177. public void deleteEntry()
178. {
179.   try
180.   {
181.     String dn = "uid=" + uidField.getText() + ",ou=people,dc=mycompany,dc=com";
182.     if (context == null) context = getContext();
183.     context.destroySubcontext(dn);
184.     uidField.setText("");
185.     remove(scrollPane);
186.     scrollPane = null;
187.     repaint();
188.   }
189.   catch (NamingException e)
190.   {
191.     JOptionPane.showMessageDialog(LDAPFrame.this, e);
192.     e.printStackTrace();
193.   }
194.   catch (IOException e)
195.   {
196.     JOptionPane.showMessageDialog(LDAPFrame.this, e);
197.     e.printStackTrace();
198.   }
199. }
200.
201. /**
202.  * Gets a context from the properties specified in the file ldapserver.properties
203.  * @return the directory context
204. */
205. public static DirContext getContext() throws NamingException, IOException
206. {
207.   Properties props = new Properties();
208.   FileInputStream in = new FileInputStream("ldapserver.properties");
209.   props.load(in);
210.   in.close();
211.
212.   String url = props.getProperty("ldap.url");
213.   String username = props.getProperty("ldap.username");
214.   String password = props.getProperty("ldap.password");
215.
216.   Hashtable<String, String> env = new Hashtable<String, String>();
217.   env.put(Context.SECURITY_PRINCIPAL, username);
218.   env.put(Context.SECURITY_CREDENTIALS, password);
219.   DirContext initial = new InitialDirContext(env);
220.   DirContext context = (DirContext) initial.lookup(url);
221.
222.   return context;
223. }
224.
225. public static final int DEFAULT_WIDTH = 300;
226. public static final int DEFAULT_HEIGHT = 200;
227.
228. private JButton findButton;
229. private JButton saveButton;
230. private JButton deleteButton;
231.
232. private JTextField uidField;
233. private DataPanel dataPanel;
234. private Component scrollPane;
235.
236. private DirContext context;
237. private String uid;
238. private Attributes attrs;
239. }
240.
241. /**
242.  * This panel displays the contents of a result set.
243. */
244. class DataPanel extends JPanel
245. {
246.   /**
247.    * Constructs the data panel.
248.    * @param attrs the attributes of the given entry
249.    */
250.   public DataPanel(Attributes attrs) throws NamingException
```

```

251.     {
252.         setLayout(new java.awt.GridLayout(0, 2, 3, 1));
253.
254.         NamingEnumeration<? extends Attribute> attrEnum = attrs.getAll();
255.         while (attrEnum.hasMore())
256.         {
257.             Attribute attr = attrEnum.next();
258.             String id = attr.getID();
259.
260.             NamingEnumeration<?> valueEnum = attr.getAll();
261.             while (valueEnum.hasMore())
262.             {
263.                 Object value = valueEnum.next();
264.                 if (id.equals("userPassword")) value = new String((byte[]) value);
265.
266.                 JLabel idLabel = new JLabel(id, SwingConstants.RIGHT);
267.                 JTextField valueField = new JTextField("") + value);
268.                 if (id.equals("objectClass")) valueField.setEditable(false);
269.                 if (!id.equals("uid"))
270.                 {
271.                     add(idLabel);
272.                     add(valueField);
273.                 }
274.             }
275.         }
276.     }
277.
278.     public Attributes getEditedAttributes()
279.     {
280.         Attributes attrs = new BasicAttributes();
281.         for (int i = 0; i < getComponentCount(); i += 2)
282.         {
283.             JLabel idLabel = (JLabel) getComponent(i);
284.             JTextField valueField = (JTextField) getComponent(i + 1);
285.             String id = idLabel.getText();
286.             String value = valueField.getText();
287.             if (id.equals("userPassword")) attrs.put("userPassword", value.getBytes());
288.             else if (!id.equals("")) && !id.equals("objectClass")) attrs.put(id, value);
289.         }
290.         return attrs;
291.     }
292. }

```



javax.naming.directory.InitialDirContext 1.3

- InitialDirContext(Hashtable env)

constructs a directory context, using the given environment settings. The hash table can contain bindings for Context.SECURITY_PRINCIPAL, Context.SECURITY_CREDENTIALS, and other keys—see the API documentation for the javax.naming.Context interface for details.



javax.naming.Context 1.3

- Object lookup(String name)

looks up the object with the given name. The return value depends on the nature of this context. It commonly is a subtree context or a leaf object.

- Context createSubcontext(String name)

creates a subcontext with the given name. The subcontext becomes a child of this context. All path components of the name, except for the last one, must exist.

- `void destroySubcontext(String name)`
destroys the subcontext with the given name. All path components of the name, except for the last one, must exist.
- `void close()`
closes this context.

API

javax.naming.directory.DirContext 1.3

- `Attributes getAttributes(String name)`
gets the attributes of the entry with the given name.
- `void modifyAttributes(String name, int flag, Attributes modes)`
modifies the attributes of the entry with the given name. The value `flag` is one of `DirContext.ADD_ATTRIBUTE`, `DirContext.REMOVE_ATTRIBUTE`, or `DirContext.REPLACE_ATTRIBUTE`.

API

javax.naming.directory.Attributes 1.3

- `Attribute get(String id)`
gets the attribute with the given ID.
- `NamingEnumeration<? extends Attribute> getAll()`
yields an enumeration that iterates through all attributes in this attribute set.
- `Attribute put(Attribute attr)`
- `Attribute put(String id, Object value)`
adds an attribute to this attribute set.

API

javax.naming.directory.BasicAttributes 1.3

- `BasicAttributes(String id, Object value)`
constructs an attribute set that contains a single attribute with the given ID and value.

API

javax.naming.directory.Attribute 1.3

- `String getID()`
gets the ID of this attribute.
- `Object get()`
gets the first attribute value of this attribute if the values are ordered or an arbitrary value if they are unordered.

- `NamingEnumeration<?> getAll()`
yields an enumeration that iterates through all values of this attribute.

API

javax.naming.NamingEnumeration<T> 1.3

- `boolean hasMore()`
returns `true` if this enumeration object has more elements.
- `T next()`
returns the next element of this enumeration.

In this chapter, you have learned how to work with relational databases in Java, and you were introduced to hierarchical databases. The next chapter covers the important topic of internationalization, showing you how to make your software usable for customers around the world.





Chapter 5. Internationalization

- [LOCALES](#)
- [NUMBER FORMATS](#)
- [DATE AND TIME](#)
- [COLLATION](#)
- [MESSAGE FORMATTING](#)
- [TEXT FILES AND CHARACTER SETS](#)
- [RESOURCE BUNDLES](#)
- [A COMPLETE EXAMPLE](#)

There's a big world out there; we hope that lots of its inhabitants will be interested in your software. The Internet, after all, effortlessly spans the barriers between countries. On the other hand, when you pay no attention to an international audience, you are putting up a barrier.

The Java programming language was the first language designed from the ground up to support internationalization. From the beginning, it had the one essential feature needed for effective internationalization: It used Unicode for all strings. Unicode support makes it easy to write programs in the Java programming language that manipulate strings in any one of multiple languages.

Many programmers believe that all they need to do to internationalize their application is to support Unicode and to translate the messages in the user interface. However, as this chapter demonstrates, there is a lot more to internationalizing programs than just Unicode support. Dates, times, currencies—even numbers—are formatted differently in different parts of the world. You need an easy way to configure menu and button names, message strings, and keyboard shortcuts for different languages.

In this chapter, we show you how to write internationalized Java applications and applets and how to localize date, time, numbers, text, and GUIs. We show you tools that Java offers for writing internationalized programs. We close this chapter with a complete example, a retirement calculator applet that can change how it displays its results depending on the location of the machine that is downloading it.

Note



For additional information on internationalization, check out the informative web site
<http://www.joconner.com/javai18n>, as well as the official Sun site
<http://java.sun.com/javase/technologies/core/basic/intl/>.

Locales

When you look at an application that is adapted to an international market, the most obvious difference you notice is the language. This observation is actually a bit too limiting for true internationalization: Countries can share a common language, but you still might need to do some work to make computer users of both countries happy.^[1]

[1] "We have really everything in common with America nowadays, except, of course, language." Oscar Wilde.

In all cases, menus, button labels, and program messages will need to be translated to the local language; they might also need to be rendered in a different script. There are many more subtle differences; for example, numbers are formatted quite differently in English and in German. The number

123,456.78

should be displayed as

123.456,78

for a German user. That is, the role of the decimal point and the decimal comma separator are reversed. There are similar variations in the display of dates. In the United States, dates are somewhat irrationally displayed as month/day/year. Germany uses the more sensible order of day/month/year, whereas in China, the usage is year/month/day. Thus, the date

3/22/61

should be presented as

22.03.1961

to a German user. Of course, if the month names are written out explicitly, then the difference in languages becomes apparent. The English

March 22, 1961

should be presented as

22. März 1961

in German, or

1961 年 3 月 22 日

in Chinese.

There are several formatter classes that take these differences into account. To control the formatting, you use the `Locale` class. A locale describes

- A language.
- Optionally, a location.
- Optionally, a variant.

For example, in the United States, you use a locale with

`language=English, location=United States.`

In Germany, you use a locale with

`language=German, location=Germany.`

Switzerland has four official languages (German, French, Italian, and Rhaeto-Romance). A German speaker in Switzerland would want to use a locale with

`language=German, location=Switzerland`

This locale would make formatting work similarly to how it would work for the German locale; however, currency values would be expressed in Swiss francs, not German marks.

If you only specify the language, say,

`language=German`

then the locale cannot be used for country-specific issues such as currencies.

Variants are, fortunately, rare and are needed only for exceptional or system-dependent situations. For example, the Norwegians are having a hard time agreeing on the spelling of their language (a derivative of Danish). They use two spelling rule sets: a traditional one called Bokmål and a new one called Nynorsk. The traditional spelling would be expressed as a variant

`language=Norwegian, location=Norway, variant=Bokmål`

To express the language and location in a concise and standardized manner, the Java programming language uses codes that were defined by the International Organization for Standardization (ISO). The local language is expressed as a lowercase two-letter code, following ISO 639-1, and the country code is expressed as an uppercase two-letter code, following ISO 3166-1. [Tables 5-1](#) and [5-2](#) show some of the most common codes.

Table 5-1. Common ISO 639-1 Language Codes

Language	Code
Chinese	zh

Danish	da
Dutch	nl
English	en
French	fr
Finnish	fi
German	de
Greek	el
Italian	it
Japanese	ja
Korean	ko
Norwegian	no
Portuguese	pt
Spanish	sp
Swedish	sv
Turkish	tr

Table 5-2. Common ISO 3166-1 Country Codes

Country	Code
Austria	AT
Belgium	BE
Canada	CA
China	CN
Denmark	DK
Finland	FI
Germany	DE
Great Britain	GB
Greece	GR
Ireland	IE
Italy	IT
Japan	JP
Korea	KR
The Netherlands	NL
Norway	NO
Portugal	PT
Spain	ES
Sweden	SE
Switzerland	CH
Taiwan	TW
Turkey	TR
United States	US

Note



For a full list of ISO 639-1 codes, see, for example, http://www.loc.gov/standards/iso639-2/php/code_list.php. You can find a full list of the ISO 3166-1 codes at <http://www.iso.org/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/index.html>.

These codes do seem a bit random, especially because some of them are derived from local languages (German = Deutsch = de, Chinese = zhongwen = zh), but at least they are standardized.

To describe a locale, you concatenate the language, country code, and variant (if any) and pass this string to the constructor of the `Locale` class.

```
Locale german = new Locale("de");
Locale germanGermany = new Locale("de", "DE");
Locale germanSwitzerland = new Locale("de", "CH");
Locale norwegianNorwayBokmål = new Locale("no", "NO", "B");
```

For your convenience, Java SE predefines a number of locale objects:

```
Locale.CANADA
Locale.CANADA_FRENCH
Locale.CHINA
Locale.FRANCE
Locale.GERMANY
Locale.ITALY
Locale.JAPAN
Locale.KOREA
Locale.PRC
Locale.TAIWAN
Locale.UK
Locale.US
```

Java SE also predefines a number of language locales that specify just a language without a location:

```
Locale.CHINESE
Locale.ENGLISH
Locale.FRENCH
Locale.GERMAN
Locale.ITALIAN
Locale.JAPANESE
Locale.KOREAN
Locale.SIMPLIFIED_CHINESE
Locale.TRADITIONAL_CHINESE
```

Besides constructing a locale or using a predefined one, you have two other methods for obtaining a locale object.

The static `getDefault` method of the `Locale` class initially gets the default locale as stored by the local operating system. You can change the default Java locale by calling `setDefault`; however, that change only affects your program, not the operating system. Similarly, in an applet, the `getLocale` method returns the locale of the user viewing the applet.

Finally, all locale-dependent utility classes can return an array of the locales they support. For example,

```
Locale[] supportedLocales = DateFormat.getAvailableLocales();
```

returns all locales that the `DateFormat` class can handle.

Tip



For testing, you might want to switch the default locale of your program. Supply language and region properties when you launch your program. For example, here we set the default locale to German (Switzerland):

```
java -Duser.language=de -Duser.region=CH Program
```

Once you have a locale, what can you do with it? Not much, as it turns out. The only useful methods in the `Locale` class are the ones for identifying the language and country codes. The most important one is `getDisplayName`. It returns a string describing the locale. This string does not contain the cryptic two-letter codes, but it is in a form that can be presented to a user, such as

German (Switzerland)

Actually, there is a problem here. The display name is issued in the default locale. That might not be appropriate. If your user already selected German as the preferred language, you probably want to present the string in German. You can do just that by giving the German locale as a parameter: The code

```
Locale loc = new Locale("de", "CH");
System.out.println(loc.getDisplayName(Locale.GERMAN));
```

prints

```
Deutsch (Schweiz)
```

This example shows why you need `Locale` objects. You feed it to locale-aware methods that produce text that is presented to users in different locations. You can see many examples in the following sections.

API

java.util.Locale 1.1

- `Locale(String language)`
- `Locale(String language, String country)`
- `Locale(String language, String country, String variant)`

constructs a locale with the given language, country, and variant.
- `static Locale getDefault()`

returns the default locale.
- `static void setDefault(Locale loc)`

sets the default locale.
- `String getDisplayName()`

returns a name describing the locale, expressed in the current locale.
- `String getDisplayName(Locale loc)`

returns a name describing the locale, expressed in the given locale.
- `String getLanguage()`

returns the language code, a lowercase two-letter ISO-639 code.
- `String getDisplayLanguage()`

returns the name of the language, expressed in the current locale.
- `String getDisplayLanguage(Locale loc)`

returns the name of the language, expressed in the given locale.
- `String getCountry()`

returns the country code as an uppercase two-letter ISO-3166 code.
- `String getDisplayCountry()`

returns the name of the country, expressed in the current locale.
- `String getDisplayCountry(Locale loc)`

returns the name of the country, expressed in the given locale.
- `String getVariant()`

returns the variant string.

- `String getDisplayVariant()`

returns the name of the variant, expressed in the current locale.

- `String getDisplayVariant(Locale loc)`

returns the name of the variant, expressed in the given locale.

- `String toString()`

returns a description of the locale, with the language, country, and variant separated by underscores (e.g., "de_CH").

API

java.awt.Applet 1.0

- `Locale getLocale() [1.1]`

gets the locale for this applet.



Chapter 5. Internationalization

- [LOCALES](#)
- [NUMBER FORMATS](#)
- [DATE AND TIME](#)
- [COLLATION](#)
- [MESSAGE FORMATTING](#)
- [TEXT FILES AND CHARACTER SETS](#)
- [RESOURCE BUNDLES](#)
- [A COMPLETE EXAMPLE](#)

There's a big world out there; we hope that lots of its inhabitants will be interested in your software. The Internet, after all, effortlessly spans the barriers between countries. On the other hand, when you pay no attention to an international audience, you are putting up a barrier.

The Java programming language was the first language designed from the ground up to support internationalization. From the beginning, it had the one essential feature needed for effective internationalization: It used Unicode for all strings. Unicode support makes it easy to write programs in the Java programming language that manipulate strings in any one of multiple languages.

Many programmers believe that all they need to do to internationalize their application is to support Unicode and to translate the messages in the user interface. However, as this chapter demonstrates, there is a lot more to internationalizing programs than just Unicode support. Dates, times, currencies—even numbers—are formatted differently in different parts of the world. You need an easy way to configure menu and button names, message strings, and keyboard shortcuts for different languages.

In this chapter, we show you how to write internationalized Java applications and applets and how to localize date, time, numbers, text, and GUIs. We show you tools that Java offers for writing internationalized programs. We close this chapter with a complete example, a retirement calculator applet that can change how it displays its results depending on the location of the machine that is downloading it.

Note

For additional information on internationalization, check out the informative web site
<http://www.joconner.com/javai18n>, as well as the official Sun site
<http://java.sun.com/javase/technologies/core/basic/intl/>.

Locales

When you look at an application that is adapted to an international market, the most obvious difference you notice is the language. This observation is actually a bit too limiting for true internationalization: Countries can share a common language, but you still might need to do some work to make computer users of both countries happy.^[1]

[1] "We have really everything in common with America nowadays, except, of course, language." Oscar Wilde.

In all cases, menus, button labels, and program messages will need to be translated to the local language; they might also need to be rendered in a different script. There are many more subtle differences; for example, numbers are formatted quite differently in English and in German. The number

123,456.78

should be displayed as

123.456,78

for a German user. That is, the role of the decimal point and the decimal comma separator are reversed. There are similar variations in the display of dates. In the United States, dates are somewhat irrationally displayed as month/day/year. Germany uses the more sensible order of day/month/year, whereas in China, the usage is year/month/day. Thus, the date

3/22/61

should be presented as

22.03.1961

to a German user. Of course, if the month names are written out explicitly, then the difference in languages becomes apparent. The English

March 22, 1961

should be presented as

22. März 1961

in German, or

1961 年 3 月 22 日

in Chinese.

There are several formatter classes that take these differences into account. To control the formatting, you use the `Locale` class. A locale describes

- A language.
- Optionally, a location.
- Optionally, a variant.

For example, in the United States, you use a locale with

language=English, location=United States.

In Germany, you use a locale with

language=German, location=Germany.

Switzerland has four official languages (German, French, Italian, and Rhaeto-Romance). A German speaker in Switzerland would want to use a locale with

language=German, location=Switzerland

This locale would make formatting work similarly to how it would work for the German locale; however, currency values would be expressed in Swiss francs, not German marks.

If you only specify the language, say,

language=German

then the locale cannot be used for country-specific issues such as currencies.

Variants are, fortunately, rare and are needed only for exceptional or system-dependent situations. For example, the Norwegians are having a hard time agreeing on the spelling of their language (a derivative of Danish). They use two spelling rule sets: a traditional one called Bokmål and a new one called Nynorsk. The traditional spelling would be expressed as a variant

language=Norwegian, location=Norway, variant=Bokmål

To express the language and location in a concise and standardized manner, the Java programming language uses codes that were defined by the International Organization for Standardization (ISO). The local language is expressed as a lowercase two-letter code, following ISO 639-1, and the country code is expressed as an uppercase two-letter code, following ISO 3166-1. [Tables 5-1](#) and [5-2](#) show some of the most common codes.

Table 5-1. Common ISO 639-1 Language Codes

Language	Code
Chinese	zh
Danish	da
Dutch	nl
English	en
French	fr
Finnish	fi
German	de
Greek	el
Italian	it
Japanese	ja
Korean	ko
Norwegian	no
Portuguese	pt
Spanish	sp
Swedish	sv
Turkish	tr

Table 5-2. Common ISO 3166-1 Country Codes

Country	Code
Austria	AT
Belgium	BE
Canada	CA
China	CN
Denmark	DK
Finland	FI
Germany	DE
Great Britain	GB
Greece	GR

Ireland	IE
Italy	IT
Japan	JP
Korea	KR
The Netherlands	NL
Norway	NO
Portugal	PT
Spain	ES
Sweden	SE
Switzerland	CH
Taiwan	TW
Turkey	TR
United States	US

Note

- For a full list of ISO 639-1 codes, see, for example, http://www.loc.gov/standards/iso639-2/php/code_list.php. You can find a full list of the ISO 3166-1 codes at <http://www.iso.org/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/index.html>.

These codes do seem a bit random, especially because some of them are derived from local languages (German = Deutsch = de, Chinese = zhongwen = zh), but at least they are standardized.

To describe a locale, you concatenate the language, country code, and variant (if any) and pass this string to the constructor of the `Locale` class.

```
Locale german = new Locale("de");
Locale germanGermany = new Locale("de", "DE");
Locale germanSwitzerland = new Locale("de", "CH");
Locale norwegianNorwayBokmål = new Locale("no", "NO", "B");
```

For your convenience, Java SE predefines a number of locale objects:

```
Locale.CANADA
Locale.CANADA_FRENCH
Locale.CHINA
Locale.FRANCE
Locale.GERMANY
Locale.ITALY
Locale.JAPAN
Locale.KOREA
Locale.PRC
Locale.TAIWAN
Locale.UK
Locale.US
```

Java SE also predefines a number of language locales that specify just a language without a location:

```
Locale.CHINESE
Locale.ENGLISH
Locale.FRENCH
Locale.GERMAN
Locale.ITALIAN
Locale.JAPANESE
Locale.KOREAN
Locale.SIMPLIFIED_CHINESE
Locale.TRADITIONAL_CHINESE
```

Besides constructing a locale or using a predefined one, you have two other methods for obtaining a locale object.

The static `getDefault` method of the `Locale` class initially gets the default locale as stored by the local operating system. You

can change the default Java locale by calling `setDefault`; however, that change only affects your program, not the operating system. Similarly, in an applet, the `getLocale` method returns the locale of the user viewing the applet.

Finally, all locale-dependent utility classes can return an array of the locales they support. For example,

```
Locale[] supportedLocales = DateFormat.getAvailableLocales();
```

returns all locales that the `DateFormat` class can handle.

Tip

 For testing, you might want to switch the default locale of your program. Supply language and region properties when you launch your program. For example, here we set the default locale to German (Switzerland):

```
java -Duser.language=de -Duser.region=CH Program
```

Once you have a locale, what can you do with it? Not much, as it turns out. The only useful methods in the `Locale` class are the ones for identifying the language and country codes. The most important one is `getDisplayName`. It returns a string describing the locale. This string does not contain the cryptic two-letter codes, but it is in a form that can be presented to a user, such as

```
German (Switzerland)
```

Actually, there is a problem here. The display name is issued in the default locale. That might not be appropriate. If your user already selected German as the preferred language, you probably want to present the string in German. You can do just that by giving the German locale as a parameter: The code

```
Locale loc = new Locale("de", "CH");
System.out.println(loc.getDisplayName(Locale.GERMAN));
```

prints

```
Deutsch (Schweiz)
```

This example shows why you need `Locale` objects. You feed it to locale-aware methods that produce text that is presented to users in different locations. You can see many examples in the following sections.



java.util.Locale 1.1

- `Locale(String language)`
- `Locale(String language, String country)`
- `Locale(String language, String country, String variant)`
constructs a locale with the given language, country, and variant.
- `static Locale getDefault()`
returns the default locale.
- `static void setDefault(Locale loc)`
sets the default locale.
- `String getDisplayName()`
returns a name describing the locale, expressed in the current locale.
- `String getDisplayName(Locale loc)`

returns a name describing the locale, expressed in the given locale.

- `String getLanguage()`

returns the language code, a lowercase two-letter ISO-639 code.

- `String getDisplayLanguage()`

returns the name of the language, expressed in the current locale.

- `String getDisplayLanguage(Locale loc)`

returns the name of the language, expressed in the given locale.

- `String getCountry()`

returns the country code as an uppercase two-letter ISO-3166 code.

- `String getDisplayCountry()`

returns the name of the country, expressed in the current locale.

- `String getDisplayCountry(Locale loc)`

returns the name of the country, expressed in the given locale.

- `String getVariant()`

returns the variant string.

- `String getDisplayVariant()`

returns the name of the variant, expressed in the current locale.

- `String getDisplayVariant(Locale loc)`

returns the name of the variant, expressed in the given locale.

- `String toString()`

returns a description of the locale, with the language, country, and variant separated by underscores (e.g., "de_CH").



java.awt.Applet 1.0

- `Locale getLocale() [1.1]`

gets the locale for this applet.



Number Formats

We already mentioned how number and currency formatting is highly locale dependent. The Java library supplies a collection of formatter objects that can format and parse numeric values in the `java.text` package. You go through the following steps to format a number for a particular locale:

1. Get the locale object, as described in the preceding section.
2. Use a "factory method" to obtain a formatter object.
3. Use the formatter object for formatting and parsing.

The factory methods are static methods of the `NumberFormat` class that take a `Locale` argument. There are three factory methods: `getNumberInstance`, `getCurrencyInstance`, and `getPercentInstance`. These methods return objects that can format and parse numbers, currency amounts, and percentages, respectively. For example, here is how you can format a currency value in German:

```
Locale loc = new Locale("de", "DE");
NumberFormat currFmt = NumberFormat.getCurrencyInstance(loc);
double amt = 123456.78;
String result = currFmt.format(amt);
```

The result is

123.456,78€

Note that the currency symbol is € and that it is placed at the end of the string. Also, note the reversal of decimal points and decimal commas.

Conversely, to read in a number that was entered or stored with the conventions of a certain locale, use the `parse` method. For example, the following code parses the value that the user typed into a text field. The `parse` method can deal with decimal points and commas, as well as digits in other languages.

```
TextField inputField;
...
NumberFormat fmt = NumberFormat.getNumberInstance();
// get number formatter for default locale
Number input = fmt.parse(inputField.getText().trim());
double x = input.doubleValue();
```

The return type of `parse` is the abstract type `Number`. The returned object is either a `Double` or a `Long` wrapper object, depending on whether the parsed number was a floating-point number. If you don't care about the distinction, you can simply use the `doubleValue` method of the `Number` class to retrieve the wrapped number.

Caution

 Objects of type `Number` are not automatically unboxed—you cannot simply assign a `Number` object to a primitive type. Instead, use the `doubleValue` or `intValue` method.

If the text for the number is not in the correct form, the method throws a `ParseException`. For example, leading whitespace in the string is not allowed. (Call `trim` to remove it.) However, any characters that follow the number in the string are simply ignored, so no exception is thrown.

Note that the classes returned by the `getXXXInstance` factory methods are not actually of type `NumberFormat`. The `NumberFormat` type is an abstract class, and the actual formatters belong to one of its subclasses. The factory methods merely know how to locate the object that belongs to a particular locale.

You can get a list of the currently supported locales with the static `getAvailableLocales` method. That method returns an array of the locales for which number formatter objects can be obtained.

The sample program for this section lets you experiment with number formatters (see [Figure 5-1](#)). The combo box at the top of the figure contains all locales with number formatters. You can choose between number, currency, and percentage formatters. Each time you make another choice, the number in the text field is reformatted. If you go through a few locales, then you get a good impression of how many ways a number or currency value can be formatted. You can also type a different number and click the Parse button to call the `parse` method, which tries to parse what you entered. If your input is successfully parsed, then it is passed to `format` and the result is displayed. If parsing fails, then a "Parse error" message is displayed in the text field.

Figure 5-1. The NumberFormatTest program



The code, shown in [Listing 5-1](#), is fairly straightforward. In the constructor, we call `NumberFormat.getAvailableLocales`. For each locale, we call `getDisplayName`, and we fill a combo box with the strings that the `getDisplayName` method returns. (The strings are not sorted; we tackle this issue in the "[Collation](#)" section beginning on page [318](#).) Whenever the user selects another locale or clicks one of the radio buttons, we create a new formatter object and update the text field. When the user clicks the Parse button, we call the `parse` method to do the actual parsing, based on the locale selected.

Listing 5-1. NumberFormatTest.java

Code View:

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.text.*;
4. import java.util.*;
5.
6. import javax.swing.*;
7.
8. /**
9.  * This program demonstrates formatting numbers under various locales.
10. * @version 1.13 2007-07-25
11. * @author Cay Horstmann
12. */
13. public class NumberFormatTest
14. {
15.     public static void main(String[] args)
16.     {
17.         EventQueue.invokeLater(new Runnable()
18.         {
19.             public void run()
20.             {
21.                 JFrame frame = new NumberFormatFrame();
22.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23.                 frame.setVisible(true);
24.             }
25.         });
26.     }
27. }
28.
29. /**
30. * This frame contains radio buttons to select a number format, a combo box to pick a locale,
31. * a text field to display a formatted number, and a button to parse the text field contents.
32. */
33. class NumberFormatFrame extends JFrame
34. {
35.     public NumberFormatFrame()
36.     {
37.         setLayout(new GridBagLayout());
38.
39.         ActionListener listener = new ActionListener()
40.         {
41.             setTitle("NumberFormatTest");
42.             public void actionPerformed(ActionEvent event)
43.             {
44.                 updateDisplay();
45.             }
46.         };
47.
48.         JPanel p = new JPanel();
49.         addRadioButton(p, numberRadioButton, rbGroup, listener);
50.         addRadioButton(p, currencyRadioButton, rbGroup, listener);
51.         addRadioButton(p, percentRadioButton, rbGroup, listener);
52.
53.         add(new JLabel("Locale:"), new GBC(0, 0).setAnchor(GBC.EAST));
54.         add(p, new GBC(1, 1));
55.         add(parseButton, new GBC(0, 2).setInsets(2));
56.         add(localeCombo, new GBC(1, 0).setAnchor(GBC.WEST));
57.         add(numberText, new GBC(1, 2).setFill(GBC.HORIZONTAL));
58.         locales = (Locale[]) NumberFormat.getAvailableLocales().clone();
59.         Arrays.sort(locales, new Comparator<Locale>()
60.         {
61.             public int compare(Locale l1, Locale l2)
62.             {
63.                 return l1.getDisplayName().compareTo(l2.getDisplayName());
64.             }
65.         });
66.         for (Locale loc : locales)
67.             localeCombo.addItem(loc.getDisplayName());
68.         localeCombo.setSelectedItem(Locale.getDefault().getDisplayName());
69.         currentNumber = 123456.78;
70.         updateDisplay();
```

```
71.     localeCombo.addActionListener(listener);
72.
73.     parseButton.addActionListener(new ActionListener()
74.     {
75.         public void actionPerformed(ActionEvent event)
76.         {
77.             String s = numberText.getText().trim();
78.             try
79.             {
80.                 Number n = currentNumberFormat.parse(s);
81.                 if (n != null)
82.                 {
83.                     currentNumber = n.doubleValue();
84.                     updateDisplay();
85.                 }
86.                 else
87.                 {
88.                     numberText.setText("Parse error: " + s);
89.                 }
90.             }
91.             catch (ParseException e)
92.             {
93.                 numberText.setText("Parse error: " + s);
94.             }
95.         }
96.     });
97.     pack();
98. }
99.
100. /**
101. * Adds a radio button to a container.
102. * @param p the container into which to place the button
103. * @param b the button
104. * @param g the button group
105. * @param listener the button listener
106. */
107. public void addRadioButton(Container p, JRadioButton b, ButtonGroup g,
108.                             ActionListener listener)
109. {
110.     b.setSelected(g.getButtonCount() == 0);
111.     b.addActionListener(listener);
112.     g.add(b);
113.     p.add(b);
114. }
115.
116. /**
117. * Updates the display and formats the number according to the user settings.
118. */
119. public void updateDisplay()
120. {
121.     Locale currentLocale = locales[localeCombo.getSelectedIndex()];
122.     currentNumberFormat = null;
123.     if (numberRadioButton.isSelected()) currentNumberFormat = NumberFormat
124.         .getNumberInstance(currentLocale);
125.     else if (currencyRadioButton.isSelected()) currentNumberFormat = NumberFormat
126.         .getCurrencyInstance(currentLocale);
127.     else if (percentRadioButton.isSelected()) currentNumberFormat = NumberFormat
128.         .getPercentInstance(currentLocale);
129.     String n = currentNumberFormat.format(currentNumber);
130.     numberText.setText(n);
131. }
132.
133.
134. private Locale[] locales;
135. private double currentNumber;
136. private JComboBox localeCombo = new JComboBox();
137. private JButton parseButton = new JButton("Parse");
138. private JTextField numberText = new JTextField(30);
139. private JRadioButton numberRadioButton = new JRadioButton("Number");
140. private JRadioButton currencyRadioButton = new JRadioButton("Currency");
141. private JRadioButton percentRadioButton = new JRadioButton("Percent");
142. private ButtonGroup rbGroup = new ButtonGroup();
143. private NumberFormat currentNumberFormat;
144. }
```

API**java.text.NumberFormat 1.1**

- static Locale[] getAvailableLocales()

returns an array of `Locale` objects for which `NumberFormat` formatters are available.

- static NumberFormat getInstance()

- static NumberFormat getInstance(Locale l)

- static NumberFormat getCurrencyInstance()

- static NumberFormat getCurrencyInstance(Locale l)

- static NumberFormat getPercentInstance()

- static NumberFormat getPercentInstance(Locale l)

returns a formatter for numbers, currency amounts, or percentage values for the current locale or for the given locale.

- String format(double x)

- String format(long x)

returns the string resulting from formatting the given floating-point number or integer.

- Number parse(String s)

parses the given string and returns the number value, as a Double if the input string described a floating-point number, and as a Long otherwise. The beginning of the string must contain a number; no leading whitespace is allowed. The number can be followed by other characters, which are ignored. Throws `ParseException` if parsing was not successful.

- void setParseIntegerOnly(boolean b)

- boolean isParseIntegerOnly()

sets or gets a flag to indicate whether this formatter should parse only integer values.

- void setGroupingUsed(boolean b)

- boolean isGroupingUsed()

sets or gets a flag to indicate whether this formatter emits and recognizes decimal separators (such as 100,000).

- void setMinimumIntegerDigits(int n)

- int getMinimumIntegerDigits()

- void setMaximumIntegerDigits(int n)

- int getMaximumIntegerDigits()

- void setMinimumFractionDigits(int n)

- int getMinimumFractionDigits()

- void setMaximumFractionDigits(int n)

- int getMaximumFractionDigits()

sets or gets the maximum or minimum number of digits allowed in the integer or fractional part of a number.

Currencies

To format a currency value, you can use the `NumberFormat.getCurrencyInstance` method. However, that method is not very flexible—it returns a formatter for a single currency. Suppose you prepare an invoice for an American customer in which some amounts are in dollars and others are in Euros. You can't just use two formatters

Code View:

```
NumberFormat dollarFormatter = NumberFormat.getCurrencyInstance(Locale.US);
NumberFormat euroFormatter = NumberFormat.getCurrencyInstance(Locale.GERMANY);
```

Your invoice would look very strange, with some values formatted like \$100,000 and others like 100.000 €. (Note that the Euro value uses a decimal point, not a comma.)

Instead, use the `Currency` class to control the currency that is used by the formatters. You get a `Currency` object by passing a currency identifier to the static `Currency.getInstance` method. Then call the `setCurrency` method for each formatter. Here is how you would set up the Euro formatter for your American customer:

```
NumberFormat euroFormatter = NumberFormat.getCurrencyInstance(Locale.US);
euroFormatter.setCurrency(Currency.getInstance("EUR"));
```

The currency identifiers are defined by ISO 4217—see <http://www.iso.org/iso/en/prods-services/popstds/currencycodeslist.html>. [Table 5-3](#) provides a partial list.

Table 5-3. Currency Identifiers

Currency Value	Identifier
U.S. Dollar	USD
Euro	EUR
British Pound	GBP
Japanese Yen	JPY
Chinese Renminbi (Yuan)	CNY
Indian Rupee	INR
Russian Ruble	RUB



java.util.Currency 1.4

- `static Currency getInstance(String currencyCode)`
- `static Currency getInstance(Locale locale)`

returns the `Currency` instance for the given ISO 4217 currency code or the country of the given locale.

- `String toString()`
- `String getCurrencyCode()`

gets the ISO 4217 currency code of this currency.

- `String getSymbol()`
- `String getSymbol(Locale locale)`

gets the formatting symbol of this currency for the default locale or the given locale. For example, the symbol for USD can be "\$" or "US\$", depending on the locale.

```
• int getDefaultFractionDigits()  
gets the default number of fraction digits of this currency.
```



Date and Time

When you are formatting date and time, you should be concerned with four locale-dependent issues:

- The names of months and weekdays should be presented in the local language.
- There will be local preferences for the order of year, month, and day.
- The Gregorian calendar might not be the local preference for expressing dates.
- The time zone of the location must be taken into account.

The Java `DateFormat` class handles these issues. It is easy to use and quite similar to the `NumberFormat` class. First, you get a locale. You can use the default locale or call the static `getAvailableLocales` method to obtain an array of locales that support date formatting. Then, you call one of the three factory methods:

```
fmt = DateFormat.getDateInstance(dateStyle, loc);  
fmt = DateFormat.getTimeInstance(timeStyle, loc);  
fmt = DateFormat.getDateTimeInstance(dateStyle, timeStyle, loc);
```

To specify the desired style, these factory methods have a parameter that is one of the following constants:

```
DateFormat.DEFAULT  
  
DateFormat.FULL (e.g., Wednesday, September 12, 2007 8:51:03 PM PDT for the U.S. locale)  
  
DateFormat.LONG (e.g., September 12, 2007 8:51:03 PM PDT for the U.S. locale)  
  
DateFormat.MEDIUM (e.g., Sep 12, 2007 8:51:03 PM for the U.S. locale)  
  
DateFormat.SHORT (e.g., 9/12/07 8:51 PM for the U.S. locale)
```

The factory method returns a formatting object that you can then use to format dates.

```
Date now = new Date();  
String s = fmt.format(now);
```

Just as with the `NumberFormat` class, you can use the `parse` method to parse a date that the user typed. For example, the following code parses the value that the user typed into a text field, using the default locale.

```
TextField inputField;  
. . .  
DateFormat fmt = DateFormat.getDateInstance(DateFormat.MEDIUM);  
Date input = fmt.parse(inputField.getText().trim());
```

Unfortunately, the user must type the date exactly in the expected format. For example, if the format is set to `MEDIUM` in the U.S. locale, then dates are expected to look like

Sep 12, 2007

If the user types

Sep 12 2007

(without the comma) or the short format

9/12/07

then a `ParseException` results.

A `lenient` flag interprets dates leniently. For example, February 30, 2007 will be automatically converted to March 2, 2007. This seems dangerous, but, unfortunately, it is the default. You should probably turn off this feature. The calendar object that interprets the parsed date will throw `IllegalArgumentException` when the user enters an invalid day/month/year combination.

[Listing 5-2](#) shows the `DateFormat` class in action. You can select a locale and see how the date and time are formatted in different places around the world. If you see question-mark characters in the output, then you don't have the fonts installed for displaying characters in the local language. For example, if you pick a Chinese locale, the date might be expressed as

2007 年 9 月 12 日

[Figure 5-2](#) shows the program (after Chinese fonts were installed). As you can see, it correctly displays the output.

Figure 5-2. The `DateFormatTest` program



You can also experiment with parsing. Enter a date or time, click the Parse lenient checkbox if desired, and click the Parse date or Parse time button.

We use a helper class `EnumCombo` to solve a technical problem (see [Listing 5-3](#)). We wanted to fill a combo with values such as `Short`, `Medium`, and `Long` and then automatically convert the user's selection to integer values `DateFormat.SHORT`, `DateFormat.MEDIUM`, and `DateFormat.LONG`. Rather than writing repetitive code, we use reflection: We convert the user's choice to upper case, replace all spaces with underscores, and then find the value of the static field with that name. (See Volume I, Chapter 5 for more details about reflection.)

Tip



To compute times in different time zones, use the `TimeZone` class. See <http://java.sun.com/developer/JDCTechTips/2003/tt1104.html#2> for a brief tutorial.

[Listing 5-2. DateFormatTest.java](#)

Code View:

```

1. import java.awt.*;
2. import java.awt.event.*;
3. import java.text.*;
4. import java.util.*;
5.
6. import javax.swing.*;
7.
8. /**
9.  * This program demonstrates formatting dates under various locales.
10. * @version 1.13 2007-07-25
11. * @author Cay Horstmann
12. */
13. public class DateFormatTest
14. {
15.     public static void main(String[] args)
16.     {
17.         EventQueue.invokeLater(new Runnable()
18.         {
19.             public void run()
20.             {
21.                 JFrame frame = new DateFormatFrame();
22.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
23.                 frame.setVisible(true);
24.             }
25.         });
26.     }
27. }
```

```
28.
29. /**
30.  * This frame contains combo boxes to pick a locale, date and time formats, text fields
31.  * to display formatted date and time, buttons to parse the text field contents, and a
32.  * "lenient" checkbox.
33. */
34. class DateFormatFrame extends JFrame
35. {
36.     public DateFormatFrame()
37.     {
38.         setTitle("DateFormatTest");
39.
40.         setLayout(new GridBagLayout());
41.         add(new JLabel("Locale"), new GBC(0, 0).setAnchor(GBC.EAST));
42.         add(new JLabel("Date style"), new GBC(0, 1).setAnchor(GBC.EAST));
43.         add(new JLabel("Time style"), new GBC(2, 1).setAnchor(GBC.EAST));
44.         add(new JLabel("Date"), new GBC(0, 2).setAnchor(GBC.EAST));
45.         add(new JLabel("Time"), new GBC(0, 3).setAnchor(GBC.EAST));
46.         add(localeCombo, new GBC(1, 0, 2, 1).setAnchor(GBC.WEST));
47.         add(dateStyleCombo, new GBC(1, 1).setAnchor(GBC.WEST));
48.         add(timeStyleCombo, new GBC(3, 1).setAnchor(GBC.WEST));
49.         add(dateParseButton, new GBC(3, 2).setAnchor(GBC.WEST));
50.         add(timeParseButton, new GBC(3, 3).setAnchor(GBC.WEST));
51.         add(lenientCheckbox, new GBC(0, 4, 2, 1).setAnchor(GBC.WEST));
52.         add(dateText, new GBC(1, 2, 2, 1).setFill(GBC.HORIZONTAL));
53.         add(timeText, new GBC(1, 3, 2, 1).setFill(GBC.HORIZONTAL));
54.
55.         locales = (Locale[]) DateFormat.getAvailableLocales().clone();
56.         Arrays.sort(locales, new Comparator<Locale>()
57.         {
58.             public int compare(Locale l1, Locale l2)
59.             {
60.                 return l1.getDisplayName().compareTo(l2.getDisplayName());
61.             }
62.         });
63.         for (Locale loc : locales)
64.             localeCombo.addItem(loc.getDisplayName());
65.         localeCombo.setSelectedItem(Locale.getDefault().getDisplayName());
66.         currentDate = new Date();
67.         currentTime = new Date();
68.         updateDisplay();
69.
70.         ActionListener listener = new ActionListener()
71.         {
72.             public void actionPerformed(ActionEvent event)
73.             {
74.                 updateDisplay();
75.             }
76.         };
77.
78.         localeCombo.addActionListener(listener);
79.         dateStyleCombo.addActionListener(listener);
80.         timeStyleCombo.addActionListener(listener);
81.
82.         dateParseButton.addActionListener(new ActionListener()
83.         {
84.             public void actionPerformed(ActionEvent event)
85.             {
86.                 String d = dateText.getText().trim();
87.                 try
88.                 {
89.                     currentDateFormat.setLenient(lenientCheckbox.isSelected());
90.                     Date date = currentDateFormat.parse(d);
91.                     currentDate = date;
92.                     updateDisplay();
93.                 }
94.                 catch (ParseException e)
95.                 {
96.                     dateText.setText("Parse error: " + d);
97.                 }
98.                 catch (IllegalArgumentException e)
99.                 {
100.                     dateText.setText("Argument error: " + d);
101.                 }
102.             }
103.         });
104.
105.         timeParseButton.addActionListener(new ActionListener()
106.         {
107.             public void actionPerformed(ActionEvent event)
108.             {
109.                 String t = timeText.getText().trim();
```

```
110.         try
111.         {
112.             currentDateFormat.setLenient(lenientCheckbox.isSelected());
113.             Date date = currentTimeFormat.parse(t);
114.             currentTime = date;
115.             updateDisplay();
116.         }
117.     catch (ParseException e)
118.     {
119.         timeText.setText("Parse error: " + t);
120.     }
121.     catch (IllegalArgumentException e)
122.     {
123.         timeText.setText("Argument error: " + t);
124.     }
125. }
126. });
127. pack();
128. }
129.
130. /**
131. * Updates the display and formats the date according to the user settings.
132. */
133. public void updateDisplay()
134. {
135.     Locale currentLocale = locales[localeCombo.getSelectedIndex()];
136.     int dateStyle = dateStyleCombo.getValue();
137.     currentDateFormat = DateFormat.getDateInstance(dateStyle, currentLocale);
138.     String d = currentDateFormat.format(currentDate);
139.     dateText.setText(d);
140.     int timeStyle = timeStyleCombo.getValue();
141.     currentTimeFormat = DateFormat.getTimeInstance(timeStyle, currentLocale);
142.     String t = currentTimeFormat.format(currentTime);
143.     timeText.setText(t);
144. }
145.
146. private Locale[] locales;
147. private Date currentDate;
148. private Date currentTime;
149. private DateFormat currentDateFormat;
150. private DateFormat currentTimeFormat;
151. private JComboBox localeCombo = new JComboBox();
152. private EnumCombo dateStyleCombo = new EnumCombo(DateFormat.class, new String[] { "Default",
153.     "Full", "Long", "Medium", "Short" });
154. private EnumCombo timeStyleCombo = new EnumCombo(DateFormat.class, new String[] { "Default",
155.     "Full", "Long", "Medium", "Short" });
156. private JButton dateParseButton = new JButton("Parse date");
157. private JButton timeParseButton = new JButton("Parse time");
158. private JTextField dateText = new JTextField(30);
159. private JTextField timeText = new JTextField(30);
160. private JCheckBox lenientCheckbox = new JCheckBox("Parse lenient", true);
161. }
```

Listing 5-3. `EnumCombo.java`

Code View:

```
1. import java.util.*;
2. import javax.swing.*;
3.
4. /**
5.  * A combo box that lets users choose from among static field
6.  * values whose names are given in the constructor.
7.  * @version 1.13 2007-07-25
8.  * @author Cay Horstmann
9. */
10. public class EnumCombo extends JComboBox
11. {
12.     /**
13.      Constructs an EnumCombo.
14.      @param cl a class
15.      @param labels an array of static field names of cl
16.  */
```

```
17.     public EnumCombo(Class<?> cl, String[] labels)
18.     {
19.         for (String label : labels)
20.         {
21.             String name = label.toUpperCase().replace(' ', '_');
22.             int value = 0;
23.             try
24.             {
25.                 java.lang.reflect.Field f = cl.getField(name);
26.                 value = f.getInt(cl);
27.             }
28.             catch (Exception e)
29.             {
30.                 label = "(" + label + ")";
31.             }
32.             table.put(label, value);
33.             addItem(label);
34.         }
35.         setSelectedItem(labels[0]);
36.     }
37.
38.    /**
39.     * Returns the value of the field that the user selected.
40.     * @return the static field value
41.    */
42.    public int getValue()
43.    {
44.        return table.get(getSelectedItem());
45.    }
46.
47.    private Map<String, Integer> table = new TreeMap<String, Integer>();
48. }
```

API**java.text.DateFormat 1.1**

- static Locale[] getAvailableLocales()
- returns an array of `Locale` objects for which `DateFormat` formatters are available.
- static DateFormat getDateInstance(int dateStyle)
 - static DateFormat getDateInstance(int dateStyle, Locale l)
 - static DateFormat getTimeInstance(int timeStyle)
 - static DateFormat getTimeInstance(int timeStyle, Locale l)
 - static DateFormat getDateTimeInstance(int dateStyle, int timeStyle)
 - static DateFormat getDateTimeInstance(int dateStyle, int timeStyle, Locale l)

returns a formatter for date, time, or date and time for the default locale or the given locale.

Parameters: `dateStyle, timeStyle` One of `DEFAULT, FULL, LONG, MEDIUM, SHORT`

- String format(Date d)
- returns the string resulting from formatting the given date/time.
- Date parse(String s)
- parses the given string and returns the date/time described in it. The beginning of the

string must contain a date or time; no leading whitespace is allowed. The date can be followed by other characters, which are ignored. Throws a `ParseException` if parsing was not successful.

- `void setLenient(boolean b)`
- `boolean isLenient()`

sets or gets a flag to indicate whether parsing should be lenient or strict. In lenient mode, dates such as `February 30, 1999` will be automatically converted to `March 2, 1999`. The default is lenient mode.

- `void setCalendar(Calendar cal)`
- `Calendar getCalendar()`

sets or gets the calendar object used for extracting year, month, day, hour, minute, and second from the `Date` object. Use this method if you do not want to use the default calendar for the locale (usually the Gregorian calendar).

- `void setTimeZone(TimeZone tz)`
- `TimeZone getTimeZone()`

sets or gets the time zone object used for formatting the time. Use this method if you do not want to use the default time zone for the locale. The default time zone is the time zone of the default locale, as obtained from the operating system. For the other locales, it is the preferred time zone in the geographical location.

- `void setNumberFormat(NumberFormat f)`
- `NumberFormat getNumberFormat()`

sets or gets the number format used for formatting the numbers used for representing year, month, day, hour, minute, and second.



java.util.TimeZone 1.1

- `static String[] getAvailableIDs()`
gets all supported time zone IDs.
- `static TimeZone getDefault()`
gets the default `TimeZone` for this computer.
- `static TimeZone getTimeZone(String timeZoneID)`
gets the `TimeZone` for the given ID.
- `StringgetID()`
gets the ID of this time zone.
- `StringgetDisplayName()`
- `StringgetDisplayName(Locale locale)`
- `StringgetDisplayName(boolean daylight, int style)`
- `StringgetDisplayName(boolean daylight, int style, Locale locale)`
gets the display name of this time zone in the default locale or in the given locale. If the `daylight` parameter is true, the daylight-savings name is returned. The `style`

parameter can be `SHORT` or `LONG`.

- `boolean useDaylightTime()`
returns `true` if this `TimeZone` uses daylight-savings time.
- `boolean inDaylightTime(Date date)`
returns `true` if the given date is in daylight-savings time in this `TimeZone`.



Collation

Most programmers know how to compare strings with the `compareTo` method of the `String` class. The value of `a.compareTo(b)` is a negative number if `a` is lexicographically less than `b`, zero if they are identical, and positive otherwise.

Unfortunately, unless all your words are in uppercase English ASCII characters, this method is useless. The problem is that the `compareTo` method in the Java programming language uses the values of the Unicode character to determine the ordering. For example, lowercase characters have a higher Unicode value than do uppercase characters, and accented characters have even higher values. This leads to absurd results; for example, the following five strings are ordered according to the `compareTo` method:

```
America  
Zulu  
able  
zebra  
Ångström
```

For dictionary ordering, you want to consider upper case and lower case to be equivalent. To an English speaker, the sample list of words would be ordered as

```
able  
America  
Ångström  
zebra  
Zulu
```

However, that order would not be acceptable to a Swedish user. In Swedish, the letter Å is different from the letter A, and it is collated after the letter Z! That is, a Swedish user would want the words to be sorted as

```
able  
America  
zebra  
Zulu  
Ångström
```

Fortunately, once you are aware of the problem, collation is quite easy. As always, you start by obtaining a `Locale` object. Then, you call the `getInstance` factory method to obtain a `Collator` object. Finally, you use the `compare` method of the `collator`, not the `compareTo` method of the `String` class, whenever you want to sort strings.

```
Locale loc = . . .;  
Collator coll = Collator.getInstance(loc);  
if (coll.compare(a, b) < 0) // a comes before b . . .;
```

Most important, the `Collator` class implements the `Comparator` interface. Therefore, you can pass a `collator` object to the `Collections.sort` method to sort a list of strings:

```
Collections.sort(strings, coll);
```

Collation Strength

You can set a collator's strength to select how selective it should be. Character differences are classified as primary, secondary,

tertiary, and identical. For example, in English, the difference between "A" and "Z" is considered primary, the difference between "A" and "Å" is secondary, and between "A" and "a" is tertiary.

By setting the strength of the collator to `Collator.PRIMARY`, you tell it to pay attention only to primary differences. By setting the strength to `Collator.SECONDARY`, you instruct the collator to take secondary differences into account. That is, two strings will be more likely to be considered different when the strength is set to "secondary" or "tertiary," as shown in [Table 5-4](#).

Table 5-4. Collations with Different Strengths (English Locale)

Primary	Secondary	Tertiary
Angstrom = Ångström	Angstrom \neq Ångström	Angstrom \neq Ångström
Able = able	Able = able	Able \neq able

When the strength has been set to `Collator.IDENTICAL`, no differences are allowed. This setting is mainly useful in conjunction with the second, rather technical, collator setting, the decomposition mode, which we discuss in the next section.

Decomposition

Occasionally, a character or sequence of characters can be described in more than one way in Unicode. For example, an "Å" can be Unicode character U+00C5, or it can be expressed as a plain A (U+0065) followed by a ° ("combining ring above"; U+030A). Perhaps more surprisingly, the letter sequence "ffi" can be described with a single character "Latin small ligature ffi" with code U+FB03. (One could argue that this is a presentation issue and it should not have resulted in different Unicode characters, but we don't make the rules.)

The Unicode standard defines four normalization forms (D, KD, C, and KC) for strings. See <http://www.unicode.org/unicode/reports/tr15/tr15-23.html> for the details. Two of them are used for collation. In normalization form D, accented characters are decomposed into their base letters and combining accents. For example, Å is turned into a sequence of an A and a combining ring above °. Normalization form KD goes further and decomposes compatibility characters such as the ffi ligature or the trademark symbol ™.

You choose the degree of normalization that you want the collator to use. The value `Collator.NO_DECOMPOSITION` does not normalize strings at all. This option is faster, but it might not be appropriate for text that expresses characters in multiple forms. The default, `Collator.CANONICAL_DECOMPOSITION`, uses normalization form D. This is the most useful form for text that contains accents but not ligatures. Finally, "full decomposition" uses normalization form KD. See [Table 5-5](#) for examples.

Table 5-5. Differences Between Decomposition Modes

No Decomposition	Canonical Decomposition	Full Decomposition
Å \neq A°	Å = A°	Å = A°
™ \neq TM	™ \neq TM	™ = TM

It is wasteful to have the collator decompose a string many times. If one string is compared many times against other strings, then you can save the decomposition in a collation key object. The `getCollationKey` method returns a `CollationKey` object that you can use for further, faster comparisons. Here is an example:

```
String a = . . .;
CollationKey aKey = coll.getCollationKey(a);
if(aKey.compareTo(coll.getCollationKey(b)) == 0) // fast comparison
    . . .
```

Finally, you might want to convert strings into their normalized forms even when you don't do collation; for example, when storing strings in a database or communicating with another program. As of Java SE 6, the `java.text.Normalizer` class carries out the normalization process. For example,

Code View:

```
String name = "Ångström";
String normalized = Normalizer.normalize(name, Normalizer.Form.NFD); // uses normalization form D
```

The normalized string contains ten characters. The "Å" and "ö" are replaced by "A°" and "o°" sequences.

However, that is not usually the best form for storage and transmission. Normalization form C first applies decomposition and then combines the accents back in a standardized order. According to the W3C, this is the recommended mode for transferring data over the Internet.

The program in [Listing 5-4](#) lets you experiment with collation order. Type a word into the text field and click the Add button to add it to the list of words. Each time you add another word, or change the locale, strength, or decomposition mode, the list of words is sorted again. An = sign indicates words that are considered identical (see [Figure 5-3](#)).

Figure 5-3. The CollationTest program



The locale names in the combo box are displayed in sorted order, using the collator of the default locale. If you run this program with the US English locale, note that "Norwegian (Norway,Nynorsk)" comes before "Norwegian (Norway)", even though the Unicode value of the comma character is greater than the Unicode value of the closing parenthesis.

Listing 5-4. CollationTest.java

Code View:

```

1. import java.awt.*;
2. import java.awt.event.*;
3. import java.text.*;
4. import java.util.*;
5. import java.util.List;
6.
7. import javax.swing.*;
8.
9. /**
10. * This program demonstrates collating strings under various locales.
11. * @version 1.13 2007-07-25
12. * @author Cay Horstmann
13. */
14. public class CollationTest
15. {
16.     public static void main(String[] args)
17.     {
18.         EventQueue.invokeLater(new Runnable()
19.         {
20.             public void run()
21.             {
22.
23.                 JFrame frame = new CollationFrame();
24.                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
25.                 frame.setVisible(true);
26.             }
27.         });
28.     }
29. }
```

```
31. /**
32. * This frame contains combo boxes to pick a locale, collation strength and decomposition
33. * rules, a text field and button to add new strings, and a text area to list the
34. * collated strings.
35. */
36. class CollationFrame extends JFrame
37. {
38.     public CollationFrame()
39.     {
40.         setTitle("CollationTest");
41.
42.         setLayout(new GridBagLayout());
43.         add(new JLabel("Locale"), new GBC(0, 0).setAnchor(GBC.EAST));
44.         add(new JLabel("Strength"), new GBC(0, 1).setAnchor(GBC.EAST));
45.         add(new JLabel("Decomposition"), new GBC(0, 2).setAnchor(GBC.EAST));
46.         add(addButton, new GBC(0, 3).setAnchor(GBC.EAST));
47.         add(localeCombo, new GBC(1, 0).setAnchor(GBC.WEST));
48.         add(strengthCombo, new GBC(1, 1).setAnchor(GBC.WEST));
49.         add(decompositionCombo, new GBC(1, 2).setAnchor(GBC.WEST));
50.         add(newWord, new GBC(1, 3).setFill(GBC.HORIZONTAL));
51.         add(new JScrollPane(sortedWords), new GBC(0, 4, 2, 1).setFill(GBC.BOTH));
52.
53.         locales = (Locale[]) Collator.getAvailableLocales().clone();
54.         Arrays.sort(locales, new Comparator<Locale>()
55.         {
56.             private Collator collator = Collator.getInstance(Locale.getDefault());
57.
58.             public int compare(Locale l1, Locale l2)
59.             {
60.                 return collator.compare(l1.getDisplayName(), l2.getDisplayName());
61.             }
62.         });
63.         for (Locale loc : locales)
64.             localeCombo.addItem(loc.getDisplayName());
65.         localeCombo.setSelectedItem(Locale.getDefault().getDisplayName());
66.
67.         strings.add("America");
68.         strings.add("able");
69.         strings.add("Zulu");
70.         strings.add("zebra");
71.         strings.add("\u00C5ngstr\u00F6m");
72.         strings.add("A\u030angstro\u0308m");
73.         strings.add("Angstrom");
74.         strings.add("Able");
75.         strings.add("office");
76.         strings.add("\u030ce");
77.         strings.add("Java\u2122");
78.         strings.add("JavaTM");
79.         updateDisplay();
80.
81.         addButton.addActionListener(new ActionListener()
82.         {
83.             public void actionPerformed(ActionEvent event)
84.             {
85.                 strings.add(newWord.getText());
86.                 updateDisplay();
87.             }
88.         });
89.
90.         ActionListener listener = new ActionListener()
91.         {
92.             public void actionPerformed(ActionEvent event)
93.             {
94.                 updateDisplay();
95.             }
96.         };
97.
98.         localeCombo.addActionListener(listener);
99.         strengthCombo.addActionListener(listener);
100.        decompositionCombo.addActionListener(listener);
101.        pack();
102.    }
103.
104. /**
105. * Updates the display and collates the strings according to the user settings.
106. */
107. public void updateDisplay()
108. {
109.     Locale currentLocale = locales[localeCombo.getSelectedIndex()];
110.     localeCombo.setLocale(currentLocale);
111.
112.     currentCollator = Collator.getInstance(currentLocale);
```

```
113.     currentCollator.setStrength(strengthCombo.getValue());
114.     currentCollator.setDecomposition(decompositionCombo.getValue());
115.
116.     Collections.sort(strings, currentCollator);
117.
118.     sortedWords.setText("");
119.     for (int i = 0; i < strings.size(); i++)
120.     {
121.         String s = strings.get(i);
122.         if (i > 0 && currentCollator.compare(s, strings.get(i - 1)) == 0) sortedWords
123.             .append("= ");
124.         sortedWords.append(s + "\n");
125.     }
126.     pack();
127. }
128.
129. private List<String> strings = new ArrayList<String>();
130. private Collator currentCollator;
131. private Locale[] locales;
132. private JComboBox localeCombo = new JComboBox();
133.
134. private EnumCombo strengthCombo = new EnumCombo(Collator.class, new String[] { "Primary",
135.     "Secondary", "Tertiary", "Identical" });
136. private EnumCombo decompositionCombo = new EnumCombo(Collator.class, new String[] {
137.     "Canonical Decomposition", "Full Decomposition", "No Decomposition" });
138. private JTextField newWord = new JTextField(20);
139. private JTextArea sortedWords = new JTextArea(20, 20);
140. private JButton addButton = new JButton("Add");
141. }
```



java.text.Collator 1.1

- static Locale[] getAvailableLocales()
returns an array of `Locale` objects for which `Collator` objects are available.
- static Collator getInstance()
- static Collator getInstance(Locale l)
returns a collator for the default locale or the given locale.
- int compare(String a, String b)
returns a negative value if `a` comes before `b`, 0 if they are considered identical, and a positive value otherwise.
- boolean equals(String a, String b)
returns `true` if they are considered identical, `false` otherwise.
- void setStrength(int strength)
- int getStrength()
sets or gets the strength of the collator. Stronger collators tell more words apart.
Strength values are `Collator.PRIMARY`, `Collator.SECONDARY`, and
`Collator.TERTIARY`.
- void setDecomposition(int decomp)
- int getDecompositon()
sets or gets the decomposition mode of the collator. The more a collator decomposes a string, the more strict it will be in deciding whether two strings should be considered

identical. Decomposition values are Collator.NO_DECOMPOSITION, Collator.CANONICAL_DECOMPOSITION, and Collator.FULL_DECOMPOSITION.

- CollationKey getCollationKey(String a)

returns a collation key that contains a decomposition of the characters in a form that can be quickly compared against another collation key.

API

java.text.CollationKey 1.1

- int compareTo(CollationKey b)

returns a negative value if this key comes before b, 0 if they are considered identical, and a positive value otherwise.

API

java.text.Normalizer 6

- static String normalize(CharSequence str, Normalizer.Form form)

returns the normalized form of str. The form value is one of ND, NKD, NC, or NKC.



Message Formatting

The Java library has a `MessageFormat` class that formats text with variable parts, like this:

```
"On {2}, a {0} destroyed {1} houses and caused {3} of damage."
```

The numbers in braces are placeholders for actual names and values. The static method `MessageFormat.format` lets you substitute values for the variables. As of JDK 5.0, it is a "varargs" method, so you can simply supply the parameters as follows:

Code View:

```
String msg = MessageFormat.format("On {2}, a {0} destroyed {1} houses and caused {3} of damage.",
    "hurricane", 99, new GregorianCalendar(1999, 0, 1).getTime(), 10.0E8);
```

In this example, the placeholder {0} is replaced with "hurricane", {1} is replaced with 99, and so on.

The result of our example is the string

Code View:

```
On 1/1/99 12:00 AM, a hurricane destroyed 99 houses and caused 100,000,000 of damage.
```

That is a start, but it is not perfect. We don't want to display the time "12:00 AM," and we want the damage amount printed as a currency value. The way we do this is by supplying an optional format for some of the placeholders:

Code View:

```
"On {2,date,long}, a {0} destroyed {1} houses and caused {3,number,currency} of damage."
```

This example code prints:

Code View:

```
On January 1, 1999, a hurricane destroyed 99 houses and caused $100,000,000 of damage.
```

In general, the placeholder index can be followed by a type and a style. Separate the index, type, and style by commas. The type can be any of

```
number  
time  
date  
choice
```

If the type is `number`, then the style can be

```
integer  
currency  
percent
```

or it can be a number format pattern such as `$,##0`. (See the documentation of the `DecimalFormat` class for more information about the possible formats.)

If the type is either `time` or `date`, then the style can be

```
short  
medium  
long  
full
```

or a date format pattern such as `yyyy-MM-dd`. (See the documentation of the `SimpleDateFormat` class for more information about the possible formats.)

Choice formats are more complex, and we take them up in the next section.

Caution



The static `MessageFormat.format` method uses the current locale to format the values. To format with an arbitrary locale, you have to work a bit harder because there is no "varargs" method that you can use. You need to place the values to be formatted into an `Object[]` array, like this:

```
MessageFormat mf = new MessageFormat(pattern, loc);
String msg = mf.format(new Object[] { values });
```



java.text.MessageFormat 1.1

- `MessageFormat(String pattern)`
- `MessageFormat(String pattern, Locale loc)`

constructs a message format object with the specified pattern and locale.

- `void applyPattern(String pattern)`

sets the pattern of a message format object to the specified pattern.

- `void setLocale(Locale loc)`

- `Locale getLocale()`

sets or gets the locale to be used for the placeholders in the message. The locale is only used for subsequent patterns that you set by calling the `applyPattern` method.

- `static String format(String pattern, Object... args)`
formats the pattern string by using `args[i]` as input for placeholder `{i}`.
- `StringBuffer format(Object args, StringBuffer result, FieldPosition pos)`
formats the pattern of this `MessageFormat`. The `args` parameter must be an array of objects. The formatted string is appended to `result`, and `result` is returned. If `pos` equals `new FieldPosition(MessageFormat.Field.ARGUMENT)`, its `beginIndex` and `endIndex` properties are set to the location of the text that replaces the `{1}` placeholder. Supply `null` if you are not interested in position information.



java.text.Format 1.1

- `String format(Object obj)`
formats the given object, according to the rules of this formatter. This method calls `format(obj, new StringBuffer(), new FieldPosition(1)).toString()`.

Choice Formats

Let's look closer at the pattern of the preceding section:

```
"On {2}, a {0} destroyed {1} houses and caused {3} of damage."
```

If we replace the disaster placeholder `{0}` with "earthquake", then the sentence is not grammatically correct in English.

```
On January 1, 1999, a earthquake destroyed . . .
```

That means what we really want to do is integrate the article "a" into the placeholder:

```
"On {2}, {0} destroyed {1} houses and caused {3} of damage."
```

The `{0}` would then be replaced with "a hurricane" or "an earthquake". That is especially appropriate if this message needs to be translated into a language where the gender of a word affects the article. For example, in German, the pattern would be

```
"{0} zerstörte am {2} {1} Häuser und richtete einen Schaden von {3} an."
```

The placeholder would then be replaced with the grammatically correct combination of article and noun, such as "Ein Wirbelsturm", "Eine Naturkatastrophe".

Now let us turn to the `{1}` parameter. If the disaster isn't all that catastrophic, then `{1}` might be replaced with the number 1, and the message would read:

```
On January 1, 1999, a mudslide destroyed 1 houses and . . .
```

We would ideally like the message to vary according to the placeholder value, so that it can read

```
no houses
one house
2 houses
. . .
```

depending on the placeholder value. The `choice` formatting option was designed for this purpose.

A choice format is a sequence of pairs, each of which contains

- A lower limit
- A format string

The lower limit and format string are separated by a # character, and the pairs are separated by | characters.

For example,

```
{1,choice,0#no houses|1#one house|2#{1} houses}
```

[Table 5-6](#) shows the effect of this format string for various values of {1}.

Table 5-6. String Formatted by Choice Format

{1}	Result
0	"no houses"
1	"one house"
3	"3 houses"
-1	"no houses"

Why do we use {1} twice in the format string? When the message format applies the choice format on the {1} placeholder and the value is \$2, the choice format returns "{1} houses". That string is then formatted again by the message format, and the answer is spliced into the result.

Note

 This example shows that the designer of the choice format was a bit muddleheaded. If you have three format strings, you need two limits to separate them. In general, you need one fewer limit than you have format strings. As you saw in [Table 5-4](#), the `MessageFormat` class ignores the first limit.

The syntax would have been a lot clearer if the designer of this class realized that the limits belong between the choices, such as

```
no houses|1|one house|2|{1} houses // not the actual format
```

You can use the < symbol to denote that a choice should be selected if the lower bound is strictly less than the value.

You can also use the \leq symbol (expressed as the Unicode character code \u2264) as a synonym for #. If you like, you can even specify a lower bound of -\u221E as -\u221E for the first value.

For example,

```
-\u221E<no houses|0<one house|2\leq{1} houses
```

or, using Unicode escapes,

```
-\u221E<no houses|0<one house|2\u2264{1} houses
```

Let's finish our natural disaster scenario. If we put the choice string inside the original message string, we get the following format instruction:

Code View:

```
String pattern = "On {2,date,long}, {0} destroyed {1,choice,0#no houses|1#one house|2#{1} houses}" + "and caused {3,number,currency} of damage.;"
```

Or, in German,

Code View:

```
String pattern = "{0} zerstörte am {2,date,long} {1,choice,0#kein Haus|1#ein Haus|2#{1} Häuser}"
+ "und richtete einen Schaden von {3,number,currency} an.";
```

Note that the ordering of the words is different in German, but the array of objects you pass to the `format` method is the same. The order of the placeholders in the format string takes care of the changes in the word ordering.



Text Files and Character Sets

As you know, the Java programming language itself is fully Unicode based. However, operating systems typically have their own character encoding, such as ISO-8859 -1 (an 8-bit code sometimes called the "ANSI" code) in the United States, or Big5 in Taiwan.

When you save data to a text file, you should respect the local character encoding so that the users of your program can open the text file with their other applications. Specify the character encoding in the `FileWriter` constructor:

```
out = new FileWriter(filename, "ISO-8859-1");
```

You can find a complete list of the supported encodings in Volume I, Chapter 12.

Unfortunately, there is currently no connection between locales and character encodings. For example, if your user has selected the Taiwanese locale `zh_TW`, no method in the Java programming language tells you that the Big5 character encoding would be the most appropriate.

Character Encoding of Source Files

It is worth keeping in mind that you, the programmer, will need to communicate with the Java compiler. And you do that with tools on your local system. For example, you can use the Chinese version of Notepad to write your Java source code files. The resulting source code files are not portable because they use the local character encoding (GB or Big5, depending on which Chinese operating system you use). Only the compiled class files are portable—they will automatically use the "modified UTF-8" encoding for identifiers and strings. That means that even when a program is compiling and running, three character encodings are involved:

- Source files: local encoding
- Class files: modified UTF-8
- Virtual machine: UTF-16

(See Volume I, Chapter 12 for a definition of the modified UTF-8 and UTF-16 formats.)

Tip

You can specify the character encoding of your source files with the `-encoding` flag, for example,

```
javac -encoding Big5 Myfile.java
```

To make your source files portable, restrict yourself to using the plain ASCII encoding. That is, you should change all non-ASCII characters to their equivalent Unicode encodings. For example, rather than using the string "`Häuser`", use "`H\u0084user`". The JDK contains a utility, `native2ascii`, that you can use to convert the native character encoding to plain ASCII. This utility simply replaces every non-ASCII character in the input with a `\u` followed by the four hex digits of the Unicode value. To use the `native2ascii` program, provide the input and output file names.

```
native2ascii Myfile.java Myfile.temp
```

You can convert the other way with the `-reverse` option:

```
native2ascii -reverse Myfile.temp Myfile.java
```

You can specify another encoding with the `-encoding` option. The encoding name must be one of those listed in the encodings table in Volume I, Chapter 12.

```
native2ascii -encoding Big5 Myfile.java Myfile.temp
```

Tip



It is a good idea to restrict yourself to plain ASCII class names. Because the name of the class also turns into the name of the class file, you are at the mercy of the local file system to handle any non-ASCII coded names. Here is a depressing example. Windows 95 used the so-called Code Page 437 or original PC encoding, for its file names. If you compiled a class `Bär` and tried to run it in Windows 95, you got an error message "cannot find class `BΣr`".



Resource Bundles

When localizing an application, you'll probably have a dauntingly large number of message strings, button labels, and so on, that all need to be translated. To make this task feasible, you'll want to define the message strings in an external location, usually called a resource. The person carrying out the translation can then simply edit the resource files without having to touch the source code of the program.

In Java, you use property files to specify string resources, and you implement classes for resources of other types.

Note



Java technology resources are not the same as Windows or Macintosh resources. A Macintosh or Windows executable program stores resources such as menus, dialog boxes, icons, and messages in a section separate from the program code. A resource editor can inspect and update these resources without affecting the program code.

Note



Volume I, Chapter 10 describes a concept of JAR file resources, whereby data files, sounds, and images can be placed in a JAR file. The `getResource` method of the class `Class` finds the file, opens it, and returns a URL to the resource. By placing the files into the JAR file, you leave the job of finding the files to the class loader, which already knows how to locate items in a JAR file. However, that mechanism has no locale support.

Locating Resource Bundles

When localizing an application, you produce a set of resource bundles. Each bundle is a property file or a class that describes locale-specific items (such as messages, labels, and so on). For each bundle, you provide versions for all locales that you want to support.

You need to use a specific naming convention for these bundles. For example, resources specific for Germany go to a file `bundleName_de_DE`, whereas those that are shared by all German-speaking countries go into `bundleName_de`. In general, use

```
bundleName_language_country
```

for all country-specific resources, and use

```
bundleName_language
```

for all language-specific resources. Finally, as a fallback, you can put defaults into a file without any suffix.

You load a bundle with the command

Code View:

```
 ResourceBundle currentResources = ResourceBundle.getBundle(bundleName, currentLocale);
```

The `getBundle` method attempts to load the bundle that matches the current locale by language, country, and variant. If it is not successful, then the variant, country, and language are dropped in turn. Then the same search is applied to the default locale, and finally, the default bundle file is consulted. If even that attempt fails, the method throws a `MissingResourceException`.

That is, the `getBundle` method tries to load the following bundles:

```
bundleName_currentLocaleLanguage_currentLocaleCountry_currentLocaleVariant
bundleName_currentLocaleLanguage_currentLocaleCountry
bundleName_currentLocaleLanguage
bundleName_defaultLocaleLanguage_defaultLocaleCountry_defaultLocaleVariant
bundleName_defaultLocaleLanguage_defaultLocaleCountry
bundleName_defaultLocaleLanguage
bundleName
```

Once the `getBundle` method has located a bundle, say, `bundleName_de_DE`, it will still keep looking for `bundleName_de` and `bundleName`. If these bundles exist, they become the parents of the `bundleName_de_DE` bundle in a resource hierarchy. Later, when looking up a resource, the parents are searched if a lookup was not successful in the current bundle. That is, if a particular resource was not found in `bundleName_de_DE`, then the `bundleName_de` and `bundleName` will be queried as well.

This is clearly a very useful service and one that would be tedious to program by hand. The resource bundle mechanism of the Java programming language automatically locates the items that are the best match for a given locale. It is easy to add more and more localizations to an existing program: All you have to do is add additional resource bundles.

Tip

 You need not place all resources for your application into a single bundle. You could have one bundle for button labels, one for error messages, and so on.

Property Files

Internationalizing strings is quite straightforward. You place all your strings into a property file such as `MyProgramStrings.properties`. This is simply a text file with one key/value pair per line. A typical file would look like this:

```
computeButton=Rechnen
colorName=black
defaultPaperSize=210x297
```

Then you name your property files as described in the preceding section, for example:

```
MyProgramStrings.properties
MyProgramStrings_en.properties
MyProgramStrings_de_DE.properties
```

You can load the bundle simply as

Code View:

```
 ResourceBundle bundle = ResourceBundle.getBundle("MyProgramStrings", locale);
```

To look up a specific string, call

```
String computeButtonLabel = bundle.getString("computeButton");
```

Caution

 Files for storing properties are always ASCII files. If you need to place Unicode characters into a properties file, encode them by using the `\uxxxx` encoding. For example, to specify "colorName=Grün", use

```
colorName=Gr\u00FCn
```

You can use the `native2ascii` tool to generate these files.

Bundle Classes

To provide resources that are not strings, you define classes that extend the `ResourceBundle` class. You use the standard naming convention to name your classes, for example

```
MyProgramResources.java  
MyProgramResources_en.java  
MyProgramResources_de_DE.java
```

You load the class with the same `getBundle` method that you use to load a property file:

Code View:

```
 ResourceBundle bundle = ResourceBundle.getBundle("MyProgramResources", locale);
```

Caution



When searching for bundles, a bundle in a class is given preference over a property file when the two bundles have the same base names.

Each resource bundle class implements a lookup table. You provide a key string for each setting you want to localize, and you use that key string to retrieve the setting. For example,

```
Color backgroundColor = (Color) bundle.getObject("backgroundColor");  
double[] paperSize = (double[]) bundle.getObject("defaultPaperSize");
```

The simplest way of implementing resource bundle classes is to extend the `ListResourceBundle` class. The `ListResourceBundle` lets you place all your resources into an object array and then does the lookup for you. Follow this code outline:

```
public class bundleName_language_country extends ListResourceBundle  
{  
    public Object[][] getContents() { return contents; }  
    private static final Object[][] contents =  
    {  
        { "key1", value1 },  
        { "key2", value2 },  
        . . .  
    }  
}
```

For example,

```
public class ProgramResources_de extends ListResourceBundle  
{  
    public Object[][] getContents() { return contents; }  
    private static final Object[][] contents =  
    {  
        { "backgroundColor", Color.black },  
        { "defaultPaperSize", new double[] { 210, 297 } }  
    }  
}  
  
public class ProgramResources_en_US extends ListResourceBundle  
{  
    public Object[][] getContents() { return contents; }  
    private static final Object[][] contents =  
    {  
        { "backgroundColor", Color.blue },  
        { "defaultPaperSize", new double[] { 216, 279 } }  
    }  
}
```

}

Note

The paper sizes are given in millimeters. Everyone on the planet, with the exception of the United States and Canada, uses ISO 216 paper sizes. For more information, see <http://www.cl.cam.ac.uk/~mgk25/iso-paper.html>. According to the U.S. Metric Association (<http://lamar.colostate.edu/~hillger>), only three countries in the world have not yet officially adopted the metric system: Liberia, Myanmar (Burma), and the United States of America.

Alternatively, your resource bundle classes can extend the `ResourceBundle` class. Then you need to implement two methods, to enumerate all keys and to look up the value for a given key:

```
Enumeration<String> getKeys()  
Object handleGetObject(String key)
```

The `getObject` method of the `ResourceBundle` class calls the `handleGetObject` method that you supply.

Note

As of Java SE 6, you can choose alternate mechanisms for storing your resources. For example, you can customize the resource loading mechanism to fetch resources from XML files or databases. See http://java.sun.com/developer/technicalArticles/javase/i18n_enhance for more information.

**java.util.ResourceBundle 1.1**

- `static ResourceBundle getBundle(String baseName, Locale loc)`

loads the resource bundle class with the given name, for the given locale or the default locale, and its parent classes. If the resource bundle classes are located in a package, then the base name must contain the full package name, such as "intl.ProgramResources". The resource bundle classes must be `public` so that the `getBundle` method can access them.
- `Object getObject(String name)`

looks up an object from the resource bundle or its parents.
- `String getString(String name)`

looks up an object from the resource bundle or its parents and casts it as a string.
- `String[] getStringArray(String name)`

looks up an object from the resource bundle or its parents and casts it as a string array.
- `Enumeration<String> getKeys()`

returns an enumeration object to enumerate the keys of this resource bundle. It enumerates the keys in the parent bundles as well.
- `Object handleGetObject(String key)`

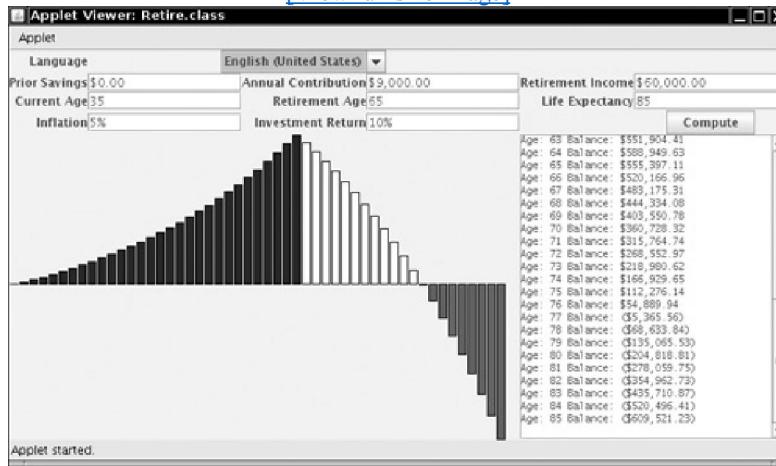
should be overridden to look up the resource value associated with the given key if you define your own resource lookup mechanism.

**A Complete Example**

In this section, we apply the material from this chapter to localize a retirement calculator applet. The applet calculates whether or not you are saving enough money for your retirement. You enter your age, how much money you save every month, and so on (see [Figure 5-4](#)).

Figure 5-4. The retirement calculator in English

[View full size image]



The text area and the graph show the balance of the retirement account for every year. If the numbers turn negative toward the later part of your life and the bars in the graph appear below the x-axis, you need to do something; for example, save more money, postpone your retirement, die earlier, or be younger.

The retirement calculator works in three locales (English, German, and Chinese). Here are some of the highlights of the internationalization:

- The labels, buttons, and messages are translated into German and Chinese. You can find them in the classes `RetireResources_de`, `RetireResources_zh`. English is used as the fallback—see the `RetireResources` file. To generate the Chinese messages, we first typed the file, using Notepad running in Chinese Windows, and then we used the `native2ascii` utility to convert the characters to Unicode.
- Whenever the locale changes, we reset the labels and reformat the contents of the text fields.
- The text fields handle numbers, currency amounts, and percentages in the local format.
- The computation field uses a `MessageFormat`. The format string is stored in the resource bundle of each language.
- Just to show that it can be done, we use different colors for the bar graph, depending on the language chosen by the user.

[Listings 5-5](#) through [5-8](#) show the code. [Listings 5-9](#) through [5-11](#) are the property files for the localized strings. [Figures 5-5](#) and [5-6](#) show the outputs in German and Chinese, respectively. To see Chinese characters, be sure you have Chinese fonts installed and configured with your Java runtime. Otherwise, all Chinese characters show up as "missing character" icons.

Figure 5-5. The retirement calculator in German

[View full size image]

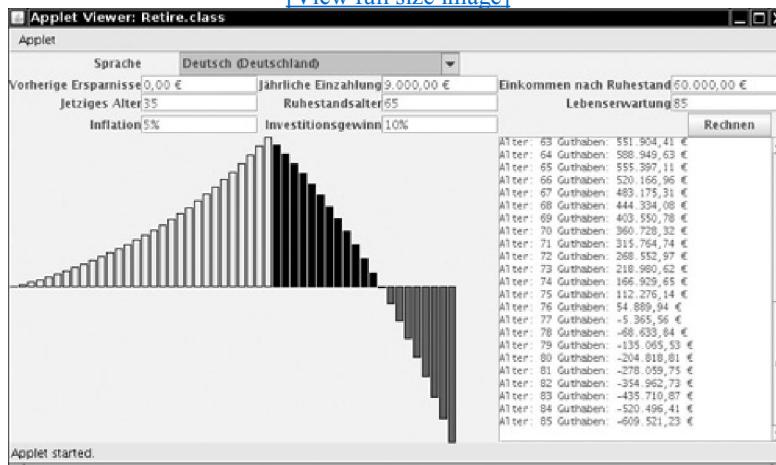
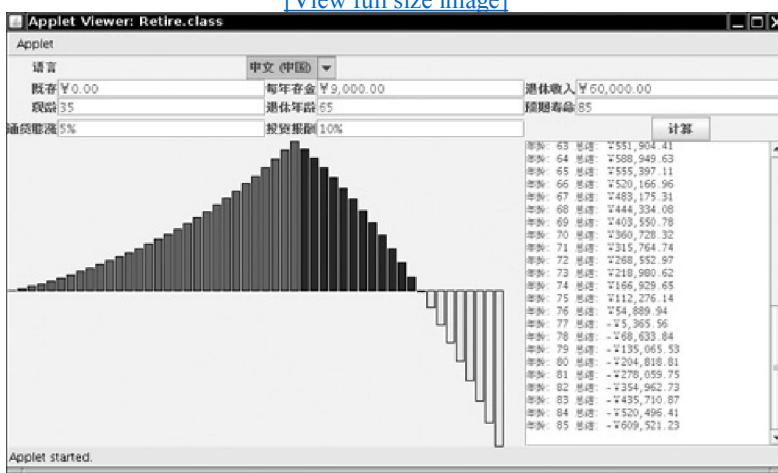


Figure 5-6. The retirement calculator in Chinese

[\[View full size image\]](#)

Listing 5-5. Retire.java

Code View:

```

1. import java.awt.*;
2. import java.awt.event.*;
3. import java.awt.geom.*;
4. import java.util.*;
5. import java.text.*;
6. import javax.swing.*;
7.
8. /**
9.  * This applet shows a retirement calculator. The UI is displayed in English, German,
10. * and Chinese.
11. * @version 1.22 2007-07-25
12. * @author Cay Horstmann
13. */
14. public class Retire extends JApplet
15. {
16.     public void init()
17.     {
18.         EventQueue.invokeLater(new Runnable()
19.         {
20.             public void run()
21.             {
22.                 initUI();
23.             }
24.         });
25.     }
26.
27.     public void initUI()
28.     {
29.         setLayout(new GridBagLayout());
30.         add(languageLabel, new GBC(0, 0).setAnchor(GBC.EAST));
31.         add(savingsLabel, new GBC(0, 1).setAnchor(GBC.EAST));
32.         add(contribLabel, new GBC(2, 1).setAnchor(GBC.EAST));
33.         add(incomeLabel, new GBC(4, 1).setAnchor(GBC.EAST));
34.         add(currentAgeLabel, new GBC(0, 2).setAnchor(GBC.EAST));
35.         add(retireAgeLabel, new GBC(2, 2).setAnchor(GBC.EAST));
36.         add(deathAgeLabel, new GBC(4, 2).setAnchor(GBC.EAST));
37.         add(inflationPercentLabel, new GBC(0, 3).setAnchor(GBC.EAST));
38.         add(investPercentLabel, new GBC(2, 3).setAnchor(GBC.EAST));
39.         add(localeCombo, new GBC(1, 0, 3, 1));
40.         add(savingsField, new GBC(1, 1).setWeight(100, 0).setFill(GBC.HORIZONTAL));
41.         add(contribField, new GBC(3, 1).setWeight(100, 0).setFill(GBC.HORIZONTAL));
42.         add(incomeField, new GBC(5, 1).setWeight(100, 0).setFill(GBC.HORIZONTAL));
43.         add(currentAgeField, new GBC(1, 2).setWeight(100, 0).setFill(GBC.HORIZONTAL));
44.         add(retireAgeField, new GBC(3, 2).setWeight(100, 0).setFill(GBC.HORIZONTAL));
45.         add(deathAgeField, new GBC(5, 2).setWeight(100, 0).setFill(GBC.HORIZONTAL));
46.         add(inflationPercentField, new GBC(1, 3).setWeight(100, 0).setFill(GBC.HORIZONTAL));
47.         add(investPercentField, new GBC(3, 3).setWeight(100, 0).setFill(GBC.HORIZONTAL));
48.         add(retireCanvas, new GBC(0, 4, 4, 1).setWeight(100, 100).setFill(GBC.BOTH));
49.         add(new JScrollPane(retireText), new GBC(4, 4, 2, 1).setWeight(0, 100).setFill(GBC.BOTH));
50.

```

```
51.     computeButton.setName("computeButton");
52.     computeButton.addActionListener(new ActionListener()
53.     {
54.         public void actionPerformed(ActionEvent event)
55.         {
56.             getInfo();
57.             updateData();
58.             updateGraph();
59.         }
60.     });
61.     add(computeButton, new GBC(5, 3));
62.
63.     retireText.setEditable(false);
64.     retireText.setFont(new Font("Monospaced", Font.PLAIN, 10));
65.
66.     info.setSavings(0);
67.     info.setContrib(9000);
68.     info.setIncome(60000);
69.     info.setCurrentAge(35);
70.     info.setRetireAge(65);
71.     info.setDeathAge(85);
72.     info.setInvestPercent(0.1);
73.     info.setInflationPercent(0.05);
74.
75.     int localeIndex = 0; // US locale is default selection
76.     for (int i = 0; i < locales.length; i++)
77.         // if current locale one of the choices, select it
78.         if (getLocale().equals(locales[i])) localeIndex = i;
79.     setCurrentLocale(locales[localeIndex]);
80.
81.     localeCombo.addActionListener(new ActionListener()
82.     {
83.         public void actionPerformed(ActionEvent event)
84.         {
85.             setCurrentLocale((Locale) localeCombo.getSelectedItem());
86.             validate();
87.         }
88.     });
89. }
90.
91. /**
92. * Sets the current locale.
93. * @param locale the desired locale
94. */
95. public void setCurrentLocale(Locale locale)
96. {
97.     currentLocale = locale;
98.     localeCombo.setSelectedItem(currentLocale);
99.     localeCombo.setLocale(currentLocale);
100. }
101. res = ResourceBundle.getBundle("RetireResources", currentLocale);
102. resStrings = ResourceBundle.getBundle("RetireStrings", currentLocale);
103. currencyFmt = NumberFormat.getCurrencyInstance(currentLocale);
104. numberFmt = NumberFormat.getNumberInstance(currentLocale);
105. percentFmt = NumberFormat.getPercentInstance(currentLocale);
106.
107.     updateDisplay();
108.     updateInfo();
109.     updateData();
110.     updateGraph();
111. }
112.
113. /**
114. * Updates all labels in the display.
115. */
116. public void updateDisplay()
117. {
118.     languageLabel.setText(resStrings.getString("language"));
119.     savingsLabel.setText(resStrings.getString("savings"));
120.     contribLabel.setText(resStrings.getString("contrib"));
121.     incomeLabel.setText(resStrings.getString("income"));
122.     currentAgeLabel.setText(resStrings.getString("currentAge"));
123.     retireAgeLabel.setText(resStrings.getString("retireAge"));
124.     deathAgeLabel.setText(resStrings.getString("deathAge"));
125.     inflationPercentLabel.setText(resStrings.getString("inflationPercent"));
126.     investPercentLabel.setText(resStrings.getString("investPercent"));
127.     computeButton.setText(resStrings.getString("computeButton"));
128. }
129.
130. /**
131. * Updates the information in the text fields.
132. */
```

```
133.     public void updateInfo()
134.     {
135.         savingsField.setText(currencyFmt.format(info.getSavings()));
136.         contribField.setText(currencyFmt.format(info.getContrib()));
137.         incomeField.setText(currencyFmt.format(info.getIncome()));
138.         currentAgeField.setText(numberFmt.format(info.getCurrentAge()));
139.         retireAgeField.setText(numberFmt.format(info.getRetireAge()));
140.         deathAgeField.setText(numberFmt.format(info.getDeathAge()));
141.         investPercentField.setText(percentFmt.format(info.getInvestPercent()));
142.         inflationPercentField.setText(percentFmt.format(info.getInflationPercent()));
143.     }
144.
145.     /**
146.      * Updates the data displayed in the text area.
147.      */
148.     public void updateData()
149.     {
150.         retireText.setText("");
151.         MessageFormat retireMsg = new MessageFormat("");
152.         retireMsg.setLocale(currentLocale);
153.         retireMsg.applyPattern(resStrings.getString("retire"));
154.
155.         for (int i = info.getCurrentAge(); i <= info.getDeathAge(); i++)
156.         {
157.             Object[] args = { i, info.getBalance(i) };
158.             retireText.append(retireMsg.format(args) + "\n");
159.         }
160.     }
161.
162.     /**
163.      * Updates the graph.
164.      */
165.     public void updateGraph()
166.     {
167.         retireCanvas.setColorPre((Color) res.getObject("colorPre"));
168.         retireCanvas.setColorGain((Color) res.getObject("colorGain"));
169.         retireCanvas.setColorLoss((Color) res.getObject("colorLoss"));
170.         retireCanvas.setInfo(info);
171.         repaint();
172.     }
173.
174.     /**
175.      * Reads the user input from the text fields.
176.      */
177.     public void getInfo()
178.     {
179.         try
180.         {
181.             info.setSavings(currencyFmt.parse(savingsField.getText()).doubleValue());
182.             info.setContrib(currencyFmt.parse(contribField.getText()).doubleValue());
183.             info.setIncome(currencyFmt.parse(incomeField.getText()).doubleValue());
184.             info.setCurrentAge(numberFmt.parse(currentAgeField.getText()).intValue());
185.             info.setRetireAge(numberFmt.parse(retireAgeField.getText()).intValue());
186.             info.setDeathAge(numberFmt.parse(deathAgeField.getText()).intValue());
187.             info.setInvestPercent(percentFmt.parse(investPercentField.getText()).doubleValue());
188.             info.setInflationPercent(percentFmt.parse(
189.                 inflationPercentField.getText()).doubleValue());
190.         }
191.         catch (ParseException e)
192.         {
193.         }
194.     }
195.
196.     private JTextField savingsField = new JTextField(10);
197.     private JTextField contribField = new JTextField(10);
198.     private JTextField incomeField = new JTextField(10);
199.     private JTextField currentAgeField = new JTextField(4);
200.     private JTextField retireAgeField = new JTextField(4);
201.     private JTextField deathAgeField = new JTextField(4);
202.     private JTextField inflationPercentField = new JTextField(6);
203.     private JTextField investPercentField = new JTextField(6);
204.     private JTextArea retireText = new JTextArea(10, 25);
205.     private RetireCanvas retireCanvas = new RetireCanvas();
206.     private JButton computeButton = new JButton();
207.     private JLabel languageLabel = new JLabel();
208.     private JLabel savingsLabel = new JLabel();
209.     private JLabel contribLabel = new JLabel();
210.     private JLabel incomeLabel = new JLabel();
211.     private JLabel currentAgeLabel = new JLabel();
212.     private JLabel retireAgeLabel = new JLabel();
213.     private JLabel deathAgeLabel = new JLabel();
214.     private JLabel inflationPercentLabel = new JLabel();
```

```
215.     private JLabel investPercentLabel = new JLabel();
216.
217.     private RetireInfo info = new RetireInfo();
218.
219.     private Locale[] locales = { Locale.US, Locale.CHINA, Locale.GERMANY };
220.     private Locale currentLocale;
221.     private JComboBox localeCombo = new LocaleCombo(locales);
222.     private ResourceBundle res;
223.     private ResourceBundle resStrings;
224.     private NumberFormat currencyFmt;
225.     private NumberFormat numberFmt;
226.     private NumberFormat percentFmt;
227. }
228.
229. /**
230. * The information required to compute retirement income data.
231. */
232. class RetireInfo
233. {
234.     /**
235.      * Gets the available balance for a given year.
236.      * @param year the year for which to compute the balance
237.      * @return the amount of money available (or required) in that year
238.      */
239.     public double getBalance(int year)
240.     {
241.         if (year < currentAge) return 0;
242.         else if (year == currentAge)
243.         {
244.             age = year;
245.             balance = savings;
246.             return balance;
247.         }
248.         else if (year == age) return balance;
249.         if (year != age + 1) getBalance(year - 1);
250.         age = year;
251.         if (age < retireAge) balance += contrib;
252.         else balance -= income;
253.         balance = balance * (1 + (investPercent - inflationPercent));
254.         return balance;
255.     }
256.
257.     /**
258.      * Gets the amount of prior savings.
259.      * @return the savings amount
260.      */
261.     public double getSavings()
262.     {
263.         return savings;
264.     }
265.
266.     /**
267.      * Sets the amount of prior savings.
268.      * @param newValue the savings amount
269.      */
270.     public void setSavings(double newValue)
271.     {
272.         savings = newValue;
273.     }
274.
275.     /**
276.      * Gets the annual contribution to the retirement account.
277.      * @return the contribution amount
278.      */
279.     public double getContrib()
280.     {
281.         return contrib;
282.     }
283.
284.     /**
285.      * Sets the annual contribution to the retirement account.
286.      * @param newValue the contribution amount
287.      */
288.     public void setContrib(double newValue)
289.     {
290.         contrib = newValue;
291.     }
292.
293.     /**
294.      * Gets the annual income.
295.      * @return the income amount
296.      */
```

```
297.     public double getIncome()
298.     {
299.         return income;
300.     }
301.
302.    /**
303.     * Sets the annual income.
304.     * @param newValue the income amount
305.     */
306.    public void setIncome(double newValue)
307.    {
308.        income = newValue;
309.    }
310.
311.    /**
312.     * Gets the current age.
313.     * @return the age
314.     */
315.    public int getCurrentAge()
316.    {
317.        return currentAge;
318.    }
319.
320.    /**
321.     * Sets the current age.
322.     * @param newValue the age
323.     */
324.    public void setCurrentAge(int newValue)
325.    {
326.        currentAge = newValue;
327.    }
328.
329.    /**
330.     * Gets the desired retirement age.
331.     * @return the age
332.     */
333.    public int getRetireAge()
334.    {
335.        return retireAge;
336.    }
337.
338.    /**
339.     * Sets the desired retirement age.
340.     * @param newValue the age
341.     */
342.    public void setRetireAge(int newValue)
343.    {
344.        retireAge = newValue;
345.    }
346.
347.    /**
348.     * Gets the expected age of death.
349.     * @return the age
350.     */
351.    public int getDeathAge()
352.    {
353.        return deathAge;
354.    }
355.
356.    /**
357.     * Sets the expected age of death.
358.     * @param newValue the age
359.     */
360.    public void setDeathAge(int newValue)
361.    {
362.        deathAge = newValue;
363.    }
364.
365.    /**
366.     * Gets the estimated percentage of inflation.
367.     * @return the percentage
368.     */
369.    public double getInflationPercent()
370.    {
371.        return inflationPercent;
372.    }
373.
374.    /**
375.     * Sets the estimated percentage of inflation.
376.     * @param newValue the percentage
377.     */
378.    public void setInflationPercent(double newValue)
```

```
379.     {
380.         inflationPercent = newValue;
381.     }
382.
383.    /**
384.     * Gets the estimated yield of the investment.
385.     * @return the percentage
386.     */
387.    public double getInvestPercent()
388.    {
389.        return investPercent;
390.    }
391.
392.    /**
393.     * Sets the estimated yield of the investment.
394.     * @param newValue the percentage
395.     */
396.    public void setInvestPercent(double newValue)
397.    {
398.        investPercent = newValue;
399.    }
400.
401.    private double savings;
402.    private double contrib;
403.    private double income;
404.    private int currentAge;
405.    private int retireAge;
406.    private int deathAge;
407.    private double inflationPercent;
408.    private double investPercent;
409.
410.    private int age;
411.    private double balance;
412. }
413.
414. /**
415.  * This panel draws a graph of the investment result.
416. */
417. class RetireCanvas extends JPanel
418. {
419.     public RetireCanvas()
420.     {
421.         setSize(PANEL_WIDTH, PANEL_HEIGHT);
422.     }
423.
424. /**
425.  * Sets the retirement information to be plotted.
426.  * @param newInfo the new retirement info.
427. */
428. public void setInfo(RetireInfo newInfo)
429. {
430.     info = newInfo;
431.     repaint();
432. }
433.
434. public void paintComponent(Graphics g)
435. {
436.     Graphics2D g2 = (Graphics2D) g;
437.     if (info == null) return;
438.
439.     double minValue = 0;
440.     double maxValue = 0;
441.     int i;
442.     for (i = info.getCurrentAge(); i <= info.getDeathAge(); i++)
443.     {
444.         double v = info.getBalance(i);
445.         if (minValue > v) minValue = v;
446.         if (maxValue < v) maxValue = v;
447.     }
448.     if (maxValue == minValue) return;
449.
450.     int barWidth = getWidth() / (info.getDeathAge() - info.getCurrentAge() + 1);
451.     double scale = getHeight() / (maxValue - minValue);
452.
453.     for (i = info.getCurrentAge(); i <= info.getDeathAge(); i++)
454.     {
455.         int x1 = (i - info.getCurrentAge()) * barWidth + 1;
456.         int y1;
457.         double v = info.getBalance(i);
458.         int height;
459.         int yOrigin = (int) (maxValue * scale);
460.     }
}
```

```
461.         if (v >= 0)
462.         {
463.             y1 = (int) ((maxValue - v) * scale);
464.             height = yOrigin - y1;
465.         }
466.         else
467.         {
468.             y1 = yOrigin;
469.             height = (int) (-v * scale);
470.         }
471.
472.         if (i < info.getRetireAge()) g2.setPaint(colorPre);
473.         else if (v >= 0) g2.setPaint(colorGain);
474.         else g2.setPaint(colorLoss);
475.         Rectangle2D bar = new Rectangle2D.Double(x1, y1, barWidth - 2, height);
476.         g2.fill(bar);
477.         g2.setPaint(Color.black);
478.         g2.draw(bar);
479.     }
480. }
481.
482. /**
483. * Sets the color to be used before retirement.
484. * @param color the desired color
485. */
486. public void setColorPre(Color color)
487. {
488.     colorPre = color;
489.     repaint();
490. }
491.
492. /**
493. * Sets the color to be used after retirement while the account balance is positive.
494. * @param color the desired color
495. */
496. public void setColorGain(Color color)
497. {
498.     colorGain = color;
499.     repaint();
500. }
501.
502. /**
503. * Sets the color to be used after retirement when the account balance is negative.
504. * @param color the desired color
505. */
506. public void setColorLoss(Color color)
507. {
508.     colorLoss = color;
509.     repaint();
510. }
511.
512. private RetireInfo info = null;
513. private Color colorPre;
514. private Color colorGain;
515. private Color colorLoss;
516. private static final int PANEL_WIDTH = 400;
517. private static final int PANEL_HEIGHT = 200;
518. }
```

Listing 5-6. `RetireResources.java`

Code View:

```
1. import java.awt.*;
2.
3. /**
4. * These are the English non-string resources for the retirement calculator.
5. * @version 1.21 2001-08-27
6. * @author Cay Horstmann
7. */
8. public class RetireResources extends java.util.ListResourceBundle
9. {
10.     public Object[][] getContents()
```

```
11.     {
12.         return contents;
13.     }
14.
15.    static final Object[][] contents = {
16.        // BEGIN LOCALIZE
17.        { "colorPre", Color.blue }, { "colorGain", Color.white }, { "colorLoss", Color.red }
18.        // END LOCALIZE
19.    };
20. }
```

Listing 5-7. RetireResources_de.java

Code View:

```
1. import java.awt.*;
2.
3. /**
4.  * These are the German non-string resources for the retirement calculator.
5.  * @version 1.21 2001-08-27
6.  * @author Cay Horstmann
7. */
8. public class RetireResources_de extends java.util.ListResourceBundle
9. {
10.    public Object[][] getContents()
11.    {
12.        return contents;
13.    }
14.
15.    static final Object[][] contents = {
16.        // BEGIN LOCALIZE
17.        { "colorPre", Color.yellow }, { "colorGain", Color.black }, { "colorLoss", Color.red }
18.        // END LOCALIZE
19.    };
20. }
```

Listing 5-8. RetireResources_zh.java

Code View:

```
1. import java.awt.*;
2.
3. /**
4.  * These are the Chinese non-string resources for the retirement calculator.
5.  * @version 1.21 2001-08-27
6.  * @author Cay Horstmann
7. */
8. public class RetireResources_zh extends java.util.ListResourceBundle
9. {
10.    public Object[][] getContents()
11.    {
12.        return contents;
13.    }
14.
15.    static final Object[][] contents = {
16.        // BEGIN LOCALIZE
17.        { "colorPre", Color.red }, { "colorGain", Color.blue }, { "colorLoss", Color.yellow }
18.        // END LOCALIZE
19.    };
20. }
```

Listing 5-9. RetireStrings.properties

```

1. language=Language
2. computeButton=Compute
3. savings=Prior Savings
4. contrib=Annual Contribution
5. income=Retirement Income
6. currentAge=Current Age
7. retireAge=Retirement Age
8. deathAge=Life Expectancy
9. inflationPercent=Inflation
10. investPercent=Investment Return
11. retire=Age: {0,number} Balance: {1,number,currency}

```

Listing 5-10. RetireStrings_de.properties

```

1. language=Sprache
2. computeButton=Rechnen
3. savings=Vorherige Ersparnisse
4. contrib=J\u00e4hrliche Einzahlung
5. income=Einkommen nach Ruhestand
6. currentAge=Jetziges Alter
7. retireAge=Ruhestandsalter
8. deathAge=Lebenserwartung
9. inflationPercent=Inflation
10. investPercent=Investitionsgewinn
11. retire=Alter: {0,number} Guthaben: {1,number,currency}

```

Listing 5-11. RetireStrings_zh.properties

```

1. language=\u8bed\u8a00
2. computeButton=\u8ba1\u7b97
3. savings=\u65e2\u5b58
4. contrib=\u6bcf\u5e74\u5b58\u91d1
5. income=\u9000\u4f11\u6536\u5165
6. currentAge=\u73b0\u9f84
7. retireAge=\u9000\u4f11\u5e74\u9f84
8. deathAge=\u9884\u671f\u5bff\u547d
9. inflationPercent=\u901a\u8d27\u81a8\u6da8
10. investPercent=\u6295\u8d44\u62a5\u916c
11. retire=\u5e74\u9f84: {0,number} \u603b\u7ed3: {1,number,currency}

```



java.applet.Applet 1.0

- Locale getLocale() 1.1

gets the current locale of the applet. The current locale is determined from the client computer that executes the applet.

You have now seen how to use the internationalization features of the Java language. You use resource bundles to provide translations into multiple languages, and you use formatters and collators for locale-specific text processing.

In the next chapter, we delve into advanced Swing programming.

